

## Table des matières

<b>1 Généralités</b>	<b>2</b>
<b>2 Abstract</b>	<b>2</b>
<b>3 Résumé</b>	<b>2</b>
<b>4 Remerciements</b>	<b>3</b>
<b>5 Introduction</b>	<b>4</b>
5.0.1 Architecture ARM . . . . .	5
5.0.2 Architecture RISC-V . . . . .	5
5.0.3 Field Programmable Gate Array (FPGA) . . . . .	6
<b>6 Le projet EVEEX</b>	<b>7</b>
6.1 Définition des exigences . . . . .	7
6.2 Maîtrise de l'algorithme . . . . .	9
6.3 Développement de l'algorithme . . . . .	13
6.3.1 Haut Niveau : code en python . . . . .	13
6.3.2 Bas niveau : code en C . . . . .	14
6.3.3 Langage moderne : Golang (Go) . . . . .	14
6.3.4 La problématique du VHDL . . . . .	15
6.4 Implémentation sur l'embarqué . . . . .	17
6.4.1 Alternative 1 : FPGA . . . . .	17
6.4.2 Alternative 2 : ARM . . . . .	17
6.4.3 Alternative 3 : RISCV . . . . .	17
6.5 Déroulement agile du projet . . . . .	17
<b>7 Conclusion</b>	<b>19</b>
<b>8 Annexes</b>	<b>21</b>
<b>9 Bibliographie</b>	<b>22</b>
<b>10 Glossaire</b>	<b>23</b>

# 1 Généralités

## Membres du groupe :

- Guillaume Leinen (guillaume.leinen@ensta-bretagne.org)
- Jean-Noël Clink (jean-noel.clink@ensta-bretagne.org)
- Hussein Saad (hussein.saad@ensta-bretagne.org)
- Alexandre Froehlich (alexandre.froehlich@ensta-bretagne.org)
- Hugo Questroy (hugo.questroy@ensta-bretagne.org)

## Encadrants du projet :

- Pascal Cotret (pascal.cotret@ensta-bretagne.fr)
- Jean-Christophe Le Lann (jean-christophe.le\_lann@ensta-bretagne.fr)
- Joël Champeau (joel.champeau@ensta-bretagne.fr)

## Liens utiles :

- Code source du projet : <https://github.com/EVEEX-Project/EVEEX-Code>
- Code source de la documentation : <https://github.com/EVEEX-Project/EVEEX-Documentation>
- Eportfolio Mahara : <https://mahara.ensta-bretagne.fr/view/groupviews.php?group=348>

# 2 Abstract

# 3 Résumé

## 4 Remerciements

Nous n'aurions jamais pu atteindre le niveau de compétences et remplir nos objectifs sans l'aide des acteurs suivants :

- 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'à 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.

Ainsi nous remercions toutes ces personnes qui ont contribué à notre projet ou bien qui nous ont guidé vers des chemins que nous n'aurions jamais pensé réalisables.

## 5 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** et **résolutions** de lecture augmentent elles aussi, ainsi que la part de "**live**" (flux direct) très gourmands en bande passante.

Dans cette perspective, il est évident qu'un **algorithme de compression** pour compresser un flux vidéo brut à toute son importance. Le moindre pourcent de **bande passante** économisée permet de libérer plusieurs TB/s (terabyte par seconde) de bande passante sans compter l'impact écologique lié à une consommation électrique moins importante. Cependant il existe une multitude d'algorithmes de compression, appelés "codecs". Ces **codecs** sont relativement peu connus du grand public, en voici une liste non exhaustive des plus répandus :

- **MPEG-4** (H.264) : il s'agit d'un des codecs les plus connus. Il génère des fichiers d'extension **.mp4** et est embarqué dans un grand nombre d'appareils numériques. Il est important de savoir que, comme le H.265 (vu plus loin), ce codec est **protégé** par un brevet, et les services et constructeurs souhaitant utiliser cet algorithme ou un connecteur basé sur l'algorithme doivent verser des royalties à MPEG-LA [2] (*La fabrication d'un connecteur display-port coûtant 0.20\$ de licence au constructeur*), coalitions de plusieurs entreprises du numérique comme Sony, Panasonic ou encore l'université de Columbia.
- **VPx** : appartenant à l'origine à Onetechnology, l'entreprise fut rachetée par Google à la suite. Les codecs VP (dernière version VP9) sont ouverts et sans royalties. Ce codec est plutôt performant en terme de compression mais son encodage est lent [3]. Par exemple avec un processeur core i7 d'intel en 720p, la vitesse de compression s'approche de 2 images par seconde, occasionnant des coûts non négligeables en puissance informatique pour les entreprises productrices de contenu (comme Netflix ou bien YouTube).
- **H.265** : l'un des codecs les plus récents, et digne successeur du H.264. Il permet une réduction significative de la bande passante nécessaire au flux vidéo, notamment pour le streaming. Cependant à l'instar du VP9, le H.265 est aussi lent à l'encodage, et demande en plus de verser des royalties de licence.

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, à but non lucratif, afin de développer un codec libre de droit aussi efficace que les autres : l'**Alliance 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 H.265 [5] mais occasionne, par les différentes bibliothèques utilisées, une utili-

sation des ressources informatiques (puissance processeur) bien plus importante, aussi bien du côté **encodeur** que **décodeur**. La transition vers l'AV1 sur les grandes plateformes vidéos (Netflix, Youtube) n'est pas encore effectif mais est déjà en cours de réalisation.

Ce besoin en ressources CPU devient critique et un point économiquement important pour les entreprises du secteur multimédia. A l'heure actuelle, l'**architecture PC** (jeu d'instruction x64, x86) reste la plus utilisée dans l'informatique moderne, mais cela pourrait changer d'ici quelques années. En effet, les architectures à destination de matériel embarqué, ont fait d'immenses progrès ces dernières années, au point que même un géant du secteur comme Apple décide de basculer l'ensemble de ses produits vers une **architecture ARM** (vu plus loin).

Ces architectures embarquées, plus récentes et profitant des avancées modernes, possèdent en effet des caractéristiques de **consommation** et de **performances** qui les rendent très intéressantes pour le traitement vidéo. Voyons ensemble quelques unes de ces architectures et technologies actuelles :

### 5.0.1 Architecture ARM

Ce jeu d'instruction est très présent dans les appareils mobiles et embarqués comme les smartphones ou tablettes. Il a l'avantage de proposer un jeu d'instruction réduit (RISC : Reduced Instruction Set Computer), 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** (x64 pour AMD, x86 pour Intel). Les SOC (System On Chip) ARM **embarquent** tous les composants nécessaires au fonctionnement du système (CPU : Computer Processing Unit, GPU : Graphical Processing Unit, DSP : Digital Signal Processor, gestions des I/O : Input/Output) sur une seule puce ce qui rend les systèmes extrêmement compacts. Le jeu d'instructions réduit en revanche l'**inter-compatibilité** entre x64/86 et les architectures de type RISC comme l'ARM ou le RISC-V (vu plus loin).

### 5.0.2 Architecture RISC-V

L'**architecture RISC-V** est très proche de l'ARM, qui est aussi un RISC. Historiquement la création du RISC-V est motivé par la volonté de ne pas "à trainer" du code compatible avec des machines vieilles de 20 ans, afin de proposer un système plus moderne, plus léger et plus simple à maintenir. La différence profonde se situe dans l'adoption de la technologie dans l'industrie. ARM est présent dans une écrasante majorité des applications par rapport au RISC-V, ce qui implique une faible quantité de code existant pour ce dernier. En revanche, RISC-V possède un avantage notable : elle est complètement **Open-Source et libre de droits**.

### 5.0.3 Field Programmable Gate Array (FPGA)

L'**architecture FPGA** est complètement différente des autres citées précédemment. Un processeur ARM ou x86 est gravé sur une puce en silicium et inaltérable dans son architecture, 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 quelques éléments :

- Comme on peut reprogrammer les portes logiques qui le constituent, 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 dispose 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 tâches simples et répétitives comme le minage de cryptomonnaies.
- En revanche, l'appréhension d'une telle technologie est loin d'être aisée, le **développement** avec des Hardware Description Language (HDL) tel que le Verilog ou le VHDL est **loin d'être facile**, il n'existe que peu d'abstraction dans le langage et son développement. Enfin la formation, l'apprentissage et le temps de développement sont bien plus longs sur une telle puce en comparaison avec le développement classique sur architecture x86 ou ARM.
- 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 et un point de travail important qui n'est pas nécessairement reconnu de tous.

Afin de répondre à cette problématique nouvelle, l'ENSTA Bretagne souhaite développer un algorithme de compression vidéo, qui soit **open-source** et doté de performances convaincantes (définies par la suite). Dans un second temps cet algorithme sera implémenté sur un système embarqué. La réalisation et l'implémentation de cet algorithme constituent notre travail pour ce projet et se nomme **EVEEX** (projet **E**ncodage **V**idéo **E**NSTA Bretagne **E**Xpérimental).

## 6 Le projet EVEEX

### 6.1 Définition des exigences

Afin de définir clairement nos objectifs pour ce projet, il est primordial d'être capable de définir et d'extraire les exigences à partir des souhaits de notre client : l'ENSTA Bretagne, que ces exigences 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. Nos exigences tourneront donc autour de ce flux de travail passant d'un encodeur sur un premier système à un décodeur sur un second système.

**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 langage est un point crucial dans la réalisation du projet.

Nous avons donc défini un certain nombre d'exigences avec les performances souhaitées 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

Numéro identifiant l'exigence	Exigence	Performances attendues
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
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 (i.e. l'algorithme EVEEX).



## 6.2 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 faits sur cet algorithme seront déterminants en termes de performances et de facilité d'implémentation, le niveau des membres de l'équipe étant hétérogène.

Pour cet algorithme de traitement des données, nous nous sommes inspirés du MJPEG et ce pour plusieurs raisons. Il est relativement simple à appréhender, son fonctionnement est connu et répandu, la documentation technique est disponible en ligne. **L'objectif de notre algorithme est de compresser l'image de référence le plus possible** (i.e. obtenir 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 caméras), est tout d'abord **convertie au format chrominance/luminance (YUV)**. Ce choix est motivé par le fait qu'au format YUV, l'essentiel de l'information de l'image se trouve dans la luminance. Les chrominances pourront alors être approximées pour gagner en espace mémoire, on parlera ici de **compression avec pertes**.

Insérer image RGB/YUV (issu de la VFU)

Ensuite, l'image est découpée en **macrobloques** de  $16 \times 16$  pixels. En réalité, comme une image RGB contient 3 canaux de couleur, les macrobloques sont en fait de taille  $16 \times 16 \times 3$ , mais, par abus de langage, et par souci de simplicité, nous dirons simplement qu'ils ont une taille de  $16 \times 16$  (ou  $N \times N$  dans le cas général). Cette taille de macrobloc 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 macrobloques  $16 \times 16$  étaient ceux qui produisaient les meilleurs taux de compression parmi les tailles standards de macrobloques, à savoir  $8 \times 8$ ,  $16 \times 16$  et  $32 \times 32$  pixels, pour un temps donné. Cette décomposition en macrobloques 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 macrobloques **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.

Insérer image macrobloc vs image complète (en mode wikipedia)

Après cette étape, on applique diverses transformations **à chacune de ces matrices-macrobloques 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  $16 \times 16 = 256$ . **On passe donc d'une matrice à 2 dimensions à une liste en une seule dimension.** Ce découpage va se faire selon les  $2 \times 16 - 1 = 31$  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 représentatives non-négligeables, et puis, partout ailleurs, ces valeurs seront moindres.

insérer image linéarisation zig zag

- 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_THRESHOLD` 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-macrobloccs générées aléatoirement. **On a choisi `threshold = 10`, car il s'agissait de la valeur maximale qui permet subjectivement d'avoir une bonne qualité d'image en sortie.** Il est important de noter que cette étape de seuillage est **irréversible**, on parle ici d'une étape de traitement avec pertes car on perd de l'information dans les détails.
- 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 macrobloccs, sera alors une liste de listes de tuples RLE.**

La partie suivante concerne le formatage des données. On utilise pour cela un **encodage de Huffman** qui permet à la fois de compresser et de formater les données en utilisant un arbre binaire afin d'obtenir un encodage plus petit que l'encodage naïf, mais surtout un **encodage non ambiguë**. On appellera

la trame à transmettre un **bitstream**.

image arbre huffman

```

1 Encoded string :
2 1010100110011000001100110110111001001101010000101000100101111
3 10101111100000101011001110101110000001001001101011111110101110
4 10100111110011000101111101110101011001011101100110010010011011
5 11000011111001000010
6 String decoded back : le chic de l'ensta bretagne sur la compression
   vide

```

L'arbre se base sur la récurrence des caractères dans une phrase afin de les ordonner par fréquence 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 d'encodage 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 utilise 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 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 métadon-

nées 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 du thread d'encodage. 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 le bitstream au complet associé à l'image compressée, on va pouvoir commencer l'étape de **décodage**, qui constitue la troisième 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 (i.e. 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 dans sa version en python :

graph statistiques des performances en python

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 :

graph statistiques entre la DCT et l'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.**

### 6.3 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. Nous allons vous expliquer notre exploration dans ces différents langages en trois parties.

#### 6.3.1 Haut Niveau : code en python

La première étape consiste en un développement orienté objet, dans un langage de haut niveau. Nous avons choisi le **python** car nous avons tous ou presque de bonnes compétences et 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.
- La génération d'un bitstream en binaire car python converti directement les 0 et les 1 dans leur représentation en ASCII

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. Vous pourrez d'ailleurs retrouver ce package ici : <https://test.pypi.org/project/EVEEX/>

Le prototype python est fonctionnel et a été obtenu relativement rapidement. Voici quelques statistiques sur la performance atteinte par l'algorithme :

insérer image des statistiques en python

### 6.3.2 Bas niveau : code en C

Après un prototype fonctionne en python mais peu performant, nous voulions améliorer l'algorithme et notamment le rendre plus rapide. Notre choix c'est donc vite porté sur un langage plus bas niveau comme le **C**, avec absence d'interpréteur et un programme compilé machine. Cependant le code C présente quelques problèmes :

- l'allocation en mémoire n'est pas automatique (faite via un ensemble de `malloc()`, `calloc()` et `realloc()`)
- les types sont peu nombreux et la création de structures (`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** fourni par nos encadrants. Nous avons donc commencé la construction de types non natifs en programmation orienté objet en C avant d'attaquer l'implémentation de l'encodeur.

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 complexe 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 grâce à l'outil Valgrind.

Voici les performances que l'on a obtenu pour le prototype partiel en C :

Insérer une image sortie de console avec le temps mis

### 6.3.3 Langage moderne : Golang (Go)

Devant l'échec de la complexité d'implémentation du code en C, nous nous sommes orientés vers un langage plus facile à prendre en main, tout en disposant de performances bien supérieures au python. Nous nous sommes donc orienté sur un langage financé, promu par Google et adopté par beaucoup d'entreprises dans le monde du numérique et des serveurs : le Golang.

Ce langage, comme tout langage de programmation, apporte son lot d'avantage et d'inconvénients :

#### **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 Programmation orientée objet, tout en gardant une structure proche de la programmation fonctionnelle
- un Garbage collector intégré au langage qui se charge de libérer les variables une fois que celles-ci ne sont plus utilisées
- des outils de débogage plus poussés, le golang donne la localisation d’une erreur et est beaucoup plus verbeux que le C avec GDB
- une suite d’outils de profiling du code intégré et des bibliothèque de tests unitaires intégrés
- beaucoup de garde fou : il n’y a pas de runtime error, juste des erreurs de compilation, de plus la structure du code est codifiée ainsi il n’existe au sein d’un programme qu’un seul type de programmation, ce qui rend le code plus lisible et universel
- des outils de cross-compilation existante pour le langage, ainsi à partir de linux nous pouvons compiler pour ARM ou RISC-V

**Inconvénients :**

- taille du binaire en sortie conséquente (un programme simple affichant “Hello World!” peut faire 2Mo)
- **cross-compilation impossible en riscV32 bits**, ainsi que la HLS via vivado.

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

Le Golang possède, comme indiqué précédemment, un outil de profiling très évolué, nous pouvons donc avec simplicité trouver les éléments de l’algorithme qui prennent du temps, ou bien ne sont pas optimisés :

insérer image du profiling

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

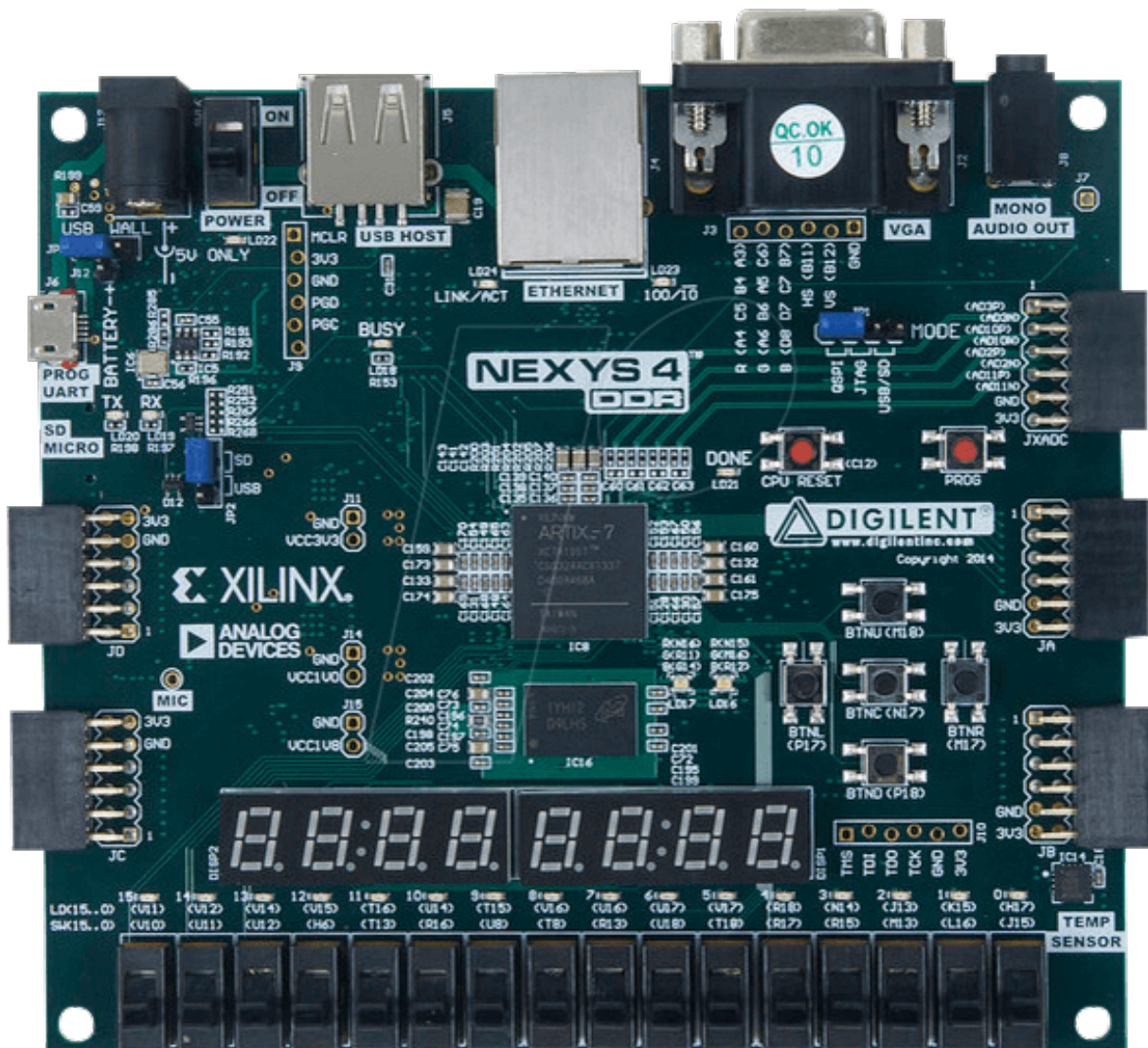
Maintenant, on constate qu’on accélère considérablement le calcul d’encodage des frames, ce qui nous prouve l’utilité du développement de Taylor ainsi que du profiling.

**6.3.4 La problématique du VHDL**

Dans nos plans initiaux, nous cherchions à implémenter le code de manière matérielle au sein d’une puce FPGA. Cela suppose d’utiliser un HDL ou *Hardware Description Language*. Les plus connus sont **Verilog** et **VHDL**, mais il en existe d’autres en python, ruby, etc...



Avant de pouvoir développer et surtout tester sur une carte directement, il nous faut installer la *Tool-chain* de développement. Les cartes dont nous disposons sont des FPGA artix-7 construit par la société Digilent, les cartes NEXYS4 DDR.



**FIG. 1 :** Carte FPGA Nexys4 DDR et ses entrées sorties

Ces cartes possèdent une DDR (Double Data Rate) embarqué, ainsi que la plupart des entrées sorties nécessaires à l'élaboration d'un prototype (ethernet, vga, p-mod pour la camera). Pour capturer l'image, nous avons à notre disposition des camera **OV7670**, capturant une image 480p et ayant l'avantage d'être très bas coût (2 euros l'unité), ce qui est utile pour en acheter plusieurs (l'une d'elles a d'ailleurs succombé à nos manipulations...).



Les premières manipulations se déroulent sans soucis, nous parvenons à afficher un aperçu de la caméra sur un écran VGA, et tout ça par du code VHDL. Cependant les choses se sont très vite compliquées quand il a fallu s'attaquer à la RAM.

Sur un FPGA, on dispose de BRAM intégrés au chip qui sont facile a utilisé mais de taille réduite (sur nos designs nous n'arrivions pas a dépasser 32 Mo de BRAM), cependant nos estimations en termes d'usage mémoire d'EVEEX dépassaient la quantité de BRAM utilisable sur un design. La DDR à l'avantage d'être sur une puce a part et d'être beaucoup plus grande (128Mo), cependant il est nécessaire de développer un contrôleur pour cette RAM qui, de l'avis même des encadrants, dépasse nos capacités de développement en école d'ingénieur. Il a donc fallu passer par une solution alternative.

La première solution envisagée fut d'utiliser Vivado HLS, l'outil de Xilinx pour la synthèse de code. Il permet moyennant un formalisme dans le code de traduire du code C vers du HDL comme vhdl.

## 6.4 Implémentation sur l'embarqué

### 6.4.1 Alternative 1 : FPGA

### 6.4.2 Alternative 2 : ARM

### 6.4.3 Alternative 3 : RISCv

## 6.5 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 équipé réduite en informatique. La première étape a été la définition des rôles au sein du projet :

- **Scrum Master** : 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 coordinateur 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 tâches 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 tâches 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 tâches sont regroupées en Issues comme lorsque l'on remonte un bug à un développeur, la différence étant qu'on peut facilement pipeliner la réalisation des issues, et les regrouper 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 :



**FIG. 2 :** Diagramme de Velocity provenant de Zenhub

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. Au sprint 5 nous avons

accueilli Hussein dans l'équipe, ce qui explique la rapide montée en travail effectué par l'équipe. Cela peut aussi être expliqué par une évaluation plus précise et réaliste de la pondération des tâches.

Nous nous sommes servis des descriptions des issues pour conserver les user-stories. Entre autres, 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 tâches en fonctions des compétences de chacun et de la confiance en la réalisation de la tâche.
- **travail réalisé** : L'issue a-t-elle été réalisée en partie ? en totalité ?
- **perspectives futures** : ce qui va découler de la réalisation de la tâche.

Nous complétons 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 : <https://mahara.ensta-bretagne.fr/view/groupviews.php?group=348>

## 7 Conclusion

La **démarche agile** a été plus que nécessaire dans ce projet. En effet nous avons rencontrés plusieurs **branches** qui se sont avérées mortes ou sans issues à 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 nous disposions pas d'un **recul suffisant** en hardware pour travailler sur une implémentation FPGA de manière efficace au début du projet. Il s'est avéré que la quasi-totalité des cours qui nous aurait été utile pendant le projet nous ont été dispensé vers la fin du projet (Architecture des ordinateurs, Traitement et Protection de l'Information, Conception Logicielle).

Premièrement, le **développement en C reste très compliqué** notamment dans le traitement d'un algorithme complexe. Si il est très performant, l'allocation en mémoire rend compliqué l'implémentation de structures à taille variable, notamment le code de Huffman (Variable Length Coding). La solution Golang est aussi très attrayante sur le papier, notamment au regard du processus de développement. Cependant la non-compatibilité de la **cross-compilation** Golang vers RISC-V 32 bits, découverte après coup, rend impossible l'implémentation sur un SOC à base de FPGA, comme on l'avait envisagé. il aurait de toute façon fallu des cartes FPGA puissantes pour placé le SOC avec sa mémoire, et profiter d'un gros parallélisme.

Enfin, nous n'avons pas saisi au départ les possibilités du FPGA en matière de programmation "objet". Nous avons pu constater pendant le cours de 2ème année de M. Le Lann sur le VHDL à quel point le langage dispose d'une certaine "abstraction" sur les types, et il est possible que si devions commencer

le projet maintenant, nous aurions cherché à développer au moins une partie du code en FPGA natif. Concernant Litex, l'approche est très séduisante, la programmation Migen est plus rapide à développer d'une manière générale et permet de synthétiser beaucoup de code à partir de quelques fichiers pythons.

Toutefois, nous avons pu nous concentrer sur la **maîtrise de l'algorithme** et les **possibilités d'intégration future** au sein du matériel.

## **8 Annexes**

## 9 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-teliviseur/av1-nouveau-standard-video-adopte-a-unanimité-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](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](https://www.xilinx.com/support/documentation/application_notes/xapp616.pdf) [Accessed 1 December 2020].

## 10 Glossaire

**Flux-Vidéo ou Streaming** : 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 Intégré.

**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 silicium intégrant plusieurs composants, comme de la mémoire, un processeur, et un composant de gestion d'entrées/sorties.