Rapport de projet S9

Optimisation et amélioration d'un récepteur LoRa pour nanosatellite

Etudiants :

ROMERO Valentin
GOUJON Clarisse
MAGNANT Matthieu
ARRIVE Alexandre

Encadrants :

Bertrand LE GAL

Optimisation et amélioration d'un récepteur LoRa pour nanosatellite

Sommaire

I. Contexte du projet

L'IoT est une révolution de notre monde moderne. Il désigne l'interconnexion de plusieurs systèmes communicants avec Internet. Ces systèmes transmettent peu de données avec de faibles puissances d'émission, mais ils sont soumis à de nombreuses contraintes de fiabilité, de consommation d'énergie, mais surtout de portée.

Les applications IoT rencontrent beaucoup de difficultés à trouver des relais afin de transmettre leurs informations. Les satellites semblent être la meilleure solution pour contrer ce problème.

Cependant, de nos jours, la couverture radio concerne moins de 30% de la surface de la Terre. Cette faible couverture représente un frein pour le développement des applications IoT localisés dans les zones isolés.

De nombreuses solutions voient le jour, tel que les constellations de satellites *Kinéis* ou *Swarm* de Eutelsat. Cependant, ces solutions utilisent des bandes de fréquences non libres, tel que la 2G, 4G ou la bande ARGOS.

L'entreprise Semtech quant à elle, propose des puces permettant de communiquer sur des bandes ISM. Les bandes ISM sont libres d'utilisation, à la seule condition de respecter une durée d'émission maximale en fonction de la puissance utilisée.

L'utilisation de la technologie Semtech pour connecter les objets IoT sur les bandes ISM pourrait être une nouvelle révolution pour le monde de l'IoT.

LoRa est un protocole de communication utilisant une bande ISM. Il permet déjà de connecter des millions d'objets IoT sur Terre, il est donc très pertinent de vouloir son implémentation dans le spatial. Cependant, certains verrous technologiques tel que l'effet Doppler ont empêchés qu'un quelconque service spatial ne l'utilise.

Ces verrous sont maintenant levés grâce à l'expertise scientifique de l'équipe de Guillaume FERRE, enseignant chercheur à l'IMS. Son équipe a conçu une communication LoRa modifiée, qui est capable d'être implémentée dans un satellite grâce à un encodage différentiel. Une chaîne de communication complète fonctionnelle a été conçue sous Matlab.

L'équipe de Bertrand LE GAL, également enseignant chercheur à l'IMS, s'est alors occupée de réaliser une implémentation logicielle en C++ à l'aide de radio-logicielles.

Les essais étant concluant, la prochaine étape était donc d'implémenter cette communication dans l'espace.

Le NAASC, une agence spatiale universitaire, a accepté d'implémenter notre communication dans un nanosatellite dont la mise en orbite sera autour de 2025.

L'expérience est nommée ELIOT. Elle propose de tirer parti d'une puce commerciale de Semtech afin de concevoir une charge utile capable de démoduler des signaux de type LoRa que nous enverrons au satellite.

Cette charge utile propose de nombreux avantages. Elle pourrait permettre l'essor d'une communication en bande libre, robuste aux interférences, avec de faibles puissances d'émission (14dBm), et permettant de connecter des millions d'objets utilisant déjà le protocole LoRa.

L'implémentation de notre communication dans un contexte spatial permettrait de réaliser une démonstration sur des distances de plusieurs centaines de kilomètres, tout en étant soumis à l'effet Doppler et à la réception simultanée de nombreuses communications.

Le satellite NanoNAASC est équipé d'un Zynq 7020 contenant un FPGA et deux cœurs ARM.

La charge utile ELIOT n'utilise qu'un seul cœur et profite d'une conception conjointe CPU/FPGA pour décoder les trames LoRa reçues en temps réel.

Le récepteur LoRa développé dispose d'une implémentation sur Zynq. Ce composant est plutôt lent, pourtant de nombreuses améliorations futures sont à prévoir. Il est donc nécessaire d'améliorer ses performances temporelles pour permettre l'ajout de ses futures améliorations sur le récepteur final.

II. Communication LoRa DCSS

Avant de présenter les travaux que nous avons réalisé, il est nécessaire de détailler rapidement le fonctionnement de la communication LoRa implémentée dans notre système.

1. Une trame LoRa

Les trames LoRa que nous utilisons sont constitués d'un préambule, d'un mot de silence et d'un payload.

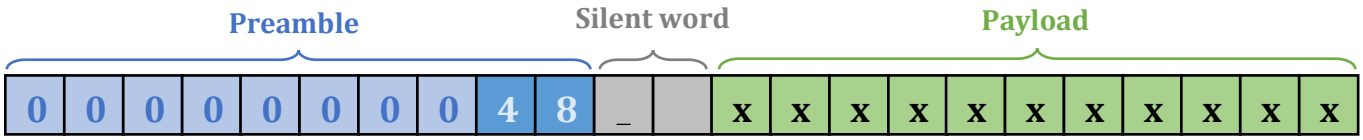


Figure 1 – Exemple d'une trame LoRa

Le préambule est constitué de 8 zéros afin de permettre la détection du début de la trame par les récepteurs. Il permet aussi de corriger le décalage temporel « STO ».

Les symboles « 4 » et « 8 » du préambule sont fixes. Ils permettent donc de détecter le décalage fréquentiel « CFO » lors de la réception.

Le mot de silence quant à lui permet de corriger le décalage temporel « STO » tout comme les zéros du préambule.

Finalement, le Payload contient les données à transmettre.

2. Modulation par étalement de spectre (CSS)

Le protocole LoRa utilise une modulation par étalement de spectre.

On génère une fréquence porteuse à 868MHz, puis on fait évoluer linéairement la fréquence de $\pm 62.5kHz$ autour de cette porteuse.

Sur la figure 2, on peut voir que la manière dont évolue la fréquence autour de la porteuse permet de communiquer un symbole différent.

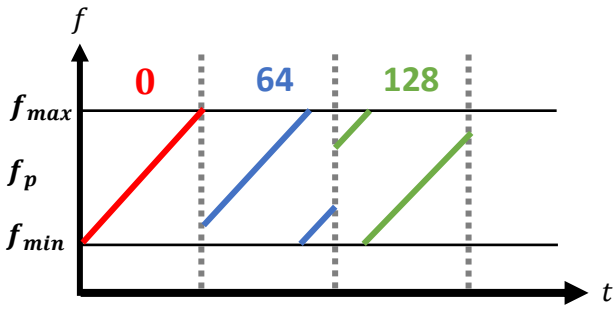


Figure 2 – Exemple d'une modulation LoRa

Le nombre de bits par symbole s'appelle le « Spreading factor » noté « SF », il est compris entre 7 et 12.

Ce facteur est très important puisqu'il va déterminer le nombre de symboles possible $M = 2^{SF}$.

Ainsi, lorsque SF est très faible, on aura un débit très important mais une communication relativement peu robuste. A contrario, pour un SF élevé, le débit sera faible mais la communication sera très robuste au bruit et aux interférences. Dans le cas du satellite, seuls les SF 11 et 12 permettent une bonne qualité de réception.

3. Encodage différentiel (DCSS)

La particularité de notre communication LoRa est l'encodage différentiel. Chaque valeur de symbole dépend du symbole précédent. Ainsi, notre communication est bien plus robuste aux décalages fréquentiels, ce qui est indispensable pour résoudre le problème d'effet Doppler engendré par la vitesse relative du satellite en orbite.

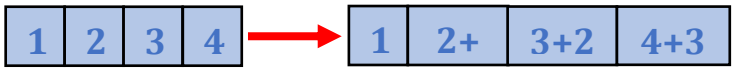


Figure 3 – Exemple d'un encodage différentiel

III. Chaine de communication

La chaîne de communication LoRa implémentée dans notre composant Zynq 7020 est présentée ci-dessous :

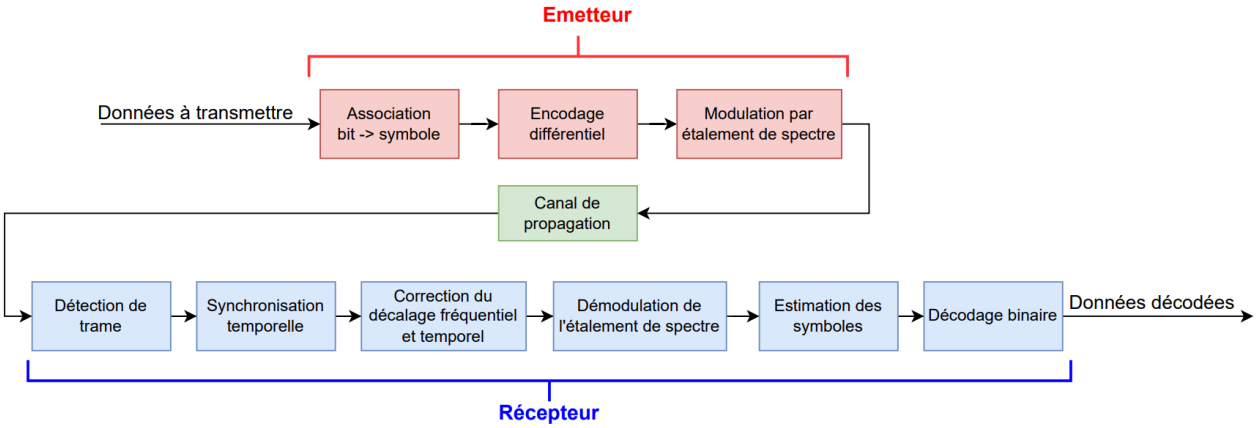


Figure 4 – Chaîne de communication LoRa théorique

Cette chaîne de communication est implémentée sur une Pynq-Z2. Elle réalise à la fois la génération et le décodage des trames

IV. Projet NanoNAASC

1. Contexte

Afin de mettre à l'épreuve la communication LoRa DCSS dans un contexte spatial, il a été décidé de travailler avec l'agence spatial universitaire NAASC.

Cette dernière travaille en collaboration avec le CNES et de nombreux établissements supérieurs de Nouvelle-Aquitaine afin d'envoyer un CubeSat à l'horizon 2025. Ce nanosatellite contiendra 4 expériences scientifiques, dont notre communication LoRa modifiée.

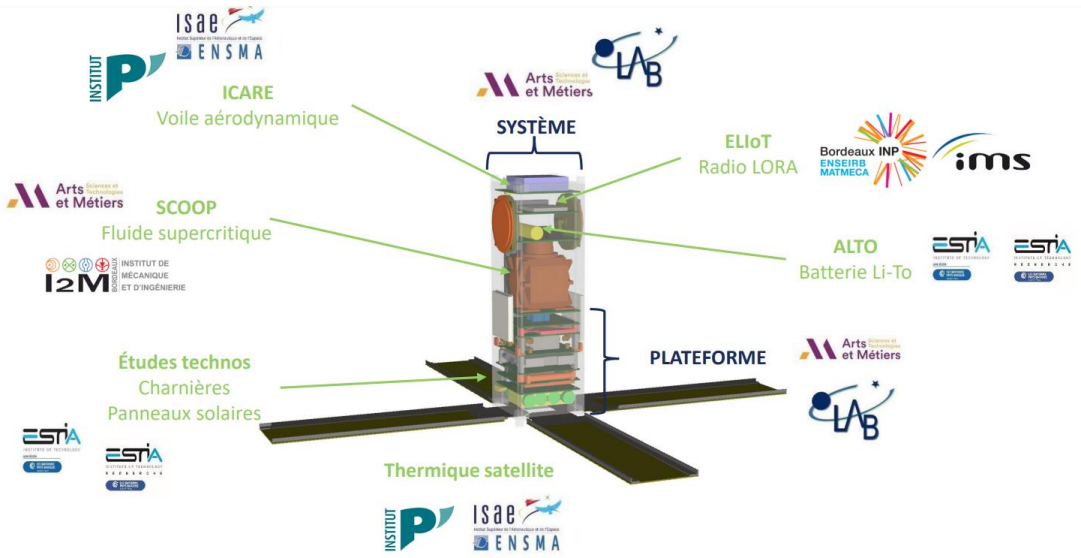


Figure 5 – Charges utiles et acteurs du projet NanoNAASC

Notre but est d'améliorer une charge utile nommée ***ELIOT*** : *Experimental Long-Range Internet of Things Network*.

Cette charge utile doit permettre au satellite de recevoir et de décoder des trames LoRa envoyées depuis la Terre.

2. Charge utile ELIOT

Le CubeSat dispose de deux antennes patch de Semtech afin de recevoir les signaux LoRa. Ces signaux sont alors convertis en couple I/Q à l'aide du récepteur radio AD9361. L'algorithme de réception implémenté dans le Zynq 7030 de la carte Ninado s'occupe alors de décoder les signaux reçus.

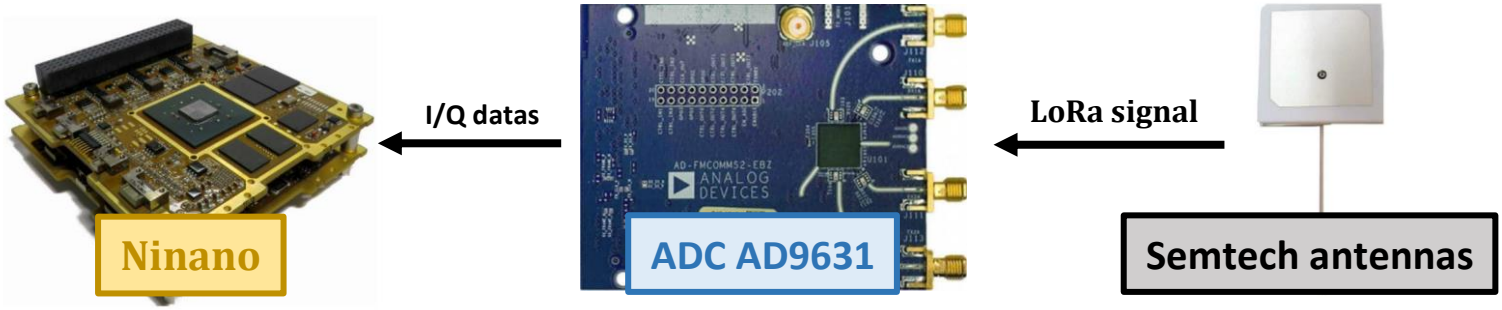


Figure 6 – Ressources matérielles de la charge ELIOT

La figure 6 permet de comprendre l'agencement de la charge ELIOT.

Tout d'abord, le CAN convertit les données reçues par l'antenne en couples IQ. Ces couples sont transmis à travers 24 canaux différentiels LVDS à la carte Ninano.

Ces 24 canaux sont alors sérialisés à l'aide d'un bloc FPGA de mise en mémoire, puis les données sont mises dans la mémoire DDR.

Le processeur ARM peut alors traiter les trames reçues.

Il peut gagner du temps d'exécution en déportant ses calculs sur les accélérateurs implémentés sur FPGA. On utilise pour cela un « AXI_DMA » qui permet de transmettre rapidement une grande quantité de données aux accélérateurs.

Et pour finir, le processeur ARM peut communiquer avec un microcontrôleur en SPI pour configurer le CAN.

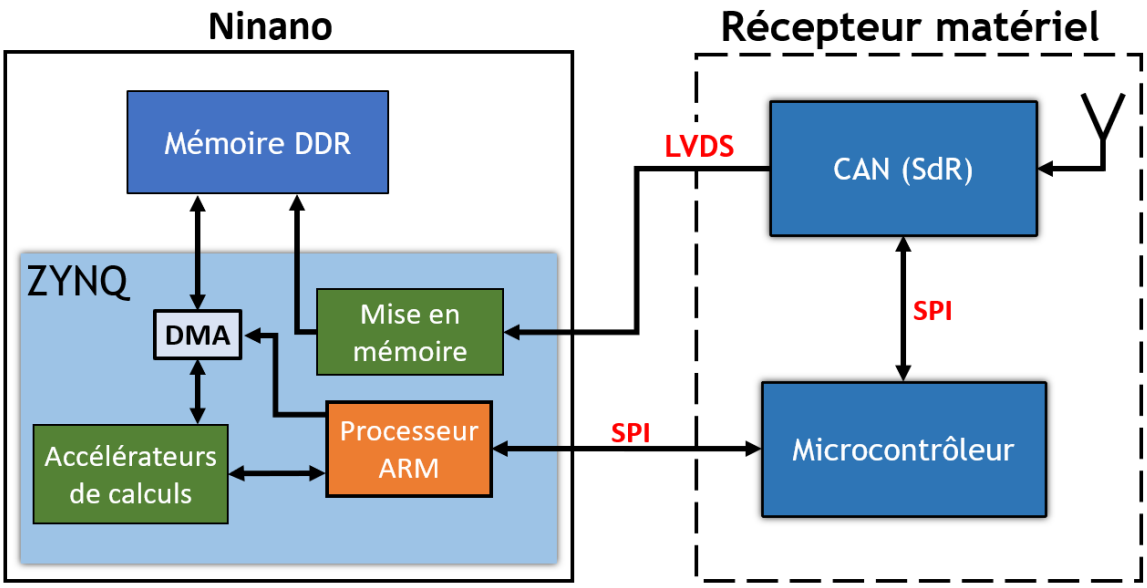


Figure 7 – Ressources matérielles de la charge ELIOT

V. Etat de départ du projet

Maintenant que le contexte et la communication ont étés présentés, il est nécessaire de décrire l'état de départ de notre projet.

Tout d'abord, la communication LoRa améliorée a été développée par l'équipe du chercheur Guillaume FERRE qui a validé ses travaux en Matlab.

Une simulation logicielle complète en C++ a été réalisée par l'équipe du chercher Bertrand LE GAL. Cette même équipe a ensuite implémenté la communication en matériel sur Raspberry en utilisant des radio-logicielles.

Finalement, une implémentation matérielle en conception conjointe CPU/FPGA sur Zynq 7020 a été réalisé par Valentin ROMERO avec l'encadrement de Bertrand LE GAL.

Figure 8– Plateformes d'implémentation ou de développement de la communication LoRa améliorée

Comme nous l'avons expliqué dans la partie « Contexte », le récepteur LoRa est voué à être fortement amélioré dans le futur. Ces améliorations ont pour but de le rendre robuste à l'effet Doppler et de le rendre capable de démoduler plusieurs communications reçus simultanément.

Cependant, l'ajout de nouvelles fonctionnalités à notre récepteur impactera forcément les performances temporelles. Notre récepteur ne sera donc plus temps réel.

Pour répondre à cette problématique, nous allons modifier les accélérateurs utilisés par le récepteur sur Zynq. Notre but est d'augmenter sa marge de manœuvre temporelle, et donc nous facilitons sa capacité à rester temps réel sur le long terme.

<u>Notre projet se décompose ainsi en trois parties distinctes :</u>

- L'implémentation d'un algorithme permettant de détecter la présence de plusieurs trames dans un buffer de réception

- L'amélioration d'un accélérateur FPGA de calculs dichotomiques pour obtenir de meilleures performances temporelles lors de la réception

- L'implémentation d'un accélérateur FPGA calculant des FFTs afin d'améliorer à nouveau les performances temporelles

VI. Implémentation d'un algorithme de détection de multiples préambules

La communication LoRa actuellement implémentée en C++ n'est pas robuste à la réception simultanée de plusieurs communications. Dans le cas où notre récepteur reçoit N communications en même temps, une seule d'entre elles sera décodée.

Pour détecter ce problème, l'équipe de Guillaume FERRE a conçu un algorithme Matlab permettant la détection de plusieurs préambules LoRa reçus simultanément. Notre but est donc d'implémenter cet algorithme en C++.

1. Introduction

Le récepteur LoRa développé en C++ est capable de démoduler des trames de SF 7 à 12. Cependant, dans le cadre de ce projet, notre algorithme de détection sera limité aux SF 7 à 9.

Notre algorithme doit réaliser trois recherches au maximum de la présence d'interférences pour chaque SF afin de limiter les dégradations temporelles de notre récepteur.

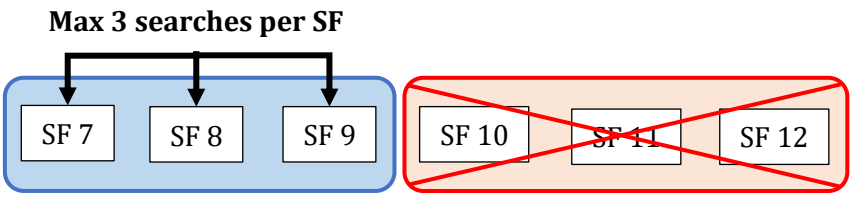


Figure 9 – Liste des Spread Factor pris en compte par l'algorithme développé

Cette implémentation C++ est purement logicielle, elle ne sera donc pas à implémenter sur le Zynq-7020 lors de ce projet.

L'algorithme de détection de multiples préambules est constitué de trois parties principales :

Figure 10 – Algorithme de détection de plusieurs préambules

La partie « buffer shapping » s'occupe de mettre en forme le buffer de réception afin de détecter la présence de trames LoRa. Cette mise en forme consiste à mettre en évidence les aspects énergétique et fréquentielle de notre signal.

La partie « preamble search » s'occupe de détecter les pics d'énergie et de vérifier qu'ils sont bien associés à une communication LoRa.

Pour finir, lorsqu'une trame est détectée, la partie « deleting and storing » va incrémenter le nombre de trame trouvée. La trame est alors supprimée pour trouver de nouvelles trames dans le buffer de réception.

Afin de faciliter la compréhension, les différentes parties sont décrites à nouveau ci-dessous :

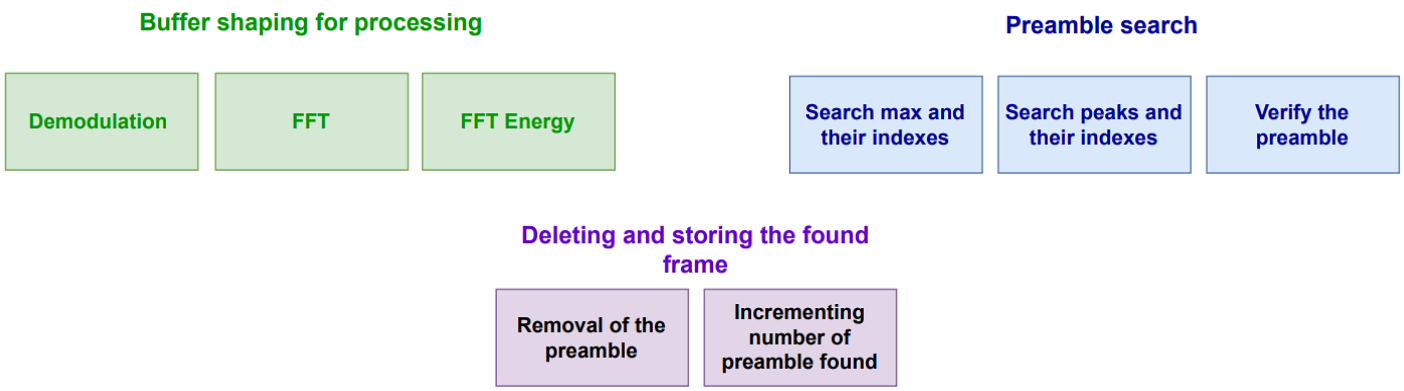


Figure 11 – Algorithme de détection de plusieurs préambules

2. Stratégie d'implémentation

L'implémentation de cet algorithme nécessite une stratégie de vérification efficace et fiable. Nous avons ainsi choisi la stratégie du « Golden Model » décrite sur le schéma ci-dessous :

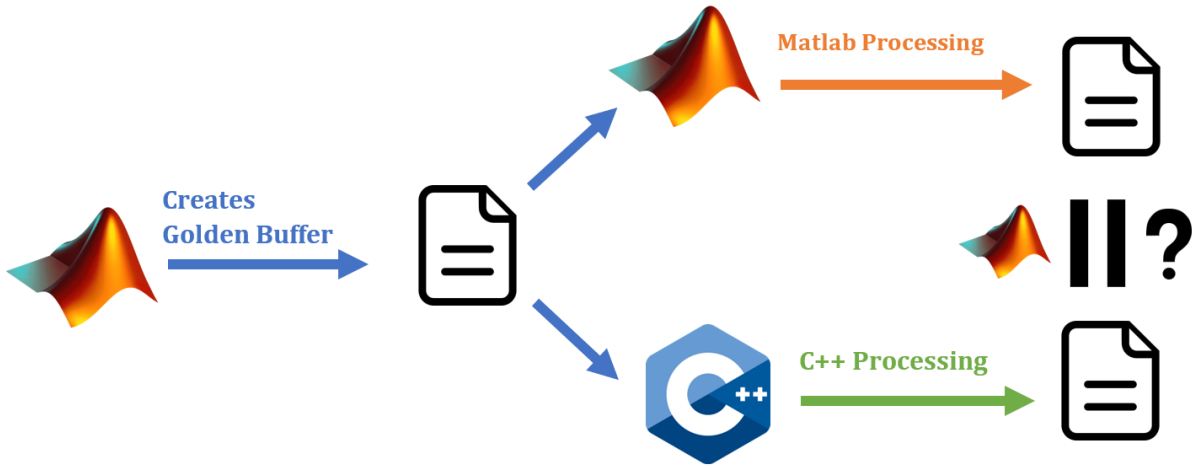


Figure 12 – Stratégie d'implémentation de l'algorithme

Comme nous savons que le programmable Matlab est fiable, c'est lui qui doit s'occuper de générer le buffer de réception ainsi que les paramètres dont a besoin l'algorithme.

Ce Golden Model permet de comparer les résultats de Matlab aux résultats obtenus en C++.

Dans Matlab, nous avons réalisé des modifications qui permettent aux codes d'avoir 3 modes de fonctionnement.

1- Génération d'une trame aléatoire pour C++
2- Génération d'une trame « Golden Model » pour Matlab et C++
3- Lecture d'un « Golden Model » déjà généré par Matlab et comparaison avec les fichiers générés en C++

Dans notre cas, nous utiliserons une fois le 2e mode (génération du golden model), puis nous resterons focalisés sur le 3e mode (lecture du golden model) pour développer notre code C++ tout en comparant les résultats avec ceux de Matlab.

Lors du développement, certaines erreurs peuvent être très difficiles à détecter. Pour gagner en efficacité et en fiabilité, nous allons faire en sorte que chaque étape soit vérifiée et visualisée à l'aide de Matlab.

3. Buffer d'entrée

Comme nous l'avons expliqué dans notre stratégie d'implémentation, nous voulons pouvoir vérifier et visualiser chaque étape de notre algorithme C++.

Lors de l'exécution de notre code, on récupère les paramètres et le buffer de données fournie par Matlab :

```
#(II) Lora Multiple Preamble Detection
#(II) ---------------------------------------------
#(II) Parameters obtained from file : matlab_gold_param.txt
#(II) ---------------------------------------------
#(II) +                SF              = [9,8,7]
#(II) +                M               = [512,256,128]
#(II) +                Nb_preambule    = [2,4,8]
#(II) +                th              = 2.21082
#(II) +                Pfa             = 0.001
#(II) +                Fse             = 1
#(II) +                alpha           = 4
#(II) +                Nb_max          = 3
#(II) +                buffer_size     = 156250
```

Figure 13 – Paramètres reçus par l'algorithme C++

Dans un premier temps, nous allons donc vérifier que l'implémentation Matlab et C++ trouvent bien le même buffer de donnée en entrée.

```
SF7
-----------------------------------------------------------------
CPP buffer   OK:     R_dif_max= 0.00000073   I_dif_max= 0.00000073
```

```
SF8
-----------------
CPP buffer   OK:
```

```
SF9
-----------------
CPP buffer   OK:
```

Figure 14 – Vérification du buffer de réception C++ par Matlab

On obtient bien le même résultat. Matlab nous permet également de visualiser le buffer d'entrée pour s'assurer qu'il n'y ait pas d'erreur.

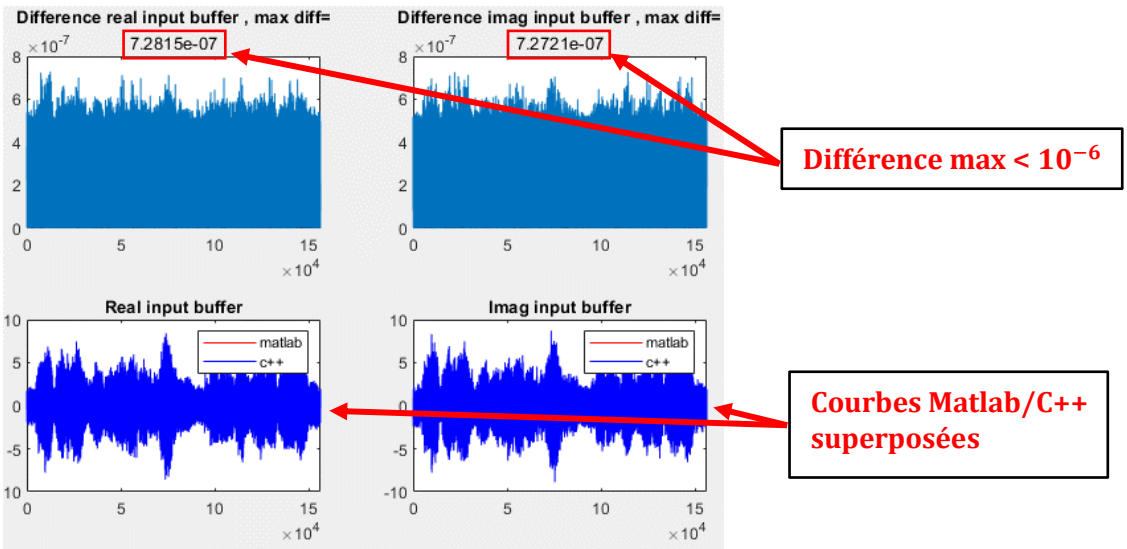


Figure 15 – Visualisation du buffer de réception C++ par Matlab

Sur cette figure on peut visualiser différentes choses :

- En haut à gauche : : La différence Matlab/C++ sur les réels
- En haut à droite : La différence Matlab/C++ sur les imaginaires
- En bas à gauche : Les réels Matlab (rouge) et C++ (bleue)
- En bas à droite : Les imaginaires Matlab (rouge) et C++ (bleue)

Avec la visualisation, nous pouvons confirmer que l'algorithme C++ a correctement lu le buffer de réception.

4. Downchirp

Afin de démoduler notre buffer, nous avons besoin d'un « Downchirp » à multiplier avec notre signal d'entrée.

Avant de s'assurer que notre démodulation est correctement réalisée, on veut s'assurer que notre « Downchirp » obtenu en C++ est identique à celui utilisé sur Matlab. On réalise nos vérifications :

```
          SF7
-------------------------------------------------------------------
CPP buffer    OK:      R_dif_max= 0.00000073    I_dif_max= 0.00000073
Down Chirp    OK:      R_dif_max= 0.00000051    I_dif_max= 0.00000051
```

```
          SF8
-----------------
CPP buffer    OK:
Down Chirp    OK:
```

```
          SF9
-----------------
CPP buffer    OK:
Down Chirp    OK:
```

Figure 16 – Vérification du Downchirp C++ par Matlab

Comme précédemment, on vérifie graphiquement notre résultat :

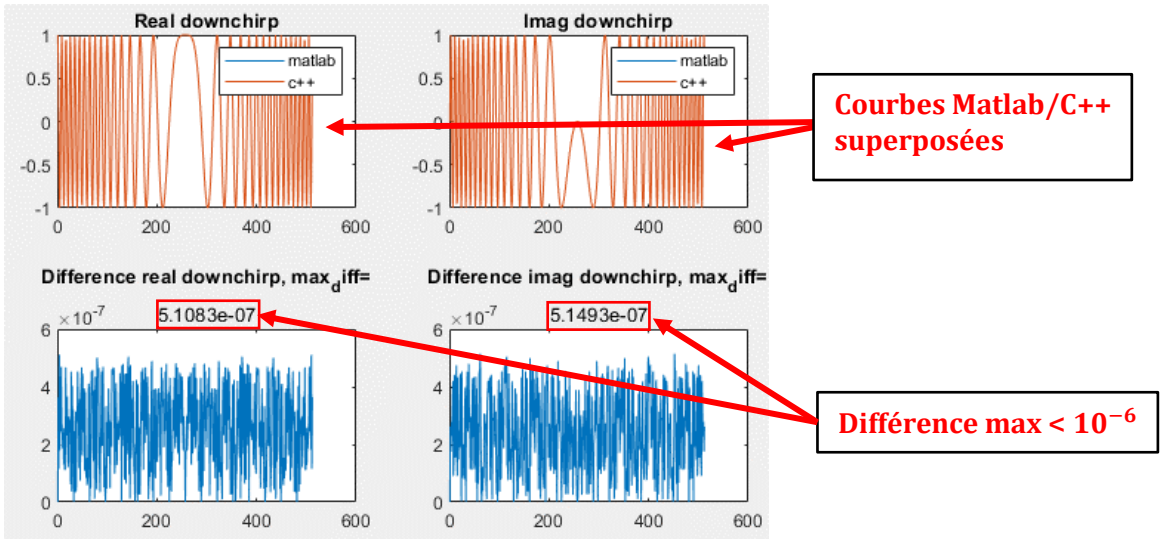


Figure 17 – Visualisation du Downchirp C++ par Matlab

Les courbes Matlab et C++ sont superposées, il n'y a aucune différence, on en conclue que notre « Downchirp » est correctement modélisé en C++.

5. Démodulation

La démodulation est réalisée en multipliant notre Downchirp avec notre buffer d'entrée.

Si le buffer contient N chirp, il faudra multiplier notre signal par N downchirp pour démoduler notre signal.

Matlab vérifie alors que notre démodulation est correcte :

```
        SF7
-------------------------------------------------------------------
CPP buffer    OK:     R_dif_max= 0.00000073    I_dif_max= 0.00000073
Down Chirp    OK:     R_dif_max= 0.00000051    I_dif_max= 0.00000051
Dechirp       OK:     R_dif_max= 0.00000118    I_dif_max= 0.00000104
```

```
        SF8
-------------------
CPP buffer    OK:
Down Chirp    OK:
Dechirp       OK:
```

```
        SF9
-------------------
CPP buffer    OK:
Down Chirp    OK:
Dechirp       OK:
```

Figure 18 – Vérification de la démodulation C++ par Matlab

On peut encore une fois visualiser la démodulation pour chaque SF.

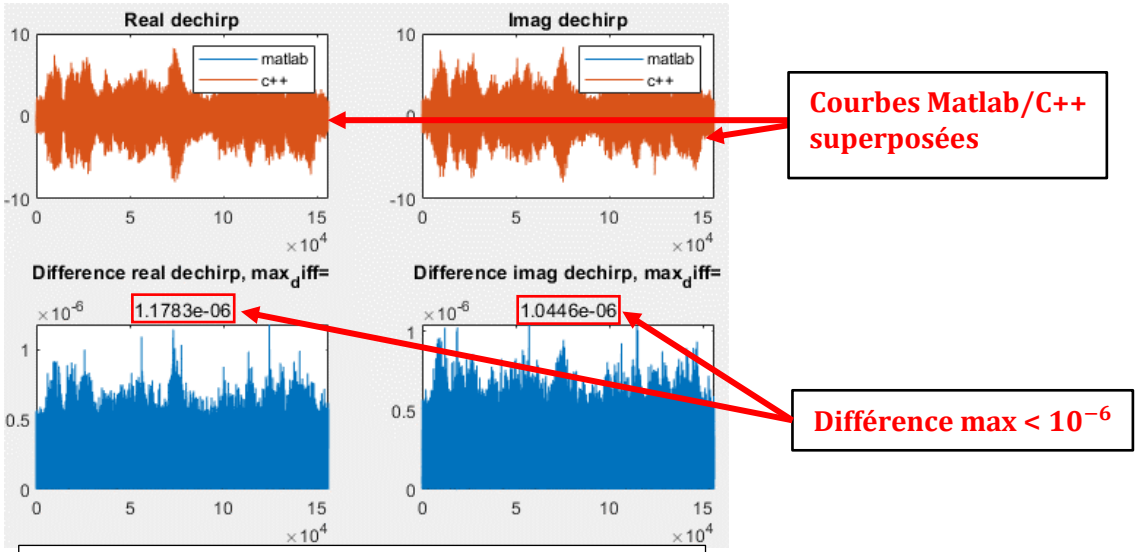


Figure – Visualisation du Downchirp C++ par Matlab

Il n'y a aucune différence entre Matlab et C++, la démodulation de notre signal est correctement réalisée.

6. Fast Fourier Transformation

Afin de détecter des préambules, il est nécessaire de connaître la densité spectrale d'énergie de chaque chirp de notre buffer de réception. Cette dernière permettra de savoir si nous avons reçu un préambule.

Dans un premier temps, nous allons calculer la FFT puis la FFT^2 de notre buffer d'entrée.

Prenons comme exemple le SF 7. Les caractéristiques sont les suivantes :

- Taille de buffer : $N = 156\,250$
- Taille d'un chirp : $M = 128 * 4$
- Nombre de chirp : $N_c = {}^{N}/_{M} = 305$

Notre algorithme C++ doit donc réaliser $N_c = 305$ FFTs de taille $M = 512$ pour le SF 7.

Ci-dessous, on peut voir la manière dont notre algorithme C++ réalise le calcul de la FFT pour le SF 7 :

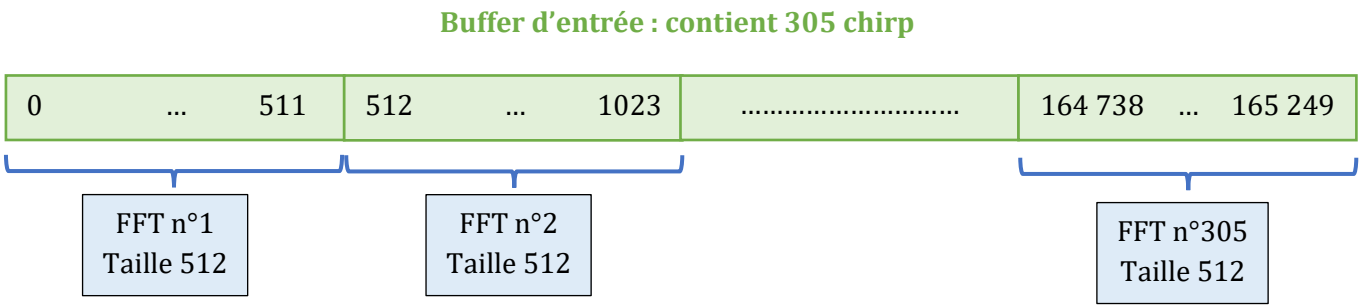


Figure 19 – Fonctionnement de la FFT implémentée en C++ pour le SF 7

Une fois notre FFT obtenue, on calcule son module au carré. On vérifie nos résultats avec ceux de Matlab :

```
                SF7
-------------------------------------------------------------------
CPP buffer    OK:     R_dif_max= 0.00000073    I_dif_max= 0.00000073
Down Chirp    OK:     R_dif_max= 0.00000051    I_dif_max= 0.00000051
Dechirp       OK:     R_dif_max= 0.00000118    I_dif_max= 0.00000104
FFT           OK:     R_dif_max= 0.00000152    I_dif_max= 0.00000152
FFT_square    OK:     dif_max=   0.00004294
```

```
          SF8
-----------------
CPP buffer    OK:
Down Chirp    OK:
Dechirp       OK:
FFT           OK:
FFT_square    OK:
```

```
          SF9
-----------------
CPP buffer    OK:
Down Chirp    OK:
Dechirp       OK:
FFT           OK:
FFT_square    OK:
```

Figure 20 – Vérification de la FFT et de son module au carré par Matlab

On peut visualiser nos résultats et se rendre compte qu'ils sont identiques :

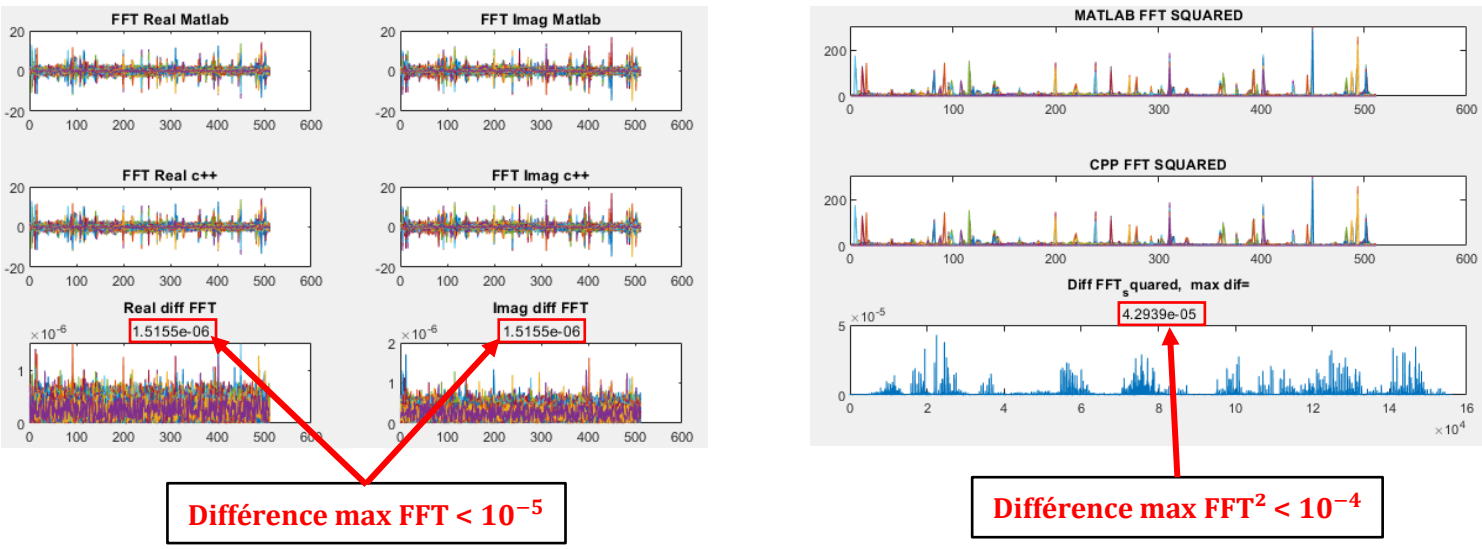


Figure 21 – Visualisation de la FFT et FFT² par Matlab

7. FFT Energy

On s'intéresse maintenant à calculer l'énergie de nos modules de FFT au carré pour détecter les communications LoRa reçues.

Pour comprendre le fonctionnement du calcul de l'énergie, on peut utiliser une représentation matricielle des données en SF 8.

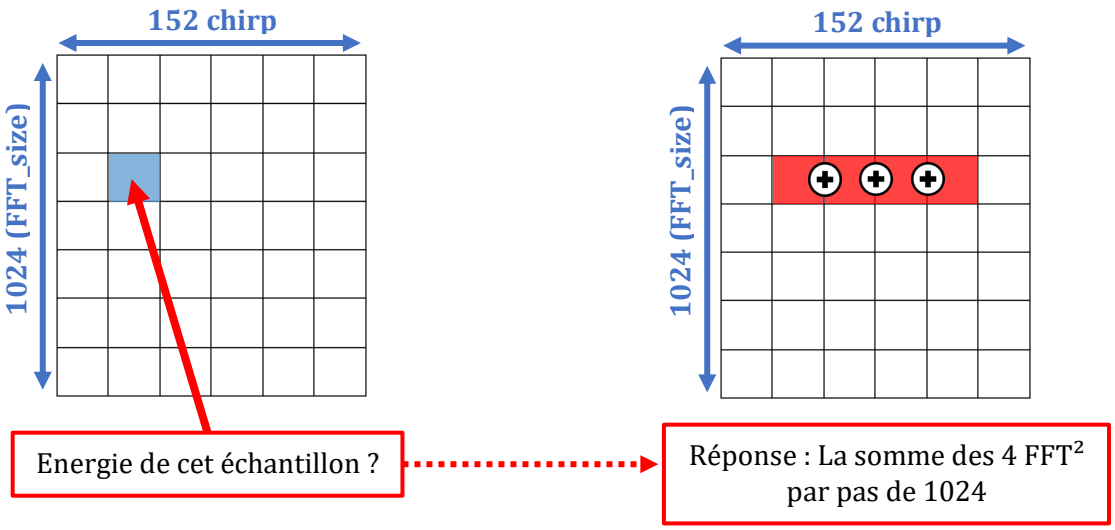


Figure 22– Calcul de l'énergie implémenté en Matlab pour le SF 8

Dans le cas des SF 7, 8 et 9, on somme respectivement 8, 4 ou 2 modules au carré de FFT.

En C++, on travaille avec un buffer 1 dimension. Ci-dessous une représentation du fonctionnement C++ en SF 8 :

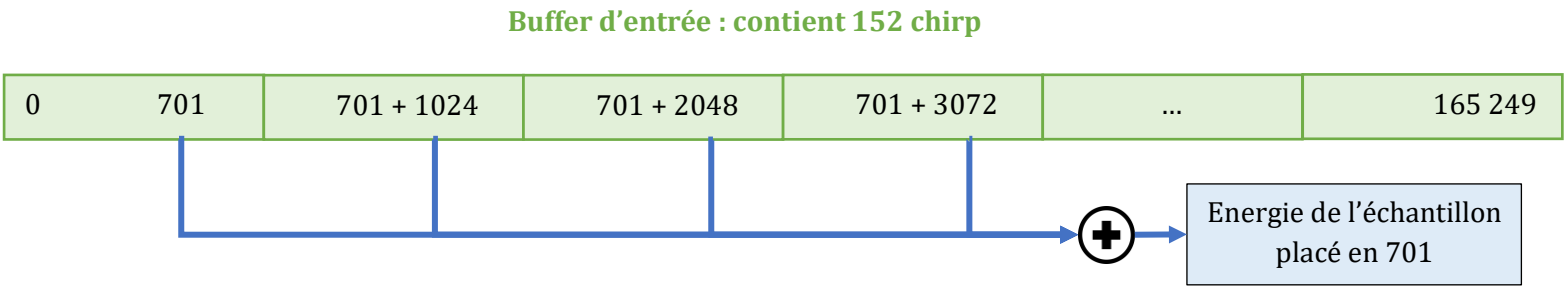


Figure 23 – Calcul de l'énergie implémenté en C++pour le SF 8

On vérifie notre implémentation C++ avec Matlab :

```
                SF7
-------------------------------------------------------------------
CPP buffer    OK:     R_dif_max= 0.00000073   I_dif_max= 0.00000073
Down Chirp    OK:     R_dif_max= 0.00000051   I_dif_max= 0.00000051
Dechirp       OK:     R_dif_max= 0.00000118   I_dif_max= 0.00000104
FFT           OK:     R_dif_max= 0.00000152   I_dif_max= 0.00000152
FFT_square    OK:     dif_max=   0.00004294
FFT_energy    OK:     dif_max=   0.00020302
```

```
                SF8
-----------------
CPP buffer    OK:
Down Chirp    OK:
Dechirp       OK:
FFT           OK:
FFT_square    OK:
FFT_energy    OK:
```

```
                SF9
-----------------
CPP buffer    OK:
Down Chirp    OK:
Dechirp       OK:
FFT           OK:
FFT_square    OK:
FFT_energy    OK:
```

Figure 24 – Vérification de l'énergie des FFTs par Matlab

Il n'y a pas de différence de résultat avec notre implémentation, on peut visualiser avec Matlab

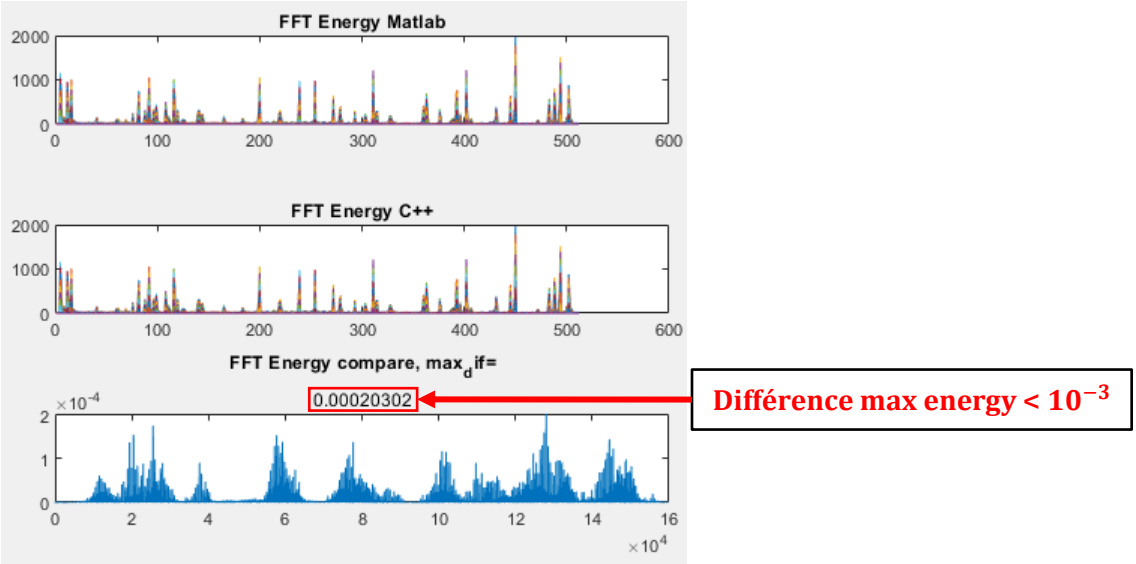


Figure 25 – Visualisation de l'énergie des FFTs par Matlab

8. Valeurs maximales par chirp

Pour trouver les pics d'énergie correspondant à une communication LoRa, on doit tout d'abord trouver les l'énergie maximale de chaque chirp. On stocke également la position de ces maximums.

Le fonctionnement étant très basique, on s'intéresse directement aux résultats :

```
          SF7
-------------------------------------------------------------------
CPP buffer    OK:     R_dif_max= 0.00000073   I_dif_max= 0.00000073
Down Chirp    OK:     R_dif_max= 0.00000051   I_dif_max= 0.00000051
Dechirp       OK:     R_dif_max= 0.00000118   I_dif_max= 0.00000104
FFT           OK:     R_dif_max= 0.00000152   I_dif_max= 0.00000152
FFT_square    OK:     dif_max=   0.00004294
FFT_energy    OK:     dif_max=   0.00020302
Max val/id    OK:     MAX_dif=   0.00020302   IDX_dif=   0.00000000
```

```
          SF8
------------------
CPP buffer    OK:
Down Chirp    OK:
Dechirp       OK:
FFT           OK:
FFT_square    OK:
FFT_energy    OK:
Max val/id    OK:
```

```
          SF9
------------------
CPP buffer    OK:
Down Chirp    OK:
Dechirp       OK:
FFT           OK:
FFT_square    OK:
FFT_energy    OK:
Max val/id    OK:
```

Figure 26 – Vérification des max et de leurs positions par Matlab

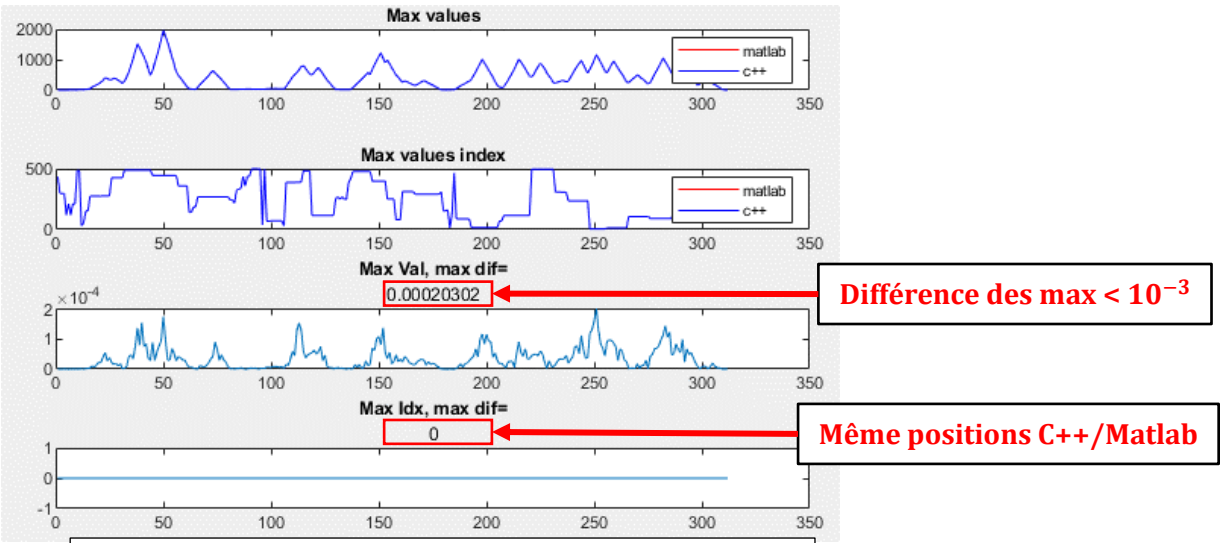


Figure 27 – Visualisation de l'énergie des FFTs par Matlab

On remarque avec nos résultats que les valeurs maximales trouvées en C++ sont quasiment identiques à celles trouvées en Matlab.

Les positions de ces maximums sont parfaitement trouvées par l'algorithme C++, cette partie est donc fonctionnelle.

9. Pics parmi les maximums

Pour trouver un pic parmi les maximums on applique l'algorithme suivant :

- 1) On trouve le maximum des maximums
- 2) On vérifie que les valeurs à droite et à gauche soient plus faibles
- 3) Autour du pic, les maximums ne deviendront jamais des pics (2, 4 ou 8 maximums autour)
- 4) On retourne à l'étape 1

Pour illustrer cet algorithme, voici un exemple graphique :

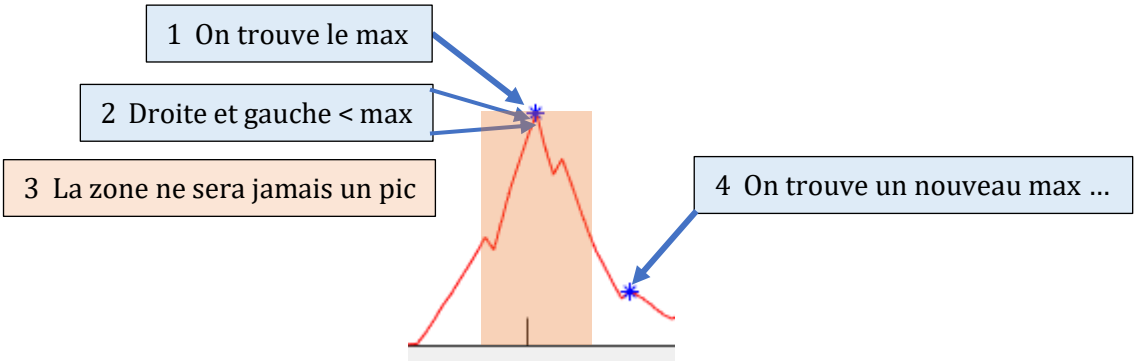


Figure 28 – Algorithme pour trouver les pics

Cet algorithme est implémenté en C++, on peut voir les résultats ci-dessous :

```
                SF7
-------------------------------------------------------------------
CPP buffer    OK:     R_dif_max= 0.00000073   I_dif_max= 0.00000073
Down Chirp    OK:     R_dif_max= 0.00000051   I_dif_max= 0.00000051
Dechirp       OK:     R_dif_max= 0.00000118   I_dif_max= 0.00000104
FFT           OK:     R_dif_max= 0.00000152   I_dif_max= 0.00000152
FFT_square    OK:     dif_max=   0.00004294
FFT_energy    OK:     dif_max=   0.00020302
Max val/id    OK:     MAX_dif=   0.00020302   IDX_dif=   0.00000000
Findpeaks     OK:     dif_max=   0.00000000
```

```
                SF8
------------------
CPP buffer    OK:
Down Chirp    OK:
Dechirp       OK:
FFT           OK:
FFT_square    OK:
FFT_energy    OK:
Max val/id    OK:
Findpeaks     OK:
```

```
                SF9
------------------
CPP buffer    OK:
Down Chirp    OK:
Dechirp       OK:
FFT           OK:
FFT_square    OK:
FFT_energy    OK:
Max val/id    OK:
Findpeaks     OK:
```

Figure 29 – Vérification pics par Matlab

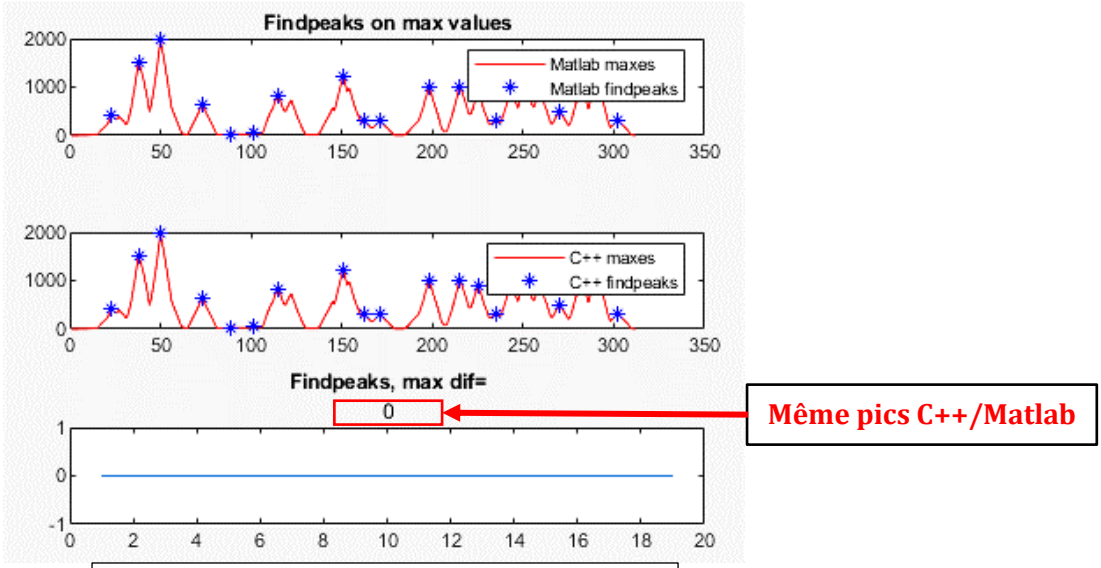


Figure 30 – Visualisation des pics par Matlab

On trouve les mêmes pics entre Matlab et C++.

10. Traitement final

Avec les traitements précédents, on peut détecter le nombre de trame reçu pour chaque SF.

Matlab et C++ trouvent les mêmes résultats :

```
DETECT        OK:      SF_dif=     0.00000000    ID_dif=     0.00000000
```

Figure 31 – Vérification des trames trouvées en C++ par Matlab

On peut voir la manière dont les trames sont détectées avec la visualisation Matlab :

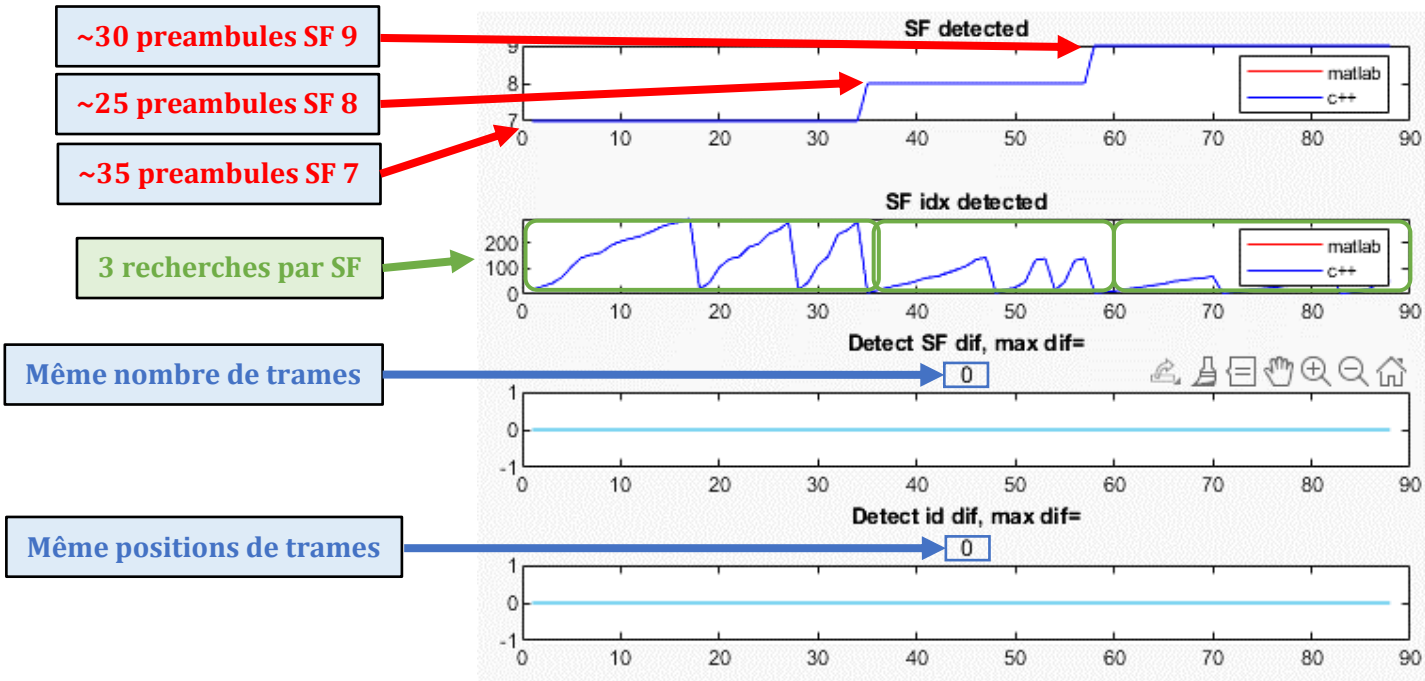


Figure 32 – Visualisation résultats finaux par Matlab

11. Conclusion

Nous avons correctement réussi à implémenter en C++ notre algorithme Matlab de détection de communications simultanées.

Le bon fonctionnement de cet algorithme est vérifié par Matlab, et il est également visualisable pour chaque étape et pour chaque SF.

La prochaine étape est maintenant d'implémenter cet algorithme sur la plateforme Zynq du nanosatellite NanoNAASC pour démoduler plusieurs communications reçues simultanéments.

VII. Prise en main d'outils pour l'implémentation d'accélérateurs FPGA

1. Nécessité d'optimisation

Notre récepteur LoRa doit fournir des performances suffisantes pour permettre une réception et une démodulation correcte des trames reçues. Si l'algorithme est trop lent pour démoduler une trame, nous risquons de manquer la réception d'autres trames, notre récepteur ne sera plus Temps Réel. De plus, notre récepteur est amené à évoluer dans le futur, notamment par l'ajout de nouvelles fonctionnalités.

La métrique utilisée pour mesurer les performances temporelles du récepteur est le temps d'exécution maximum. À l'aide d'outils de mesures temporelles développés précédemment, il est possible de mesurer précisément le temps d'exécution de notre récepteur. Mais il faut de plus pouvoir comparer ces mesures aux valeurs critiques à ne pas dépasser pour rester Temps Réel.

Ces valeurs critiques sont obtenues par des considérations théoriques. L'échantillonnage est effectué à T = 1 Mega symbole/s, tandis que le buffer de données contient
$N(SF) = 2^{SF} * 23 * 8 * 2$ couples I/Q. Ici, 2^{SF} représente le nombre de bits par symboles, 23 le nombre de chirps attendus dans une trame reçue, 8 le facteur de suréchantillonnage et 2 représente le fait que les données sont représentées par des couples I/Q. Il est donc possible de calculer, pour chaque SF, le temps d'exécution critique t = N(SF) / T à ne pas dépasser pour rester Temps Réel. Le tableau x recense ces valeurs critiques.

SF	**7**	**8**	**9**	**10**	**11**	**12**
t (ms)	47.1	94.2	188.4	376.8	753.7	1507.3

Figure 33 - Tableau des temps d'exécutions limites

2. Premières optimisations déjà réalisées

Sur la figure 34 sont reportés en rouge ces temps d'exécutions limites en fonction de SF. En orange sont représentés les temps d'exécution de notre récepteur, sans aucune optimisation. Ces temps ont été relevés en utilisant les outils de mesures déjà développés. Sans optimisation, notre récepteur n'est pas Temps Réel, quelque soit SF. Le temps d'exécution est toujours au-dessus du temps critique.Il apparaît donc clairement le besoin d'optimiser le récepteur pour améliorer ses performances et ce, pour chaque SF.

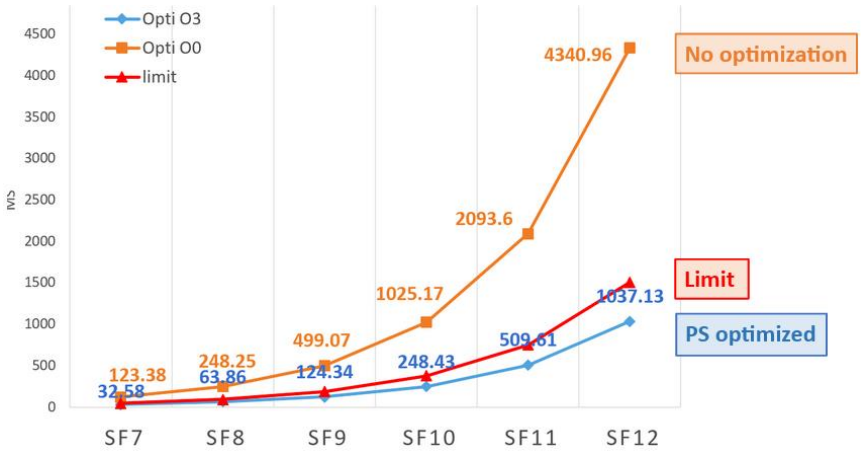


Figure 34 – Temps d'exécution maximum limites théoriques (en rouge), sans optimisation (en orange) et avec les optimisations initiales (en bleu) en fonction de SF

Les premières optimisations réalisées ont été l'utilisation des optimisations proposées par le compilateur utilisé pour compiler les fichiers C++ du projet. Le projet était initialement compilé sans optimisation i.e. avec un flag d'optimisation -O0. Le choix a été fait d'utiliser le flag d'optimisation -O3. D'autres optimisations algorithmiques ont également été apportées. Sur la figure 34, les performances après optimisations initiales sont représentées en bleu. Notre récepteur passe désormais en dessous des valeurs critiques. Cependant, la marge est très faible.

Nous souhaitons donc continuer les optimisations. Il a pu être déterminé qu'une grande partie du temps d'exécution est due aux calculs dichotomiques et de FFT (Fast Fourier Transform). Il a alors été décidé d'implémenter des accélérateurs matériels afin de décharger le CPU de ces calculs coûteux en temps.

3. Codesign, HLS

L'objectif est de déporter les calculs dichotomiques et de FFT sur des accélérateurs matériels implémentés sur FPGA. Ces accélérateurs se doivent d'effectuer ces calculs plus rapidement que le processeur.

Cette utilisation conjointe d'un processeur (ou PS pour Processing System) et d'un ou plusieurs accélérateurs (ou PL pour Programmable Logic) est appelée Codesign. Le codesign respecte un flot de développement que nous allons détailler ici. Les outils Xilinx ont été utilisés durant ce projet (Vitis HLS, Vivado et Vitis

a. Vitis HLS

La première étape est de créer notre accélérateur matériel (ou IP pour Intellectual Property). Nous utilisons le logiciel Vitis HLS (High Level Synthesis), un outil de synthèse de haut niveau. La HLS permet de générer une architecture matérielle de niveau RTL à partir d'une description comportementale algorithmique de plus haut niveau (C, C++, Python etc).

Il faut donc concevoir son IP à partir d'un code de haut niveau, en décrivant par un algorithme le comportement attendu de notre IP. Il ne s'agit donc plus d'écrire un code compilable et exécutable par un processeur, mais bien de décrire une architecture matérielle de manière algorithmique. L'outil se charge alors de traduire notre code en architecture de niveau RTL (Register Transfer Logic) en générant des fichiers HDL (VHDL ou Verilog).

Un test bench doit ensuite être conçu pour valider le fonctionnement de notre IP. Le test bench crée des données aléatoires d'entrée. Il les fournit alors à deux entités : notre IP et un golden model conçu dans le test bench. Le test bench compare alors les données de sortie fournies par ces deux entités. Si ces deux jeux de données sont identiques, le fonctionnement de l'IP est validé en HLS.

La validation par test bench est réalisée en deux étapes :

- Premier test : validation fonctionnelle de l'algorithme. Le test bench est d'abord effectué sur notre description algorithmique elle-même pour être sûr que l'algorithme décrit bien le comportement souhaité.
- Second test : validation fonctionnelle de l'IP. Le test bench est ensuite effectué sur l'IP généré par Vitis HLS afin de valider la solution proposée par Vitis HLS.

La figure 35 résume la conception de l'IP et sa validation à l'aide de Vitis HLS.

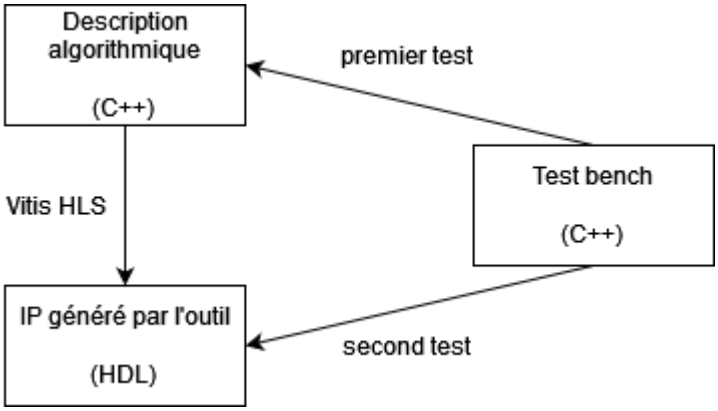


Figure 35 - Conception de l'IP dans Vitis HLS.

b. Vivado

L'outil Vitis HLS ayant généré notre IP sous la forme de fichiers HDL, nous continuons alors le flot de développement HLS par l'outil Xilinx Vivado. Utiliser ce logiciel nous permet de développer la plateforme, représentée sur la figure 36, sur laquelle notre récepteur va être implémenté.

Le but est ici de décrire comment la PS et PL doivent être agencés dans la cible matérielle Zynq. Sur la droite se trouve la PS, qui exécutera le code C++ du récepteur. Au centre se trouve l'IP. Les interconnexions entre PS et PL utilisent le bus AXI du Zynq. Les données transitent entre PS et PL à l'aide d'une DMA (Direct Memory Access). Observons également que tous les blocs sont synchronisés sur l'horloge de la PS, ainsi que sur son signal de reset. Enfin, tous les signaux d'interruptions des différents blocs sont concaténés pour être envoyés sur une entrée d'interruption de la PS.

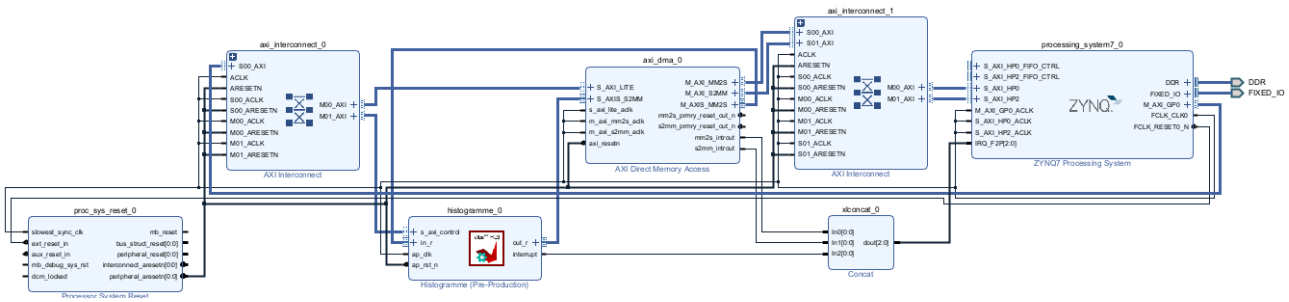


Figure 36 - Plateforme matérielle qui sera implémentée sur la cible matérielle Zynq.

c. Vitis

Le flot de conception se termine par l'utilisation de Vitis. Avec ce dernier logiciel, nous pouvons développer notre code logiciel C++ en ciblant la plateforme matérielle précédente. Vitis nous permet de compiler le code logiciel, configurer la matrice FPGA du Zynq selon notre plateforme matérielle et programmer la PS avec notre code logiciel compilé. Il est alors possible de visualiser le retour de la carte par connexion série entre la carte de développement Pynq et le PC de développement.

La figure 37 résume le flot de conception mis en oeuvre à l'aide des outils Xilinx.

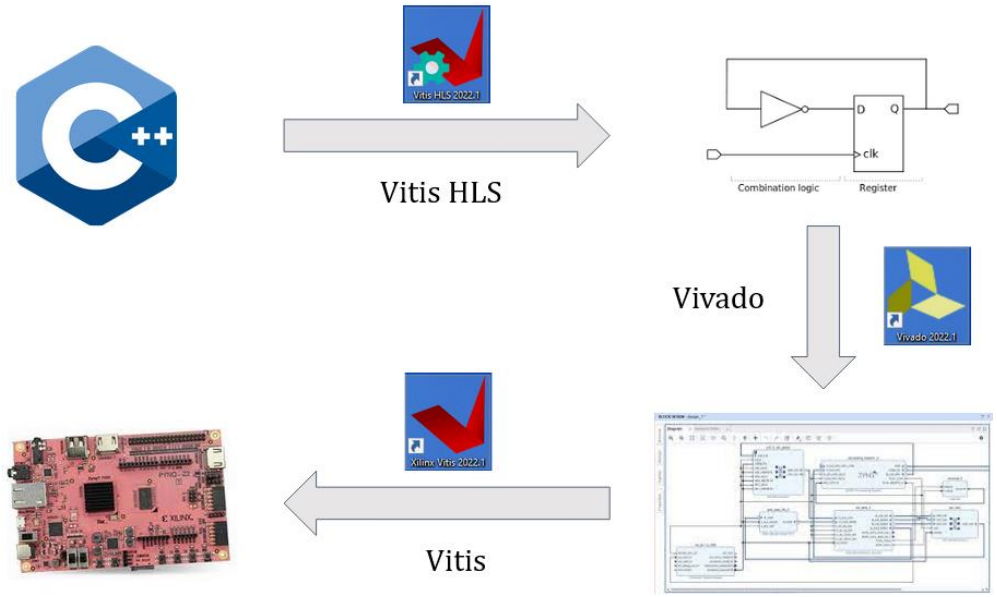


Figure 37 – Outils Xilinx pour le flot de développement HLS.

d. Exemples d'IP créées

Avant d'implémenter les IP nécessaires au projet, nous avons commencé par des exemples simples d'accélérateurs afin de saisir les outils utilisés. Le premier exemple est un calcul d'histogramme sur 1024 entiers entre 0 et 255. Ce premier exemple utilise volontairement un type de données simple (des entiers) afin de se focaliser sur la manière de coder un accélérateur. Nous avons pu comprendre comment décrire les entrées-sorties de notre accélérateur sur le bus AXI, en utilisant des Stream (FIFO). Le deuxième exemple est un calcul de module carré sur des nombres imaginaires, représentés par deux floats. Ce deuxième exemple nous a permis de nous familiariser avec l'utilisation de float dans les accélérateurs matériels, car notre projet utilise des données de type float.

Par ces deux exemples, nous avons compris comment les données sont échangées entre le processeur et les accélérateurs matériels. Le bus AXI du Zynq est utilisé. Le processeur envoie les données à traiter à l'accélérateur par le biais d'une DMA (Direct Memory Access). L'accélérateur réalise ses calculs. Pour retourner le résultat, deux options lui sont proposées. Il peut retourner le résultat comme à l'aller i.e. en utilisant la DMA ou bien en utilisant l'AXI Lite pour créer une interruption sur le processeur, qui est alors notifié de la fin des calculs de l'accélérateur.

VIII. Approx PL

1. Etat du projet initial

L'importance d'utiliser des accélérateurs pour réaliser les calculs de FFT et de la dichotomie vient du temps d'exécution de ces opérations sur CPU.

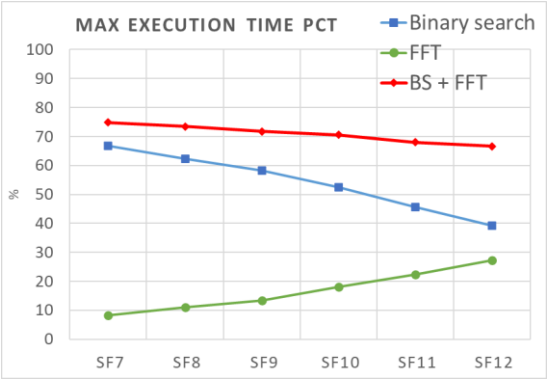


Figure 38 – Temps d'exécution nécessaires pour réaliser la FFT et la dichotomie (en pourcentage)

En effet, comme le montre la figure ci-dessus, les calculs de la FFT et la dichotomie représentent plus de deux tiers du temps de calculs sur CPU. Il est donc nécessaire d'accélérer ces calculs en les réalisant sur FPGA. Lorsque nous avons récupéré le projet S9, un accélérateur avait déjà été réalisé par Valentin Romero lors de son stage au NAASC. L'accélérateur a été nommé Approx PL et se présente ainsi :

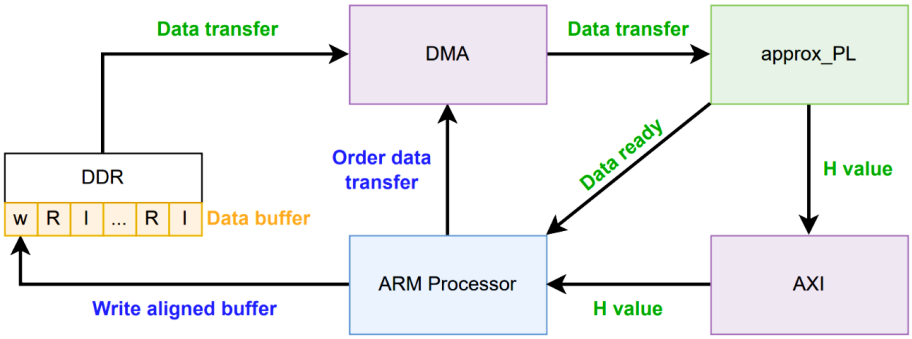


Figure 39 – Fonctionnement d'approx PL

Les couples R/I et w sont écrits par le CPU dans un buffer avant d'être transférés au FPGA par la DMA. Les données sont écrites dans une FIFO et le FPGA réalise les calculs. Une fois que le FPGA a achevé les calculs, il transmet son résultat par un bus AXI au CPU.

En utilisant cet accélérateur, il est possible de noter un fort gain de temps :

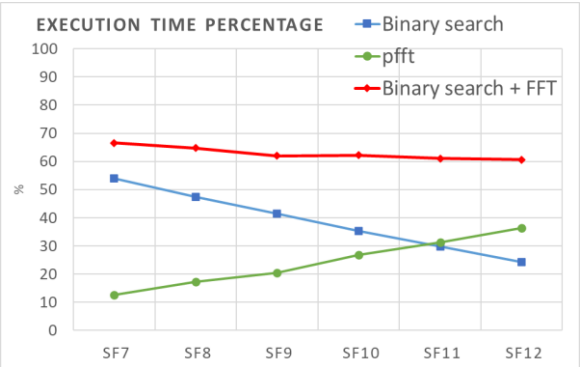


Figure 40 – Temps d'exécution nécessaires pour réaliser la FFT et la dichotomie (en pourcentage)

Désormais pour un SF12, la dichotomie représente 30% du temps d'exécution contre 45% en utilisant le CPU. Ce gain de temps n'est pas négligeable. Cependant, nous pensons qu'il est encore possible d'améliorer ce résultat en optimisant les transferts de données ou en modifiant l'architecture de l'accélérateur. C'est sur ces améliorations que porte cette parie.

2. Prise en main de l'algorithme

a. Problème

Dans un premier, nous avons voulu prendre en main l'algorithme en le faisant fonctionner sur nos cartes Pynq. Assez rapidement un problème est apparu : il semblait que le programme ne fonctionnait pas correctement. Selon le PC utilisé pour compiler et envoyer le code à la Pynq, le résultat semblait être différent. En effet, certaines trames n'étaient pas lues correctement par certaines cartes mais la vérification du début, elle, était toujours bonne.

```
(II) Accelerators status
(II) ---------------------------------------------
(II) FFT_PL      DISABLED
(II) APPROX_PL   ENABLED :   M=[1024]   fact=[2]

(II) Accelerators verify
(II) ---------------------------------------------
Approx_PL test ...
Approx_PL test OK
```

Figure 41 – La vérification est correcte

b. Première théorie

Dans un premier temps, nous avons pensé que l'origine du problème venait du *Negative Slack* obtenu après synthèse.

Modules & Loops	Issue Type	Violation Type	Distance	Slack
H_calcul				-1,12
H_calcul_Pipeline_VITIS_LOOP_37_1	II Violation			-1,12

Figure 42 – Negative Slack lors de la synthèse

Nous pensions qu'en ayant un *Negative Slack* négatif, nous avions un chemin critique qui dépassait la limite et provoquait donc une instabilité de l'IP. Cette instabilité pouvait se déclencher plus facilement selon l'ordinateur utilisé (OS et version de Vitis différente).

Cependant, l'étude de Vitis HLS nous a permis de constater que même si le *Negative Slack* était négatif lors de la synthèse, il devenait positif lors de l'implémentation. Le problème ne venait donc pas de là.

Timing Paths

Worst Negative Slack: 1.292ns(met)

Total Negative Slack: 0.000ns(met)

Figure 43 – Negative Slack lors de l'implémentation

Nous avons donc cherché une autre piste. Nous avons modifié le *Test Bench* sur Vitis HLS pour réaliser plusieurs tests, avec des paramètres *w* différents à chaque fois. Cette amélioration du *Test Bench* nous a été utile pour la suite du projet mais ne permis pas de trouver notre erreur.

c. Origine du problème

Nous avons finalement remarqué que selon le SF, les trames été lues correctement ou non. Pour un SF11 toutes les trames sont identifiées, ce n'est pas le cas pour un SF10. Nous avons donc étudié les codes sur Vitis et cela nous a permis de découvrir plusieurs problèmes.

Le premier problème était sur la fonction « verify_PL ». Cette fonction a pour but de s'assurer que les calculs en PS et PL donnent les mêmes résultats sur le même set de données générées aléatoirement.

En effet, elle permet de vérifier N échantillons PS/PL avec la même configuration, mais les tests étaient réalisés uniquement avec les paramètres stockés dans « param_PL ». Cela pose problème, car la vérification s'effectue toujours sur les mêmes paramètres, qui ne correspondent pas forcément à ceux sur lesquels nous voulons travailler. Même en changeant les paramètres, la vérification donnait toujours une validation alors que l'accélérateur ne pouvait pas fonctionner dans certaines configurations choisies.

Un second problème provenait de l'accélérateur qui ne fonctionnait pas dans toutes les configurations prévues. En effet, il aurait dû fonctionner pour un facteur de spectre SF10, mais les trames n'étaient pas trouvées. De plus, la fonction « verify_PL » nous assurait que l'accélérateur fonctionnait avec sa configuration PS associée.

d. Résolution du problème de verify_PL

Comme expliqué précédemment, verify_PL vérifie que l'accélérateur PL donne les mêmes résultats que le PS dans une configuration donnée. Il devrait donc comparer que les paramètres du programme lancé sont les mêmes pour obtenir une comparaison valide. Or, nous avons découvert que ce n'était pas le cas.

Dans un premier temps, nous avons vérifié ce que nous fournissait la fonction *H_calcul* sur la partie PS en SF10. On remarquait ainsi les intervalles suivants :

$$M = [1024]$$

$$fact = [1, 2]$$

$$R\_max = 4 \qquad R\_min = -4$$

$$I\_max = 4 \qquad I\_min = -4$$

$$R/I_{size} = [1024, 2048]$$

Or, notre accélérateur n'était prévu que pour la configuration $fact = 2$, ce qui posait problème. Il était donc nécessaire de modifier toute la chaîne d'utilisation de l'accélérateur pour qu'il puisse utiliser les cas $fact = [1, 2]$. En particulier, nous avons modifié la taille des tableaux envoyés dans l'accélérateur.

Les fichiers qui devaient être modifiés sont les suivants :

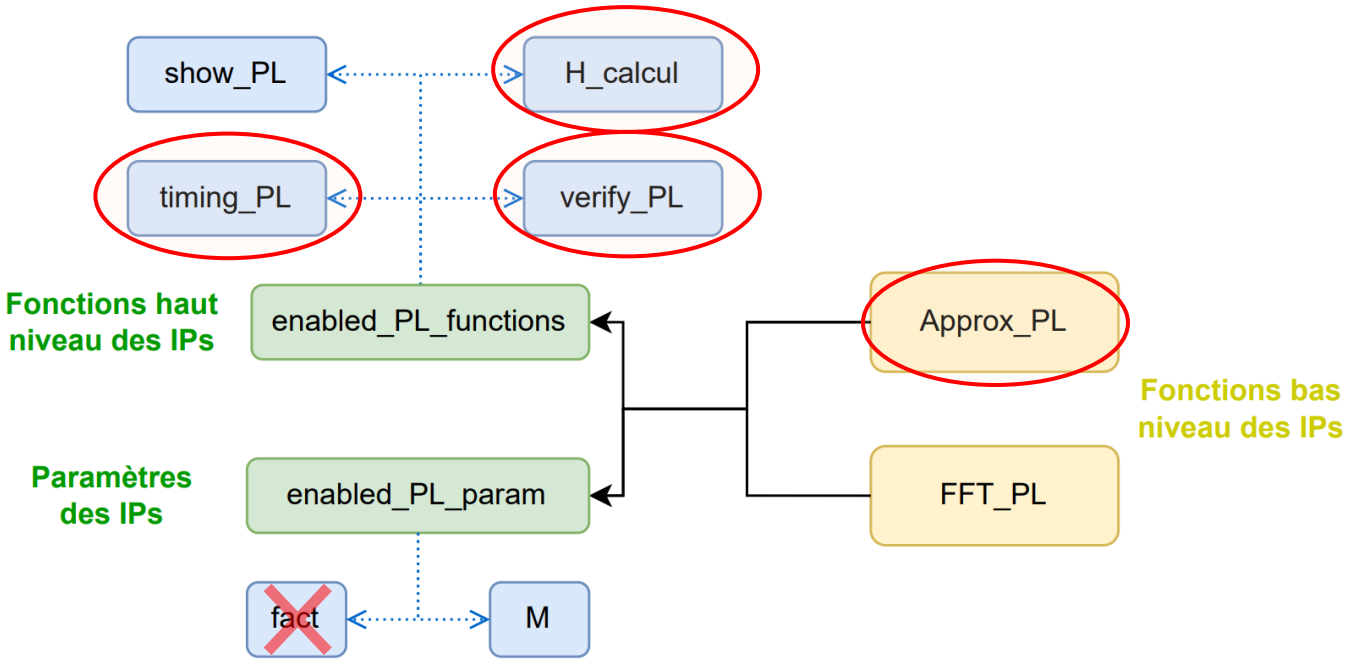


Figure 44 – Fichiers modifiés

Nous avons modifié non seulement la taille des tableaux, mais aussi les valeurs, qui sont maintenant générées aléatoirement, pour qu'elles correspondent à ce que reçoie la partie PS en pratique. Nous avons également modifié notre fonction « verify_PL » pour empêcher le programme de s'exécuter lorsque les SF sont différents.

L'accélérateur fonctionnait parfaitement pour un facteur $fact = 2$:

```
(II) Accelerators status
(II) ---------------------------------------------
(II) FFT_PL      DISABLED
(II) APPROX_PL   ENABLED :   M=[1024]

(II) Accelerators verify
(II) ---------------------------------------------
Approx_PL test ...
Approx_PL test OK

(II) Accelerators timing
(II) ---------------------------------------------
Approx_PL timing
PL processed  [10000] H_calcul for M=[1024] fact=[2]    |        [0.120425] ms/H_calcul        |        [8.303898] H_calcul/ms
PL2 processed [10000] H_calcul for M=[1024] fact=[2]    |        [0.225378] ms/H_calcul        |        [4.436983] H_calcul/ms
PS processed  [10000] H_calcul for M=[1024] fact=[2]    |        [0.452160] ms/H_calcul        |        [2.211606] H_calcul/ms
```

Figure 45 – Fonctionnement pour un fact=2

Mais le bus AXI_Lite se bloque pour $fact = 1$, ce qui reste cohérent puisqu'on ne recevait pas les 4097 données !

```
(II) Accelerators status
(II) ---------------------------------------------
(II) FFT_PL      DISABLED
(II) APPROX_PL   ENABLED :   M=[1024]

(II) Accelerators verify
(II) ---------------------------------------------
Approx_PL test ...
```

Figure 46 – Un test bloqué à cause d'une mauvaise taille de buffer

La fonction « verify » réalisait désormais les fonctions suivantes :

- Vérification que M_PL = M_PS
- Vérification que PS=PL=PL2 pour $fact = [1, 2]$
- Affichage des calculs ratés avec les valeurs associées

Il était donc nécessaire de modifier l'accélérateur pour qu'il accepte toutes les tailles de tableaux.

e. Résolution du problème de l'accélérateur

L'accélérateur Approx_PL sur *Vitis HLS* a été modifié pour utiliser des « Stream » en entrée. Cela devrait lui permettre de fonctionner pour tous les SF, et pour toutes les valeurs de *fact*.

Le *Test Bench* vérifiait ensuite que l'accélérateur fonctionnait sur 100 valeurs avec :

$SF = [7, 8, 9, 10, 11, 12] \quad fact = [1,2]$

En simulation C++, l'accélérateur fonctionnait :

```
err=0        n=0
M=[128]      fact=[1]    PL=484.55       PS=484.55
err=0        n=28
M=[128]      fact=[2]    PL=6243.59      PS=6243.59
err=0        n=99
M=[4096]     fact=[2]    PL=200457.25    PS=200457.25
```

Figure 47 – Différents tests sur l'accélérateur

En rajoutant manuellement une erreur de 0.01 au résultat PL, on obtient bien des erreurs :

```
ERR// err=1182       n=99
M=[4096]     fact=[2]    PL=200457.27    PS=200457.25
```

Figure 48 – Erreurs ajoutées manuellement pour vérifier le comportement du test

L'implémentation de l'accélérateur utilise les ressources suivantes :

	Verilog
SLICE	923
LUT	2004
FF	3339
DSP	15
BRAM	0
URAM	0
LATCH	0
SRL	34
CLB	0

Figure 49 – Empreinte matérielle de l'accélérateur

Le coût matériel est important, mais n'est pas surprenant car tous les calculs sont réalisés sur des flottants. Cette empreinte ne sera presque pas réduite lors de ce projet car l'algorithme sera très peu modifié.

Avec l'accélérateur, la vérification semble fonctionner en SF10 pour $fact = [1, 2]$, les trames sont démodulées.

3. Vérification du bon fonctionnement

La fonction « verify_PL » a été modifiée pour comparer l'accélérateur pour chaque SF avec les deux facteurs.

Introduction d'erreurs intentionnelles :

```
(II) Accelerators verify
(II) ---------------------------------------------
Approx_PL test ...
M=[ 128]    fact=[1 2]      ERR
M=[ 256]    fact=[1 2]      ERR
M=[ 512]    fact=[1 2]      ERR
M=[1024]    fact=[1 2]      OK
M=[2048]    fact=[1 2]      ERR
M=[4096]    fact=[1 2]      ERR
Approx_PL test FAILED
```

Fonctionnement normal de l'accélérateur :

```
(II) Accelerators verify
(II) ---------------------------------------------
Approx_PL test ...
M=[ 128]    fact=[1 2]      OK
M=[ 256]    fact=[1 2]      OK
M=[ 512]    fact=[1 2]      OK
M=[1024]    fact=[1 2]      OK
M=[2048]    fact=[1 2]      OK
M=[4096]    fact=[1 2]      OK
Approx_PL test OK
```

Figure 50 – vérification du bon fonctionnement des tests

La fonction « timing_PL » a elle aussi été modifiée pour vérifier les performances dans toutes les configurations.

La fonction donnait désormais les bons résultats.

4. Améliorations

Une fois que le code était fonctionnel, nous avons pu nous concentrer sur les améliorations possibles de l'IP.

Les améliorations peuvent être réalisées en différents points :

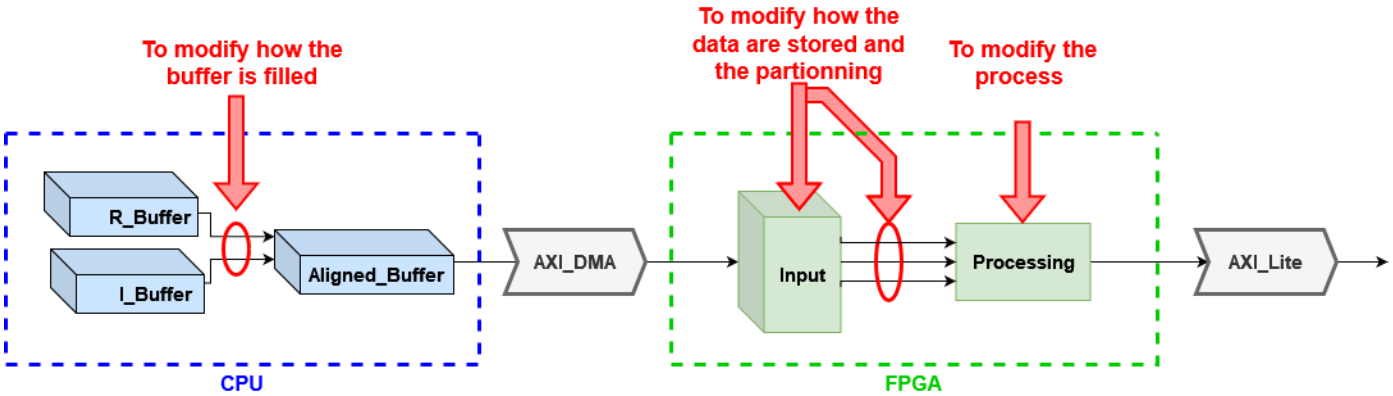


Figure 51 – Différents points d'amélioration

Dans un premier temps, nous avons décidé de nous concentrer sur la transmission des données entre CPU et FPGA. Les données peuvent être transmises par la DMA en tant que « float ». Dans le programme initial, toutes les données sont transmises comme des « int ». Il fallait donc réaliser plusieurs conversions :

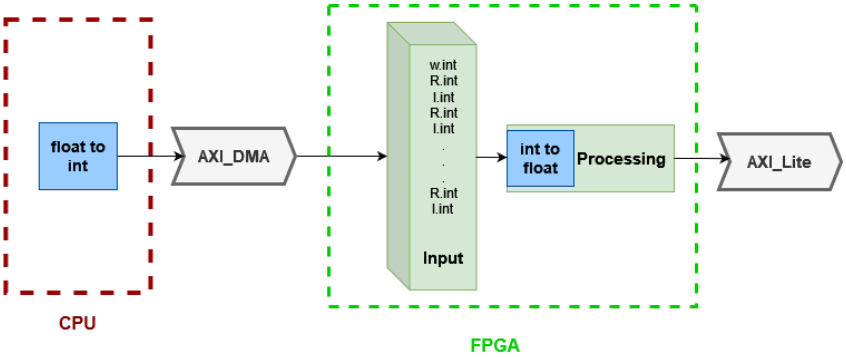


Figure 52 – Transmission des données

Nous avons donc modifié le programme et l'architecture pour transmettre directement des « float » :

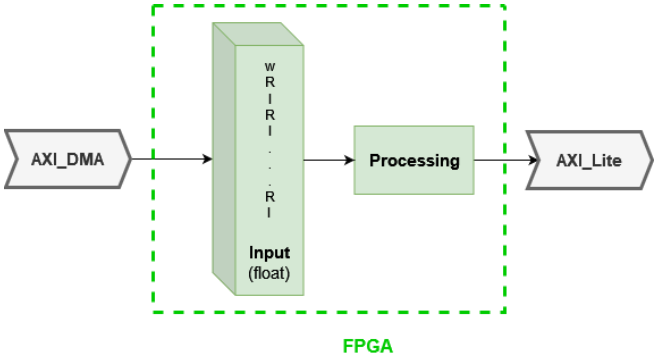


Figure 53 – Transmission des données après modification

Nous nous attendions à obtenir un fort gain de temps car les conversions prennent du temps. Cependant le résultat fut décevant :

		Before	After
Process	**Nb of I/Q**	**H_calcul/ms**	**H_calcul/ms**
PS	128	35.36	35.36
PL	128	56.64	57.16
PS	8092	0.55	0.55
PL	8092	1.108	1.106

Figure 54 – Résultat des architectures

Le gain était présent pour un faible nombre de données à traiter (bien que peu notable) mais était inexistant pour un grand nombre de données à traiter. Nous avons compris que l'architecture, en pipeline, permettait de ne pas perdre trop de temps lors des conversions. Nous avons tout de même choisi de conserver l'architecture transmettant uniquement des « float » car cette modification permettait de simplifier l'écriture des codes.

La deuxième amélioration à laquelle nous avons pensé concernait l'écriture du buffer aligné par le CPU. Le buffer était rempli octet par octet, ce qui signifiait que pour chaque valeur il fallait venir écrire quatre fois :

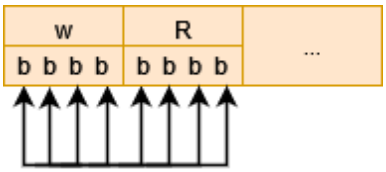


Figure 55 – Ecriture dans la mémoire alignée avec des pointeurs de 1 octet

Nous avons pensé que s'il était possible d'écrire quatre octets à la fois, nous pourrions gagner du temps.

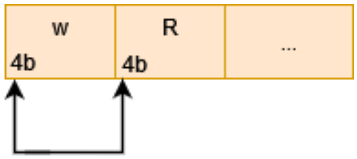


Figure 56 – Ecriture dans la mémoire avec des pointeur de 4 octets

Nous codons donc ce nouveau comportement et on étudie le résultat. Avec cette nouvelle méthode, nous avons un gain de temps assez notable :

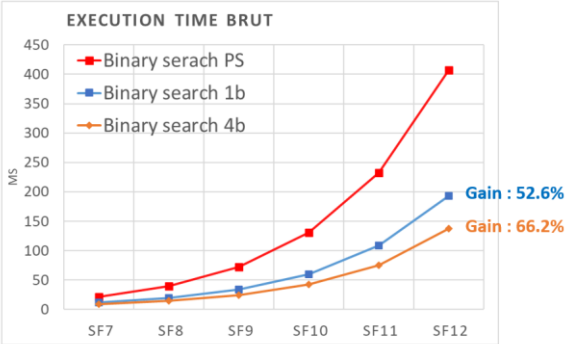


Figure 57 – Temps d'exécution pour différentes versions du code

La figure ci-dessus illustre le gain de temps obtenu grâce à cette modification. La courbe rouge correspond à une dichotomie réalisée sur CPU. La courbe bleue est une dichotomie pour laquelle la mémoire a été écrite octet par octet et en orange à une mémoire remplie par groupe de quatre octets. Nous passons d'un gain de 52% à un gain de temps de 66% pour SF12.

Une autre piste d'amélioration a été étudiée : au lieu d'utiliser un *FIFO* 32 bits en entrées de approx PL on peut utiliser une *FIFO* 64 bits :

Figure 58 – Passage d'une FIFO 32b à une FIFO 64b

Cette amélioration entraîne une légère modification de l'écriture en mémoire : en effet afin de stocker les couples R/I ensemble, il est nécessaire d'envoyer deux fois *w*.

Nous espérions obtenir un fort gain de temps avec cette amélioration mais nous n'avons jamais réussi à l'implémenter.

Au début, toutes les trames sont lues, puis, temporairement, quelques trames ne sont pas bien décodées. Après ça, tout se déroule correctement avant que le code ne défaille totalement et que plus rien ne soit correct.

Nous avons essayé de trouver l'origine du problème, sans succès. Nous avons étudié le code sous Vitis HLS et modifié le *Test Bench* pour faire plus de tests. L'architecture semblait correcte. Sur *Vivado* nous avons modifié les paramètres de la DMA (largeur des adresses, *Max Burst Size*, *Memory Map Data Width*, ...) mais sans succès. Nous n'avons pas non plus trouvé de problème dans Vitis.

f. Résultats finaux

Les résultats pour chaque version sont donc les suivants :

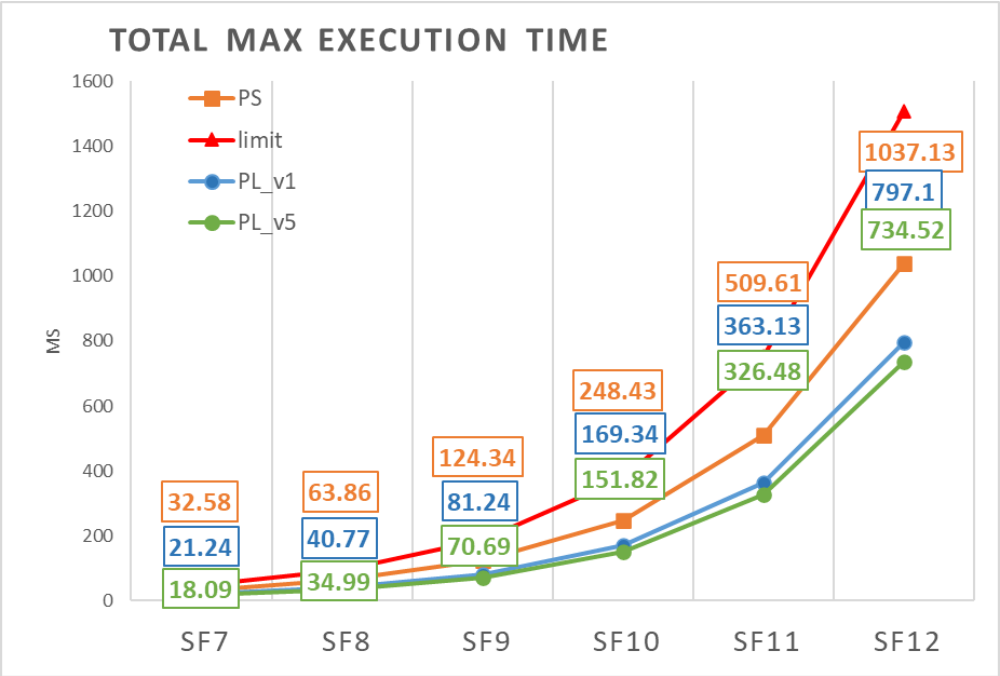


Figure 59 – Résultats pour différentes versions

La courbe en rouge représente le temps d'exécution limite pour la réalisation de la dichotomie. Il est impératif d'avoir un temps inférieur à cette limite pour être temps réel. Si le système ne respecte pas ce critère, il ne pourra pas décoder l'emble des trames que nous lui envoyons.

La courbe orange représente le temps d'exécution maximal nécessaire pour réaliser la dichotomie par un CPU. Ce temps, bien qu'inférieur à la limite pouvait être amélioré.

La courbe bleue représente la première approche de l'algorithme sur FPGA. C'est la courbe correspondant à l'état du projet avant que nous ne le prenions en main.

La courbe verte représente le temps maximal d'exécution avec l'amélioration réalisée, dans laquelle nous écrivions 4 bits par 4 bits dans le buffer aligné. Ce temps peut sembler négligeable, mais il semble difficile d'améliorer d'avantage ce résultat.

IX. Accélérateur matériel de FFT

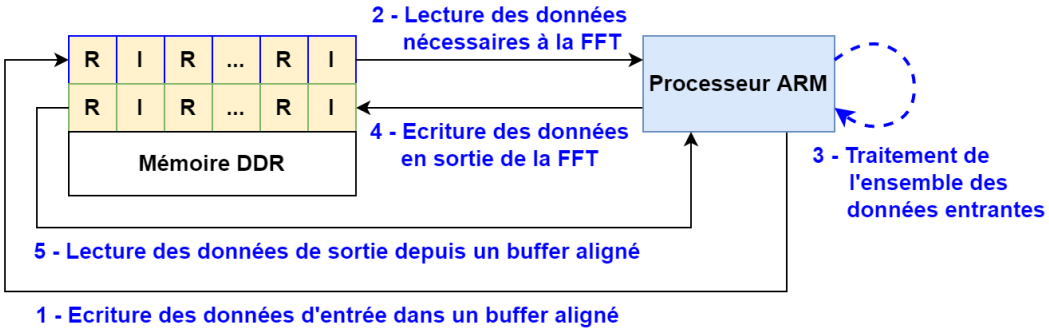
Cette section se porte sur l'accélérateur matériel (IP) réalisant des calculs de transformées de Fourier rapides (FFT). Plus particulièrement, elle se porte sur la prise en main logicielle (sur *Vitis*) de l'IP FFT et sur le nettoyage et l'enrichissement progressif du code C++ associé. Elle va également décrire les optimisations réalisées et les performances obtenues par l'IP (côté PL) et le processeur (côté PS) dans diverses situations. Enfin, les futures améliorations possibles, du code C++ voire de l'IP FFT elle-même (sur le code *HLS*) seront expliquées.

1. Introduction

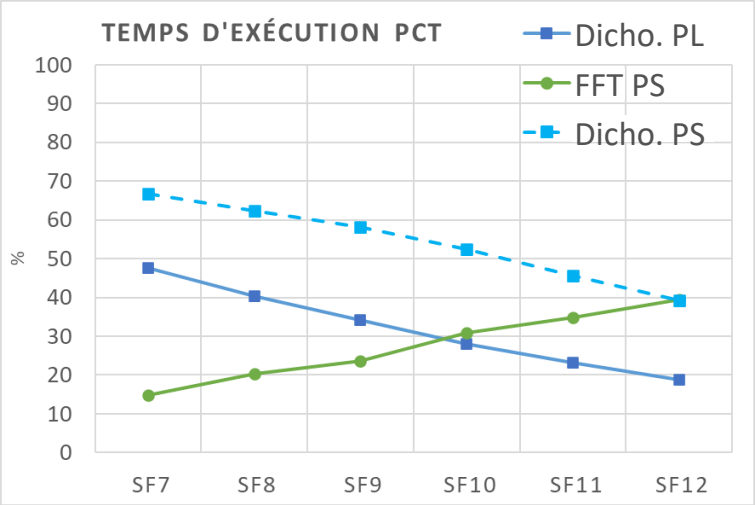
Dans l'exécution d'un calcul de FFT côté PS, le processeur *ARM*, centre de la carte et ordonnanceur des opérations matérielles, va allouer deux buffers « alignés » en mémoire. En effet, le calcul d'une FFT discrète sur M échantillons complexes (soit $2 \times M$ valeurs réelles et complexes pures) nécessite de connaître l'ensemble de ces M échantillons à tout instant. Afin de ne pas corrompre le buffer d'entrée, le résultat est ainsi stocké dans un buffer de sortie, à partir duquel d'autres calculs peuvent être réalisés (magnitude maximale, position, etc.).

Pour nos besoins de détection d'énergie sur des trames LoRa DCSS (voir parties précédentes), le nombre d'échantillons complexes à traiter M correspond au nombre de symboles possibles par trame LoRa selon le facteur d'étalement de spectre *SF*, par la relation $M = 2^{SF}$. Ainsi, les buffers alloués en mémoire sont de longueur 2×2^{SF} (ou 2^{SF+1}) en entrée comme en sortie. Cela sera vrai côté PS et pour l'IP FFT (côté PL).

Une fois les allocations réalisées, le processeur écrit les M échantillons complexes à traiter dans le buffer d'entrée, puis appelle les méthodes de la classe PS « *fft* » afin de réaliser le calcul de FFT (puis les autres calculs nécessaires). Une fois les calculs réalisés, l'objet de la classe « *fft* » est détruit, et les allocations sont libérées. L'algorithme de FFT utilisé est relativement performant, mais utilise du précieux temps d'exécution processeur. L'exécution d'un calcul de FFT côté PS peut être vue comme suit :

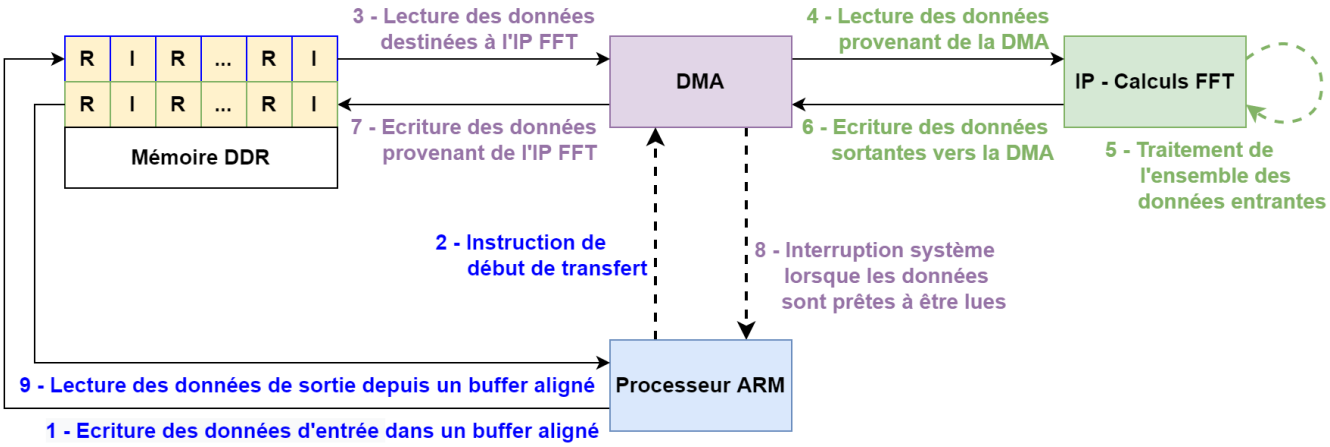


Bien que cet algorithme soit fonctionnel, réaliser un calcul aussi complexe en PS est détrimentaire aux performances temps-réel du Récepteur, l'algorithme gérant la démodulation d'une trame LoRa. Grâce à l'IP de calcul dichotomique (voir partie précédente), le système est désormais viable pour une application temps-réel, mais la marge de temps d'exécution reste faible. D'après la figure suivante, nous constatons que le calcul de FFT occupe désormais plus de temps processeur à partir du facteur de spectre *SF* 10 que le calcul dichotomique :



Afin d'élargir la marge de temps d'exécution nécessaire à la performance temps-réel du Récepteur, l'utilisation d'un accélérateur matériel dédié aux calculs de FFT se présente comme une option avantageuse, aussi bien du point de vue temporel que du point de vue matériel. Les cartes Pynq (développement) et Ninano (vol) étant toutes deux des *SoC* FPGA, un accélérateur matériel permet de prendre avantage de leur architecture tout en réduisant le temps processeur utilisé. Nous allons donc implémenter un accélérateur de FFT.

L'IP FFT est une description FPGA d'un accélérateur matériel dont le but est de réaliser des calculs de FFT entièrement en PL, indépendamment du processeur *ARM* (PS). Conçue par le doctorant Hugues, cette IP a été construite par un procédé de synthèse de haut-niveau (*HLS*) à partir d'un code source en C++, disponible sur le répertoire GitHub suivant. Suivant les éléments matériels de la Zynq, l'exécution d'un calcul de FFT à l'aide de cet accélérateur matériel peut être vue comme suit :



Afin de prendre plein avantage de l'IP, dont les données entrantes et sortantes transitent par un bus *AXI Stream*, le calcul de FFT en PL utilise un contrôleur d'accès mémoire direct (*DMA*). Lorsque le buffer de données entrantes est rempli par le processeur *ARM* (étape 1), celui-ci ordonne au contrôleur de *DMA* (étape 2) de commencer le transfert bidirectionnel de données entre la mémoire (buffers d'entrée et de sortie) et l'IP FFT (bus d'entrée et de sortie). Ainsi, le contrôleur de *DMA* seul gère les opérations de l'IP FFT, et non le processeur.

Une fois l'instruction de transfert lancée (étape 2), le contrôleur de *DMA* lit les données du buffer d'entrée en mémoire (étape 3) puis les transférer au bus d'entrée de l'IP FFT (étape 4). Une fois les M valeurs complexes ($2 \times M$ valeurs totales) transférées, l'IP réalise le calcul de FFT (étape 5) avant de renvoyer M valeurs complexes sur son bus de sortie. Ces données sont lues par le contrôleur de *DMA* (étape 6) avant d'être écrite en mémoire sur le buffer de sortie (étape 7). Une interruption système (étape 8) signale la fin du transfert.

Certaines technicités, cependant, créent des problèmes. D'abord, l'IP FFT utilise des nombres en virgule fixe (format *len<2, 14>*) tandis que le Récepteur actuel utilise des nombres à virgule flottante. Le passage de l'un à l'autre nécessite une conversion coûteuse en temps processeur. D'autre part, l'IP FFT utilisée durant ces travaux ne peut transformer au plus que M=1024 valeurs complexes, limitant les calculs de FFT que jusqu'au facteur de spectre *SF* 10. Nous identifions les avantages et inconvénients des calculs PS ou PL ci-dessous :

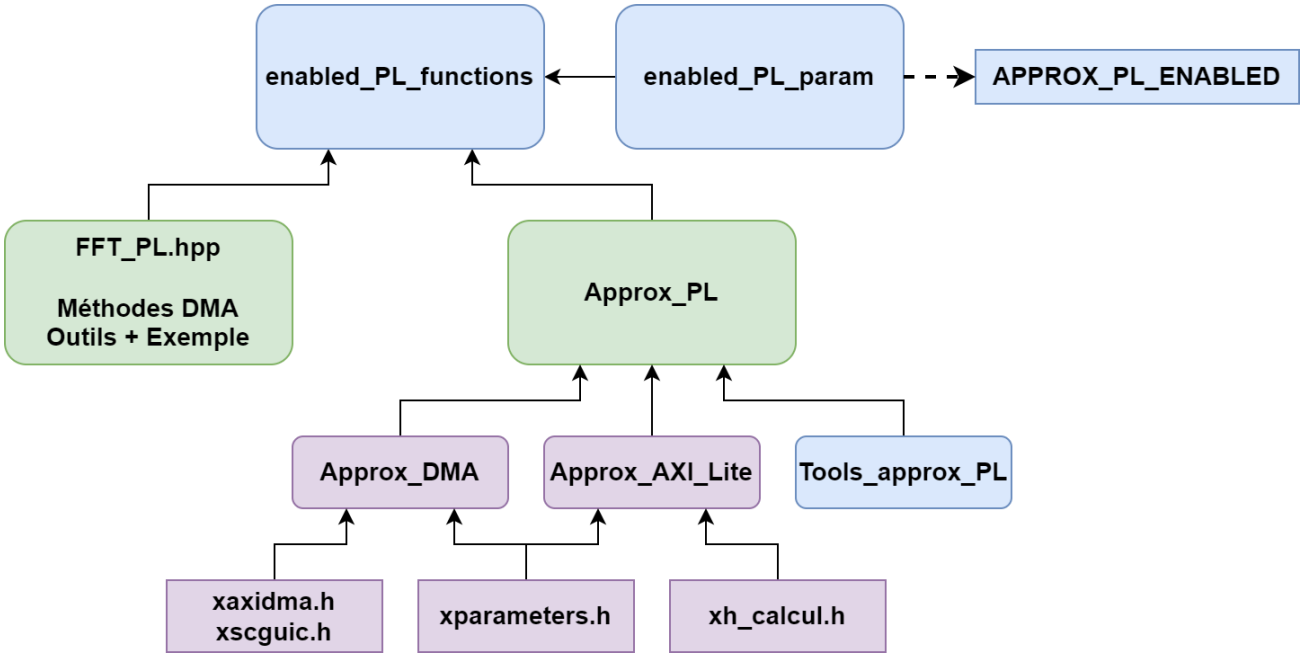
Calculs côté PS (classe « *fft_pfft_float* » héritée de « *fft* »)	**Calculs côté PL** (accélérateur matériel de FFT)
Avantages : - **Homogénéité** des formats (virgule flottante), pas de conversions coûteuses à réaliser - Méthode **déjà utilisée** par le Récepteur - Supporte tous les facteurs de spectre ***SF* de 7 à 12**	Avantages : - Décharge une opération très coûteuse du processeur, **augmentant les marges** de sécurité à la contrainte temps-réel - Utilise des **valeurs à virgule fixe**, qui pourraient être employées dans le futur
Inconvénients : - Large portion du temps processeur occupé, **réduisant les marges** de sécurité à la contrainte temps-réel imposée au Récepteur - Ne supporte que les nombres à virgule flottante, **pas d'évolution possible** si le Récepteur passe au format en virgule fixe (calculs plus efficient)	Inconvénients : - **Non-homogénéité** des formats (virgule flottante) avec le Récepteur actuel, ce qui demande des conversions coûteuses - Méthode **non-testée** sur le Récepteur - Ne supporte que les facteurs de spectre ***SF* 10 ou inférieurs**, pas *SF* 11 ou 12.

2. Analyse et nettoyage du code d'exploitation original

Maintenant que nous connaissons les enjeux et les principaux avantages et inconvénients des calculs en PS et PL, nous pouvons commencer notre travail par une analyse de code. Au début de cette étude, la plateforme matérielle contenant l'IP FFT était fournie avec un fichier source C++ nommé « FFT_PL.hpp ». Rédigé par Hugues et Valentin Romero à l'IMS Bordeaux, il s'agit d'un fichier monolithique permettant de faire fonctionner l'IP FFT pour réaliser des calculs de FFT. Le fichier contient les méthodes suivantes :

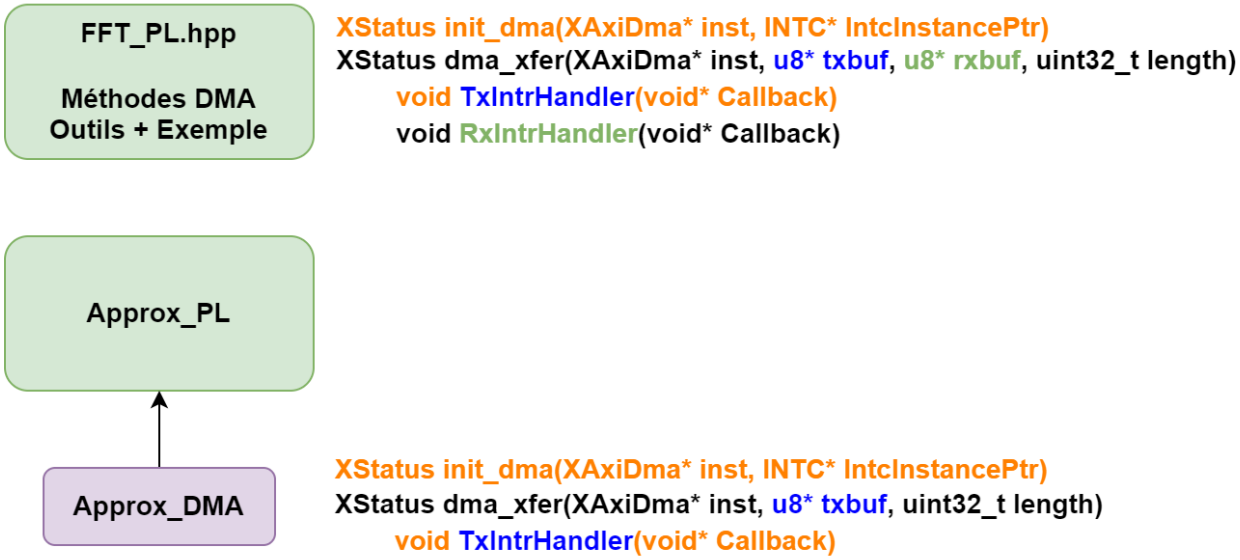
Méthode	Description
FFT_PL.hpp	
int **fft_PL_init** ()	Méthode d'**initialisation** de l'IP FFT
int **fft_PL_process** (u8* inBuf, u8* outBuf)	Méthode d'**exécution** de l'IP FFT, pointant sur les buffers d'entrée et de sortie utilisés par l'IP
int **FFT_Timing** ()	Méthode de **chronométrie** de l'IP FFT
int **fft_PL_example** ()	Méthode permettant de calculer la **FFT d'un signal** sinusoïdal réel de période discrète 1/1024
XStatus **init_dma** (XAxiDma* inst, INTC* IntcInstancePtr)	Méthode d'**initialisation** du contrôleur de ***DMA***, utilise des objets *XAxiDma* et *INTC* déclarés plus haut
XStatus **dma_xfer** (XAxiDma* inst, u8* txbuf, u8* rxbuf, uint32_t length)	Méthode de **transfert de données** via le contrôleur de ***DMA***, appelée par fft_PL_process et utilise les buffers d'entrée et de sortie de l'IP ainsi que leur taille
void **TxIntrHandler** (void* Callback)	Méthode de **gestion d'interruptions** système venant du contrôleur de *DMA* dans le **sens d'envoi**
void **RxIntrHandler** (void* Callback)	Méthode de **gestion d'interruptions** système venant du contrôleur de *DMA* dans le **sens de réception**

D'après Valentin Romero, ce fichier source peut être utilisé pour faire fonctionner la plateforme *Vivado* (fichier .xsa) contenant l'IP FFT seule, sans l'IP dichotomique. Cependant, l'objectif de nos travaux est de pouvoir utiliser les deux IP simultanément pour accélérer le fonctionnement du Récepteur. Ainsi, nous avons immédiatement procédé à une inclusion naïve du fichier dans l'arborescence des codes d'exploitation des accélérateurs matériels, en parallèle aux sources de Approx_PL et sous la source « enabled_PL_functions » :

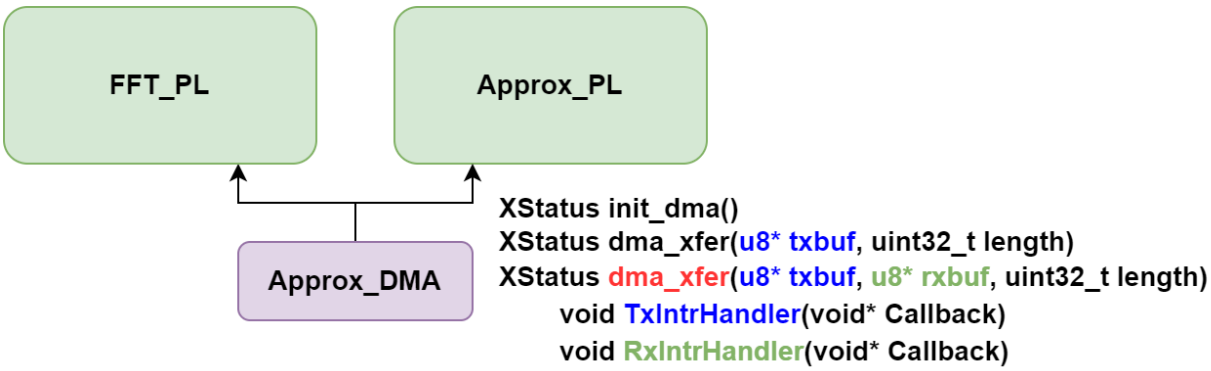


Malgré le bon fonctionnement de l'inclusion, un grand nombre de problèmes apparurent. En premier lieu, le fichier « FFT_PL.hpp » n'est pas écrit selon les mêmes règles de conception que les sources de Approx_PL, notamment sur la définition de constantes (SAMPLES_PER_TRANSFER, NFFT, XFER_SIZE, etc.) et les arguments passés aux fonctions appelées (absence d'un argument *M* ou *length* sur les méthodes, etc.). Une réécriture de l'ensemble de ces méthodes va donc être nécessaire afin de se conformer au standard de Approx_PL.

En second lieu, les méthodes *DMA* écrites sur « FFT_PL.hpp » présentent à la fois des dupliqués et des polymorphismes de méthodes écrites dans le code source Approx_DMA employé par Approx_PL. L'inclusion double de ces méthodes dans la source enabled_PL_functions génère des erreurs, et le polymorphisme d'une méthode décrite dans deux sources pour deux processus est indésirable :



Les erreurs d'inclusion et le polymorphisme décentralisé peuvent être résolus par la fusion des méthodes *DMA* de FFT_PL dans Approx_DMA. Cette source va maintenant contenir l'ensemble des méthodes *DMA* nécessaires aux deux accélérateurs matériels, avec la distinction de la méthode polymorphique dma_xfer, qui permet un transfert bidirectionnel pour l'IP FFT et un transfert unidirectionnel pour l'IP dichotomique. Les objets *XAxiDma* et *INTC* sont aussi déclarés de manière globale. La nouvelle répartition des méthodes est comme suit :

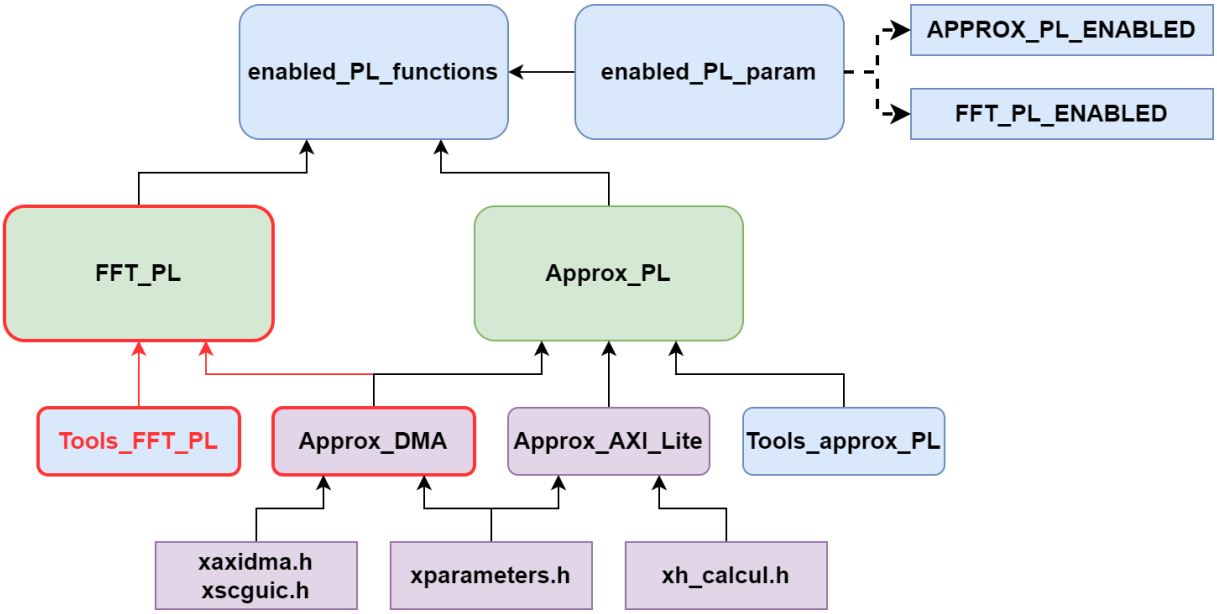


À noter que le fonctionnement de Approx_DMA a été validé pour des plateformes *Vivado* contenant soit l'IP FFT, soit l'IP dichotomique, mais pas les deux en même temps. Dans le futur, l'usage de deux contrôleurs *DMA* sur le *Block Design* de l'outil *Vivado* devra être accompagné de modifications sur Approx_PL. La méthode dma_init devra prendre en compte les deux contrôleurs de *DMA*, et les autres méthodes (transfert et interruptions) devront utiliser la bonne instance de *DMA* et les définitions systèmes associées.

Maintenant que les conflits de méthodes *DMA* sont réglés, nous devons conformer le code source de FFT_PL aux règles suivies par Approx_PL et enabled_PL_functions. Notamment, un grand nombre de variables (*SAMPLES_PER_TRANSFER*, *XFER_SIZE*, *NFFT*) de FFT_PL sont définies en préprocesseur, alors qu'Approx_PL réalise les calculs à l'intérieur de chaque méthode. Pour une perte marginale de performance (deux produits d'entiers tout au plus par méthode), les méthodes des deux IP deviennent plus sûres et plus flexibles.

La dernière modification majeure a consisté à déplacer la méthode fft_PL_example dans un nouveau fichier source, nommé Tools_FFT_PL. Calqué sur Tools_Approx_PL, il contient une méthode fft_PL_generator (modification de approx_PL_generator) utilisée par fft_PL_example pour générer un signal à transformer par l'IP FFT. Cela permet de laisser FFT_PL aux quatre méthodes principales : fft_PL_init, fft_PL_process, fft_PL_timing, et fft_PL_verify. Les trois premières ont été modifiées, et la quatrième inspirée de Approx_PL.

Après le nettoyage et la fragmentation du fichier source FFT_PL, l'arborescence des codes d'exploitation des accélérateurs matériels peut être vue comme suit :



La source enabled_PL_param contient désormais un nouveau signal de contrôle préprocesseur : *FFT_PL_ENABLED* permet l'usage de l'ensemble des méthodes PL (et *DMA*) associées à l'IP FFT, tandis qu'*APPROX_PL_ENABLED* remplit le même rôle pour l'IP dichotomique. La seule définition préprocesseur commune aux deux IP est *SAMPLE_SIZE*, qui fixe la taille (en octets) des données échangées entre les IP et le processeur. Pour les deux IP, ces données sont stockées sur 4 octets durant le transfert *DMA*.

En fin de nettoyage, les méthodes présentes dans les sources de l'IP FFT sont les suivantes :

Méthode	Description
FFT_PL	
int **fft_PL_init** ()	Méthode d'**initialisation** de l'IP FFT
int **fft_PL_process** (int M, u8* inBuf, u8* outBuf)	Méthode d'**exécution** de l'IP FFT, pointant sur les buffers d'entrée et de sortie utilisés par l'IP
int **fft_PL_timing** ()	Méthode de **chronométrie** de l'IP FFT
int **fft_PL_verify** ()	Méthode de **vérification** de l'IP FFT
Approx_DMA	
XStatus **init_dma** ()	Méthode d'**initialisation** du contrôleur de ***DMA***, commune et ajustée au fonctionnement des deux IP
XStatus **dma_xfer** (u8* txbuf, u8* rxbuf, uint32_t length)	Méthode de **transfert de données** via le contrôleur de ***DMA***, appelée par **fft_PL_process** et utilise les buffers d'entrée et de sortie de l'IP ainsi que la taille
XStatus **dma_xfer** (u8* txbuf, uint32_t length)	Méthode de **transfert de données** via le contrôleur de ***DMA***, appelée par **approx_PL_process** et utilise le buffer d'entrée de l'IP ainsi que sa taille
void **TxIntrHandler** (void* Callback)	Méthode de **gestion d'interruptions** système venant du contrôleur de *DMA* dans le **sens d'envoi**
void **RxIntrHandler** (void* Callback)	Méthode de **gestion d'interruptions** système venant du contrôleur de *DMA* dans le **sens de réception**
Tools_FFT_PL	
int **fft_PL_generator** (int M, u8* inBuf, vector<float> inR, vector<float> inI)	Méthode de **génération de signal en entrée** pour méthodes de calcul FFT en PS (vecteurs de nombres à virgules flottantes) ou PL (pointeur d'octets)
int **fft_PL_example** ()	Méthode permettant de calculer et afficher la **FFT d'un signal** généré par la méthode fft_PL_generator

3. Compatibilité entre IP et plateformes

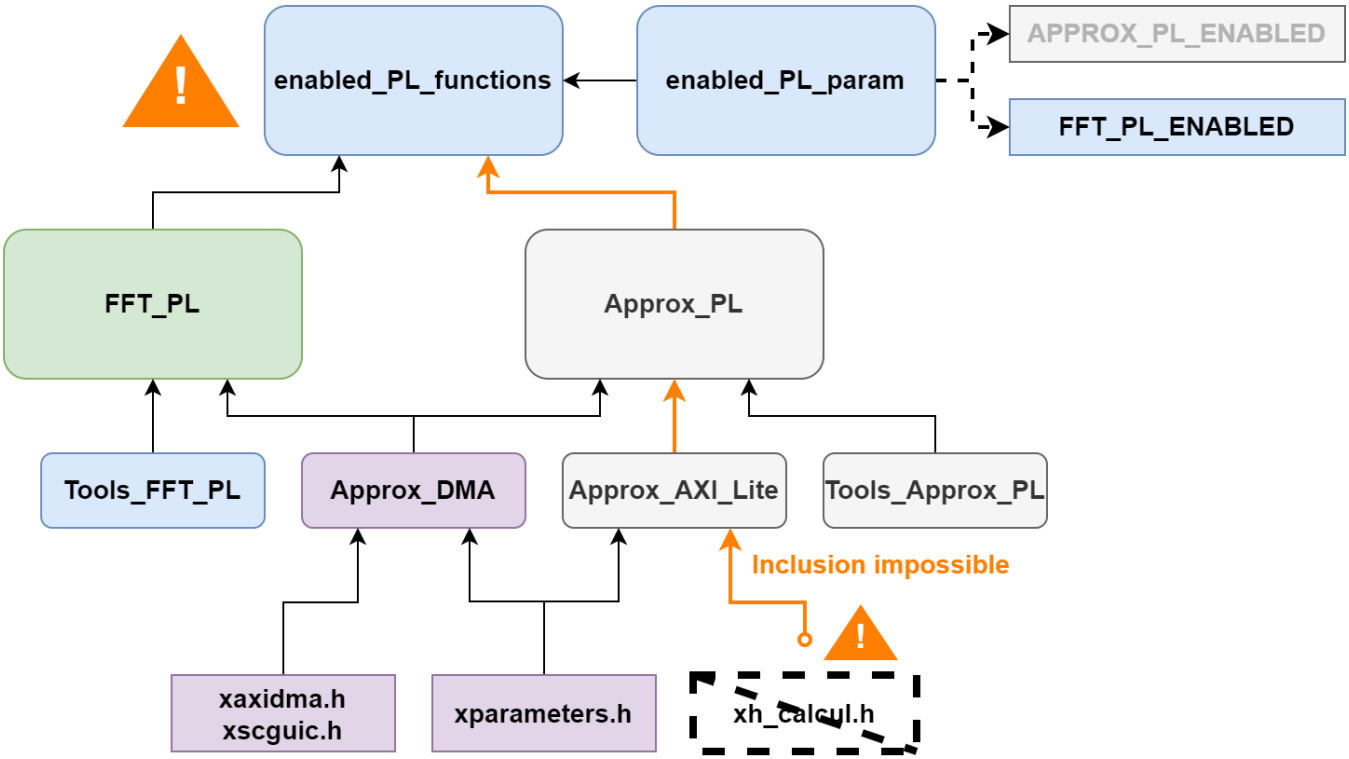
Dans la future implémentation des accélérateurs matériels du récepteur, l'IP FFT et l'IP dichotomique seront toutes les deux présentes sur une seule et même plateforme *Vivado* (fichier .xsa), contenant leur description matérielle ainsi que les librairies systèmes nécessaires à leur fonctionnement. Pour l'heure cependant, chaque IP existe sur une plateforme séparée, et réaliser des tests sur une IP revient donc à utiliser la plateforme qui lui est associée. Il faut ainsi garantir que le code d'exploitation de l'autre IP n'interfère pas.

Pour cela, le fichier source C++ enabled_PL_functions utilise un autre fichier, nommé enabled_PL_param, dont le rôle est de définir quelle IP matérielle est actuellement utilisée. Pour cela, deux signaux peuvent y être définit ou commentés (non-définis) en préprocesseur : *APPROX_PL_ENABLED* et *FFT_PL_ENABLED*. L'intérêt est de centraliser le code permettant de choisir l'IP actuellement utilisée en une seule définition dans un seul fichier facilement identifiable. Dans le futur, cela permettra aussi de désactiver une des deux IP au besoin.

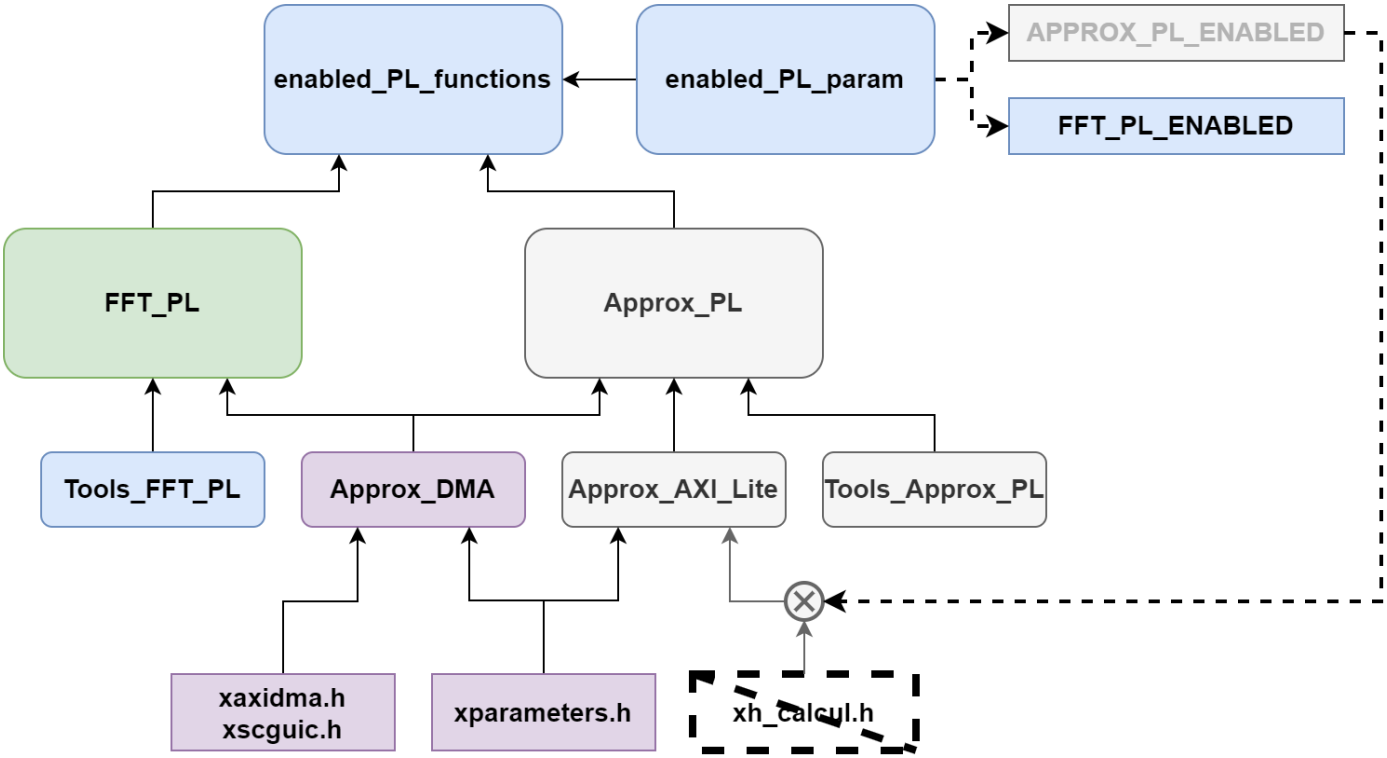
Dans l'arborescence des codes d'exploitation, nous avons constaté que certaines méthodes de Approx_PL utilisent une structure #ifndef en début de déclaration, qui permet de « désactiver » la méthode lorsqu'*APPROX_PL_ENABLED* n'est pas définit. Afin de garantir la compatibilité de toutes les méthodes, et d'introduire une sécurité dans le cas où une méthode incorrecte est appelée, nous avons donc appliqué ces structures #ifndef à l'ensemble des méthodes présentes dans les sources de l'IP FFT et de l'IP dichotomie :

```
int fft_PL_process(int M, u8* inBuf, u8* outBuf)
{
#ifndef FFT_PL_ENABLED
    cout << KRED << "fft_PL_process() was called but FFT_PL is disabled" << KNRM << endl;
    return EXIT_FAILURE;
#else
    //
    // ...
    //
    return EXIT_SUCCESS;
#endif
}
```

Durant la fin du nettoyage, un problème similaire a fait surface : durant les tests de l'IP FFT sur la plateforme *Vivado* contenant cette IP, la non-définition de *APPROX_PL_ENABLED* ne suffisait pas à désactiver entièrement les sources de Approx_PL. Spécifiquement, une inclusion réalisée dans la source Approx_AXI_Lite devenait impossible sans la plateforme contenant l'IP de dichotomie, car la librairie de définitions système de l'IP de dichotomie (« xh_calcul.h ») pour le bus *AXI Lite* n'existe pas sur la plateforme contenant juste l'IP FFT :



Puisque, dans le futur, ce problème n'aura plus lieu (les deux accélérateurs seront présents sur la même plateforme *Vivado*), il fut résolu en transformant l'inclusion standard en inclusion conditionnelle, par le biais de l'instruction préprocesseur #ifdef *APPROX_PL_ENABLED*. Aucune inclusion conditionnelle n'est nécessaire pour l'IP FFT, puisqu'elle n'utilise que les fichiers systèmes de la *DMA*. Ainsi, les tests réalisés au sol sur une plateforme *Vivado* ne contenant qu'une seule IP ne viendront plus être perturbés par les sources de l'autre IP :



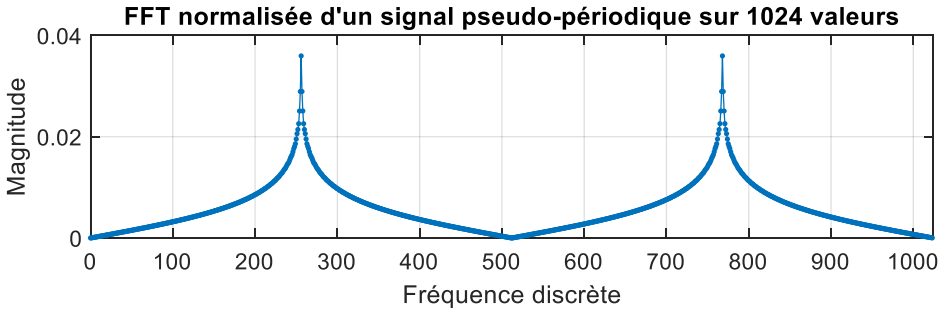
Désormais, l'ensemble des méthodes décrites dans FFT_PL, Approx_PL et leurs dépendances utilisent une structure #ifndef de sécurité observant leur signal de contrôle respectif. De plus, l'inclusion conditionnelle de la librairie « xh_calcul.h » dans Approx_AXI_Lite utilise une structure similaire. Ainsi, un programme souhaitant appeler une méthode PL définit ou incluse dans enabled_PL_functions aura accès à une méthode fonctionnelle seulement si l'IP employée est active. Le cas échéant, une erreur est renvoyée avec un retour écrit sur console.

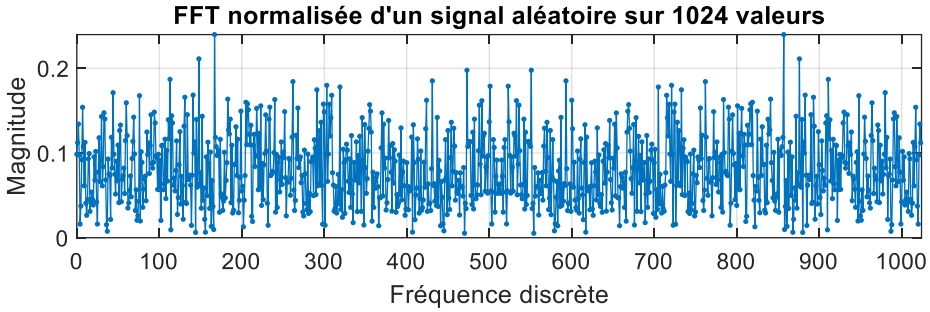
Ce mécanisme de désactivation conditionnelle a été testé avec le Récepteur entier dans plusieurs cas d'usage : définition ou non-définition de *APPROX_PL_ENABLED* sur la plateforme de l'IP dichotomique, définition ou non-définition de *FFT_PL_ENABLED* sur la plateforme de l'IP FFT, double définition et double non-définition sur les deux plateformes. Seules les erreurs attendues dans le cas de Approx_PL ont été relevées sur la plateforme contenant de l'IP FFT. Les deux IP sont donc jugées comme compatibles à l'écriture de ce rapport.

4. Vérification : protocole et résultats pour PS et PL

Maintenant que les codes source des deux accélérateurs matériels sont compatibles, nous pouvons commencer nos tests de l'IP FFT de Hugues, en utilisant d'abord la méthode fft_PL_verify. Pour vérifier le bon fonctionnement de l'accélérateur matériel, nous comparons ses résultats avec ceux de la classe « fft », qui sera notre « golden model ». Les deux méthodes utilisent le même signal d'entrée, composé de M=1024 valeurs complexes aléatoires dont les composantes réelles et imaginaires sont bornées entre -5 et +5 exclus.

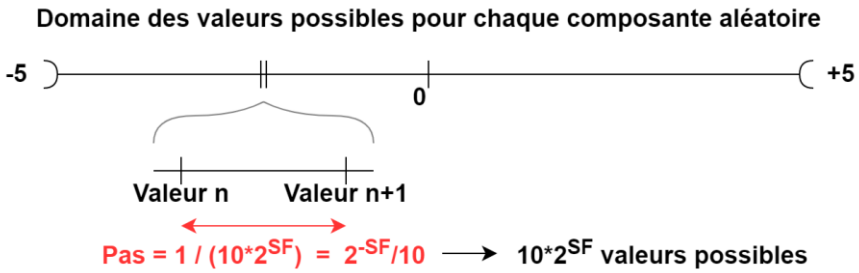
L'usage d'un signal d'entrée tel permet de placer l'IP dans la pire situation possible. Un signal réel présentera une répartition d'énergie centrée autour d'une fréquence discrète particulière (correspondant au symbole transmis), et son pic est donc « facile » à identifier. Un signal dont les valeurs, à composantes réelles pures et complexe pures, sont générées pseudo-aléatoirement a une dispersion d'énergie beaucoup plus chaotique. Trouver le même pic maximal d'énergie en PS et PL revient donc au pire cas possible pour l'IP :





L'amplitude des composantes réelles et imaginaires n'est pas non plus anodine. En observant le Récepteur dans le programme principal, on constate (empiriquement) que les calculs de FFT se font toujours sur des valeurs complexes dont les composantes réelles et complexes sont bornées entre -5 et +5 exclus. Afin de générer un signal fiable à la réalité, nous bornons aussi nos composantes. Ces bornes semblent avoir choisies délibérément d'ailleurs, puisque la sortie de l'IP entre en « overflow » pour des valeurs d'entrée plus grandes.

Enfin, le pas de résolution du signal d'entrée a été choisie avec soin. Lors d'un tirage de la méthode rand, la donnée sortie est un entier. Afin d'obtenir un nombre réel, il faut alors multiplier le résultat par une constante, le convertir en nombre à virgule flottante, puis le diviser par la même constante. L'inverse de cette constante correspond alors au pas de résolution minimal de notre signal d'entrée, l'écart le plus faible entre deux composantes (réelles pures ou complexes pures) consécutives :



Afin de mettre l'IP dans un cas défavorable, le pas choisi est égal à un dixième de la résolution minimale d'un nombre à virgule fixe *Q<2, 14>*. Puisque l'IP utilise ce format dont la résolution est connue, choisir une donnée d'entrée plus précise donne un avantage aux calculs PS, qui utilisent des nombres en virgules flottante de résolution bien supérieure. Ainsi, en cas de réussite, non seulement l'IP aura prouvé sa robustesse, mais elle aura prouvé qu'utiliser des nombres à virgules flottantes n'est pas plus robuste que des virgules fixes.

Durant la grande majorité des tests, les résultats de l'IP et de la classe « fft » concordent sur la magnitude maximale et son indice. L'indice étant un entier, sa détermination est absolue (aucune incertitude possible) et on attend que ses valeurs calculées en PS et PL soient égales. La magnitude maximale, cependant, est soumise à une incertitude. En effet, les conversions de nombres à virgule flottante vers des nombres à virgule fixe entraînent une perte irréversible de résolution.

Ainsi, la méthode de vérification fft_PL_verify ne comptera une erreur de magnitude maximale uniquement lorsque les résultats PS et PL divergeront de plus de $4 \times 2^{-14} = 2^{-12}$. Cette valeur, déterminée empiriquement, n'est pas basée sur des calculs d'incertitude précis. Néanmoins, une incertitude de mesure de 2^{-12} sur le module maximal est jugée suffisamment faible pour assurer une excellente corrélation entre les résultats PS et PL sans corrompre l'exactitude des résultats de l'IP FFT.

Utiliser une mesure basée sur un écart pose un problème opérationnel : dans un cas extrême, deux valeurs complexes (d'indices différents) peuvent avoir un module dont la différence est inférieure à 4×2^{-14}. Comme discuté plus haut, ce cas est pratiquement impossible dans un signal réel dont la magnitude maximale se démarque des autres, mais peut apparaître dans des valeurs générées aléatoirement. Dans ce rare cas, la méthode de vérification autorise jusqu'à 4 erreurs PS-PL (1 par processus PL) avant de renvoyer une erreur.

Après avoir implémenté nos 4 méthodes de calcul PL (voir section suivante), nous pouvons réaliser nos tests de l'IP. Comme pour l'IP dichotomique, ce test est composé de 250 calculs PS et PL à partir de valeurs aléatoires, où les résultats de PS et PL sont comparés à chacune de ces 250 itérations. Si plus de 4 erreurs sont détectées sur l'ensemble de ces 250 itérations, la méthode fft_PL_verify renvoie une erreur. Sinon, la méthode se termine et renvoie un succès, informant le programme principal que l'IP FFT est fonctionnelle :

```
(II) Accelerators status
(II) ------------------------------------------
(II) FFT_PL    ENABLED :   M=[1024]    Conversion mode = ARM NEON
(II) APPROX_PL  DISABLED

(II) Accelerators verify
(II) ------------------------------------------
FFT_PL test ...
M=1024          ite = 0
M=1024          ite = 50
M=1024          ite = 100
M=1024          ite = 150
M=1024          ite = 200
M=[1024]     OK  - No errors on maximum modulus (or argument of modulus) calculations within 0.006104%
FFT_PL test OK
```

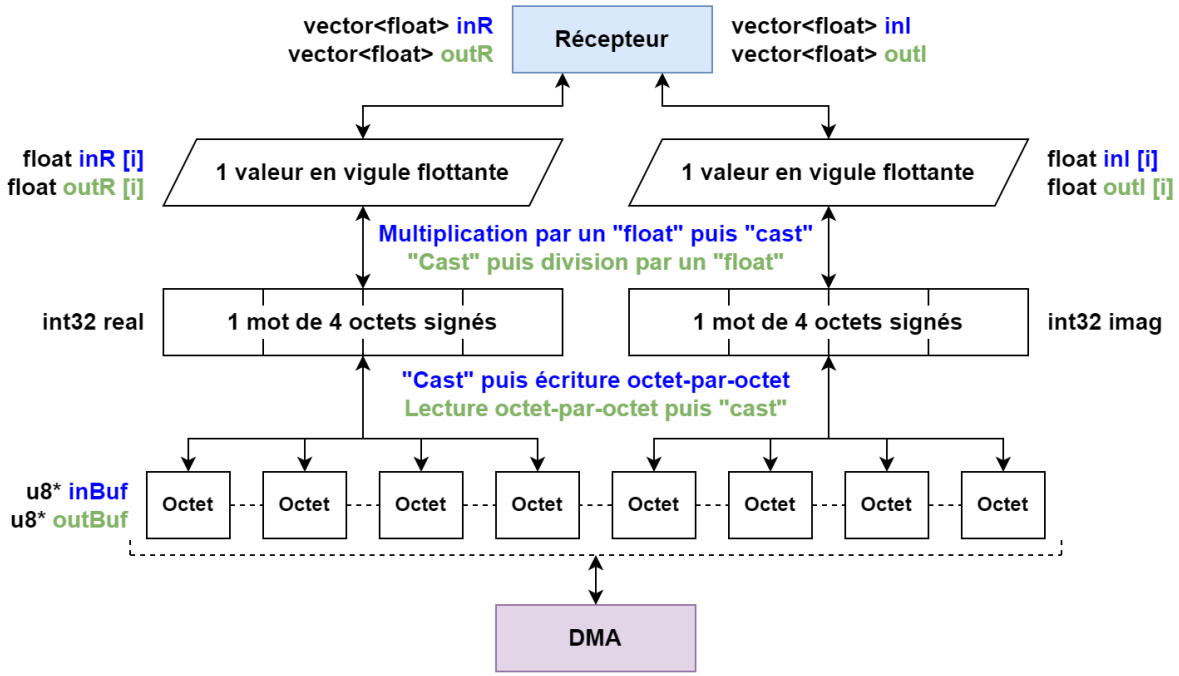
À noter que, du au nombre limité d'échantillons traitables par l'IP FFT (M=1024), l'ensemble des tests d'acuité ont été réalisés sur M=1024 valeurs d'entrées et de sortie, par praticité et afin d'utiliser un maximum de valeurs distinctes possibles. Des tests sur M<1024 valeurs sont possibles (en n'utilisant que M valeurs d'entrées non-nulles sur 1024 par exemple), mais une modification des méthodes de génération, vérification et chronométrie s'impose. La théorie étant assez simple, cette modification ne devrait donc être que bénigne.

5. Analyse des performances

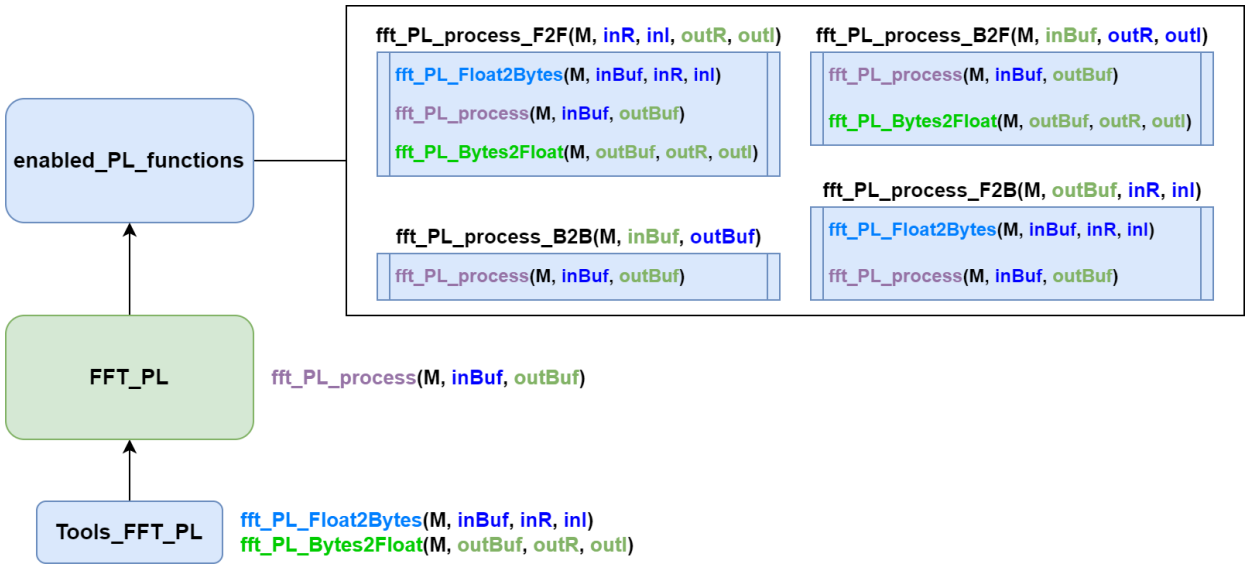
Maintenant que nous sommes assurés que l'IP FFT fonctionne, et que ses résultats sont compris dans des marges aux calculs PS acceptables, nous pouvons analyser sa performance. Il a été indiqué ci-dessus que fft_PL_verify réalise un calcul PS (classe « fft ») et 4 calculs PL par itération. Ces 4 méthodes PL ont servi à tester 4 modes de fonctionnement de l'IP FFT : octets-octets, flottant-octets, octets-flottant et flottant-flottant.

L'objectif du calcul PL est de retrouver le même module maximal que le calcul PS. Comme mentionné en introduction de cette partie, les calculs PS utilisent des nombres à virgule flottante (type C++ « float ») en entrée et en sortie de leurs méthodes (dénomination flottant-flottant). L'IP FFT, elle, utilise des nombres à virgule fixe encodés en binaire (type C++ « u8 ») sur 4 octets (dénomination octets-octets).

Afin de pouvoir comparer les calculs PS et PL, il est nécessaire de réaliser deux conversions de format : $ une conversion aller pour passer d'un « float » à 4 « u8 », puis une conversion retour pour repasser de 4 « u8 » à un « float ». Cependant, la conversion entre ces deux types n'est pas juste un « cast », mais une réinterprétation mathématique suivie d'une opération entre types « float » (multiplication ou division selon la conversion) :



Afin de visualiser l'impact réel de ces conversions flottant-octets et octets-flottant, nous avons donc écrit 4 méthodes distinctes dans enabled_PL_functions pour l'usage de l'IP FFT. Les 4 méthodes utilisent toutes le même appel du contrôleur de *DMA*, qui envoie puis reçoit des données de l'IP FFT, mais diffèrent dans la conversion de données. La méthode fft_PL_process_F2F réalise les deux conversions de type, les méthodes fft_PL_process_F2B et fft_PL_process_B2F n'en font qu'une, et la méthode fft_PL_process_B2B n'en réalise pas :



Maintenant que nos 4 méthodes de calcul distinctes sont définies (nous aborderons les méthodes de conversion fft_PL_Float2Bytes et fft_PL_Bytes2Float plus tard), nous allons les comparer. Afin d'estimer la performance des calculs PS et PL, nous employons la méthode fft_PL_timing de FFT_PL, qui est appelée depuis le programme principal par la méthode timing_enabled_PL de enabled_PL_functions. La métrique utile est le nombre de calculs de FFT réalisés par millisecondes, mais nous avons également son inverse :

```
Fin Test
FFT_PL test OK

(II) Accelerators timing
(II) ----------------------------------------
FFT_PL timing
Bytes-to-Bytes PL processed [2500] FFT for M=[1024]        |      [0.097507] ms/FFT      |      [10.255680] FFT/ms
Float-to-Bytes PL processed [2500] FFT for M=[1024]        |      [0.179333] ms/FFT      |      [5.576223] FFT/ms
Bytes-to-Float PL processed [2500] FFT for M=[1024]        |      [0.138021] ms/FFT      |      [7.245272] FFT/ms
Float-to-Float PL processed [2500] FFT for M=[1024]        |      [0.220707] ms/FFT      |      [4.530893] FFT/ms
Float-to-Float PS processed [2500] FFT for M=[1024]        |      [0.146762] ms/FFT      |      [6.813734] FFT/ms
```

Durant le test, chaque méthode de calcul (PS ou PL) va réaliser 2500 calculs de FFT sur un échantillon de M=1024 valeurs complexes d'entrée. La moyenne des temps d'exécution est prise pour calculer le temps pris pour calculer une FFT en millisecondes (ms/FFT), puis le nombre de FFT réalisées par millisecondes (FFT/ms). Dans l'ordre, ces métriques correspondent aux méthodes PL octets-octets (pas de conversions), flottant-octets, octets-flottant (une conversion) et flottant-flottant (deux conversion), puis la méthode PS (flottant-flottant).

En analysant ces métriques temporelles, nous pouvons alors synthétiser les résultats suivants :

Opération	Méthode PL (µs)	Méthode PS (µs)
Conversion flottant vers octets	81,826	Pas de conversion
Calcul de la FFT sur 1024 valeurs	**97,507**	**146,762**
Conversion octets vers flottant	40,514	Pas de conversion

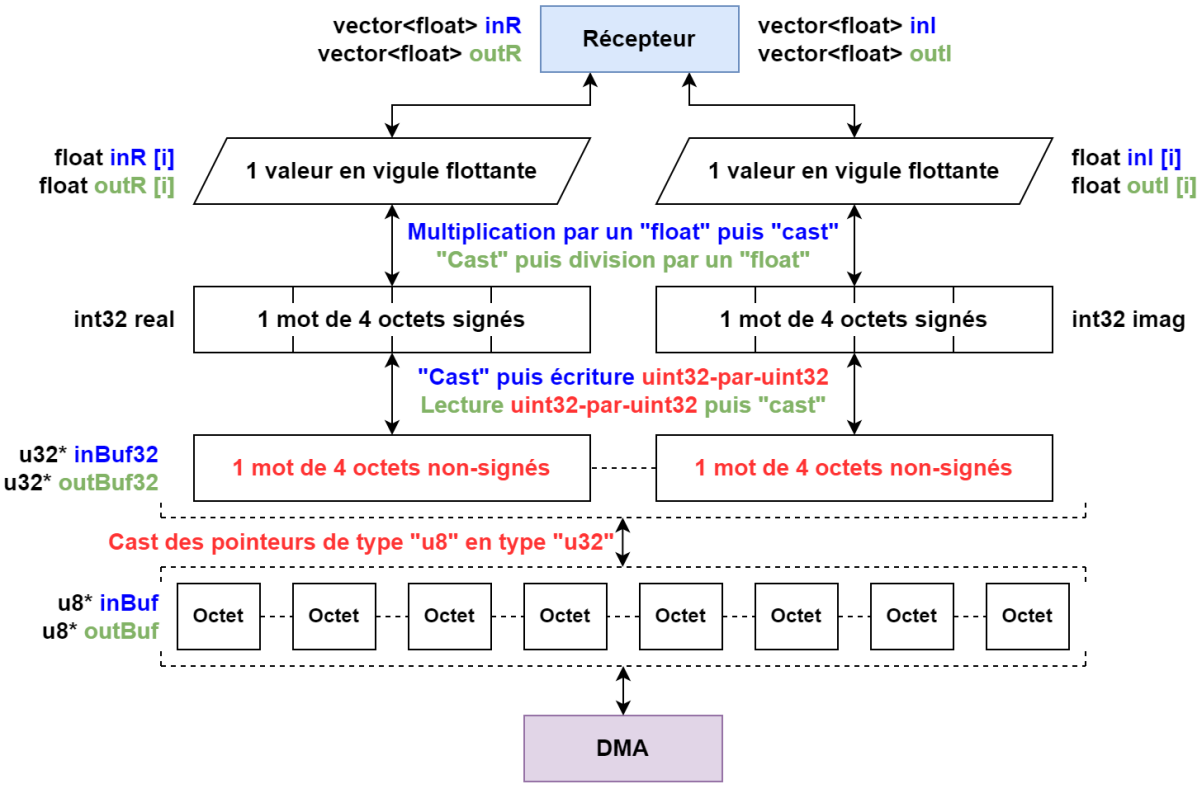
Total (flottant vers flottant)	**220,707**	**146,762**

Immédiatement, on observe deux résultats : l'IP FFT est 34% plus rapide que la classe « fft » sur l'exécution du calcul de FFT, mais la méthode PL réalisant les conversions flottant-octets puis octets-flottant est 50% plus lente que la méthode PS. En d'autres termes, l'IP en soit est rapide, mais le processus de conversion est lent, particulièrement dans le sens flottant-octet (presque aussi long que le calcul lui-même). Si nous devons optimiser notre code, nous devrons commencer par les méthodes fft_PL_Float2Bytes et fft_PL_Bytes2Float.

Dans le futur, il est possible que ces conversions ne soient pas nécessaires. En effet, si le Récepteur venait à utiliser des nombres à virgule fixe (de quelconque format $Q<E, D>$), un simple décalage binaire de la partie décimale et une borne de la partie entière (pour éviter les « overflow ») seront requis en entrée et sortie de l'IP. Ces opérations sont rapides sur des types « u8 » (ou « u32 », voir ci-dessous), ainsi l'IP FFT pourrait être utilisée à son plein potentiel. Pour l'heure cependant, ces conversions avec le type « float » sont nécessaires.

6. Optimisation : conversions 32 bits et SIMD

Une des premières optimisations réalisées fut d'employer des pointeurs de type « u32 » plutôt que de type « u8 ». D'abord employée pour l'IP de dichotomie, cette méthode, suggérée par M. Le Gal, consiste à réduire le nombre de « cast » et d'accès mémoire par 4 en employant un pointeur de type « u32 » (entier de 4 octets) plutôt que 4 éléments issus d'un pointeur de type « u8 » (entier d'un octet) :



L'implémentation de cette technique requiert un simple « cast » des pointeurs *inBuf* et/ou *outBuf* de leur type initial (« u8 ») en un type 4 fois plus long (« u32 »). Puisque la valeur en virgule fixe destinée à (ou provenant de) l'IP est contenue sur 4 octets, ce « cast » des pointeurs permet d'écrire ou lire la valeur en virgule fixe en un seul accès mémoire, et non 4. L'usage de cette optimisation est régi par la définition du signal FFT_PL_32BIT sous la source « FFT_PL.hpp ». La performance des méthodes PL est ainsi améliorée :

```
FFT_PL test OK

(II) Accelerators timing
(II) ------------------------------------------
FFT_PL timing
------------------------------------------ FFT Calculation performance only ------------------------------------------
Bytes-to-Bytes PL processed [2500] FFT for M=[1024]        |      [0.094891] ms/FFT      |      [10.538353] FFT/ms
Float-to-Bytes PL processed [2500] FFT for M=[1024]        |      [0.165618] ms/FFT      |      [6.037983] FFT/ms
Bytes-to-Float PL processed [2500] FFT for M=[1024]        |      [0.118404] ms/FFT      |      [8.445667] FFT/ms
Float-to-Float PL processed [2500] FFT for M=[1024]        |      [0.189610] ms/FFT      |      [5.273987] FFT/ms
Float-to-Float PS processed [2500] FFT for M=[1024]        |      [0.144120] ms/FFT      |      [6.938660] FFT/ms
```

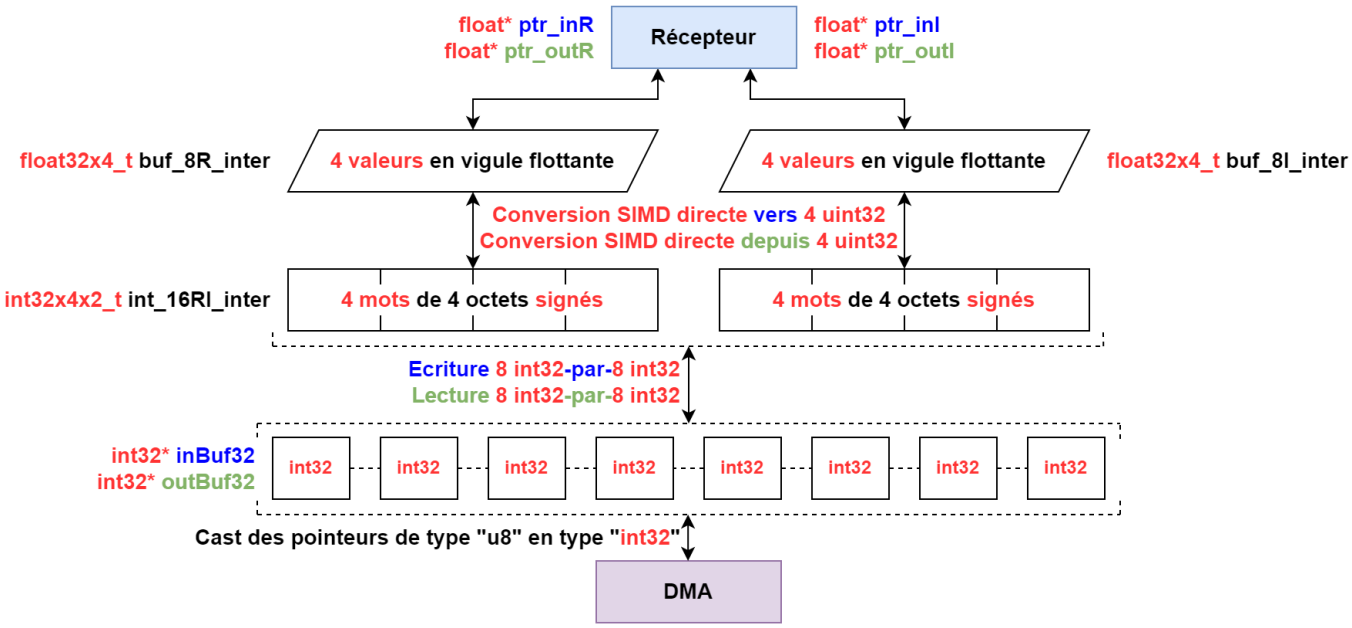
En analysant ces métriques temporelles, nous pouvons alors synthétiser les résultats suivants :

Opération	Méthode PL (µs) Conversions 32 bits	Méthode PL (µs) Conversions 8 bits	Méthode PS (µs)
Conversion flottant vers octets	**70,727**	**81,826**	Pas de conversion
Calcul de la FFT sur 1024 valeurs	94,891	97,507	146,762
Conversion octets vers flottant	**23,513**	**40,514**	Pas de conversion
Total (flottant vers flottant)	**189,610**	**220,707**	**146,762**

Le gain de temps sur la conversion flottant vers octets est faible (juste 13% plus rapide), mais la conversion inverse voit un gain plus important (42% plus rapide). Cela reste insuffisant cependant, puisque le calcul PS reste plus rapide que la méthode PL flottant-flottant. Nous devons aller plus loin.

La plus grande optimisation réalisée durant ce projet fut, une fois n'est pas coutume, suggérée par M. Le Gal : utiliser des instructions SIMD (Single Instruction Multiple Data) pour vectorialiser les opérations de conversion des méthodes fft_PL_Bytes2Float et fft_PL_Float2Bytes. Le processeur de la carte Pynq, un *ARM* Cortex-A9, peut réaliser des opérations SIMD employant jusqu'à 8 valeurs de type « float » ou « int32 » (toutes de 32 bits) par instruction. De plus, ces instructions sont aussi ou plus rapides qu'une instruction standard.

Le fonctionnement de ces instruction, trouvées dans la librairie « arm_neon.h » (NEON est le nom des instructions SIMD de AMD), est assez obtus. Le chargement et stockage en mémoire des données, notamment, est réalisé de manière interlacée (voir code de Tools_FFT_PL), il faut donc constamment garder en tête l'agencement des données dans les variables SIMD temporaires. Cependant, la vectorialisation permet d'utiliser un schéma de conversion de données bien plus direct, et par conséquent, plus rapide :



L'usage des instructions (et l'inclusion de la librairie) SIMD NEON est régi par la définition du signal FFT_PL_SIMD sous la source « FFT_PL.hpp ». À noter que ce signal prend le pas sur le signal FFT_PL_32BIT, car le code SIMD est trop différent du code 32 bits, qui devient inutile dans l'exécution de la méthode. L'usage de ces instructions permet aussi de charger ou stocker les valeurs à virgule fixe 8 par 8, deux fois plus que les pointeurs 32 bits, et les valeurs à virgule flottante 4 par 4, ou quatre fois plus. La performance en PL suit :

```
FFT_PL test OK

(II) Accelerators timing
(II) ------------------------------------------
FFT_PL timing
------------------------------------------ FFT Calculation performance only ------------------------------------------
Bytes-to-Bytes PL processed [2500] FFT for M=[1024]        |      [0.094883] ms/FFT      |      [10.539331] FFT/ms
Float-to-Bytes PL processed [2500] FFT for M=[1024]        |      [0.116107] ms/FFT      |      [8.612748] FFT/ms
Bytes-to-Float PL processed [2500] FFT for M=[1024]        |      [0.117669] ms/FFT      |      [8.498444] FFT/ms
Float-to-Float PL processed [2500] FFT for M=[1024]        |      [0.138957] ms/FFT      |      [7.196461] FFT/ms
Float-to-Float PS processed [2500] FFT for M=[1024]        |      [0.144074] ms/FFT      |      [6.940892] FFT/ms
```

En analysant ces métriques temporelles, nous pouvons alors synthétiser les résultats suivants :

Opération	Méthode PL (µs) Conversions SIMD	Méthode PL (µs) Conversions 8 bits	Méthode PS (µs)
Conversion flottant vers octets	**21,224**	**81,826**	Pas de conversion
Calcul de la FFT sur 1024 valeurs	94,883	97,507	146,762
Conversion octets vers flottant	**22,786**	**40,514**	Pas de conversion
Total (flottant vers flottant)	**144,074**	**220,707**	**146,762**

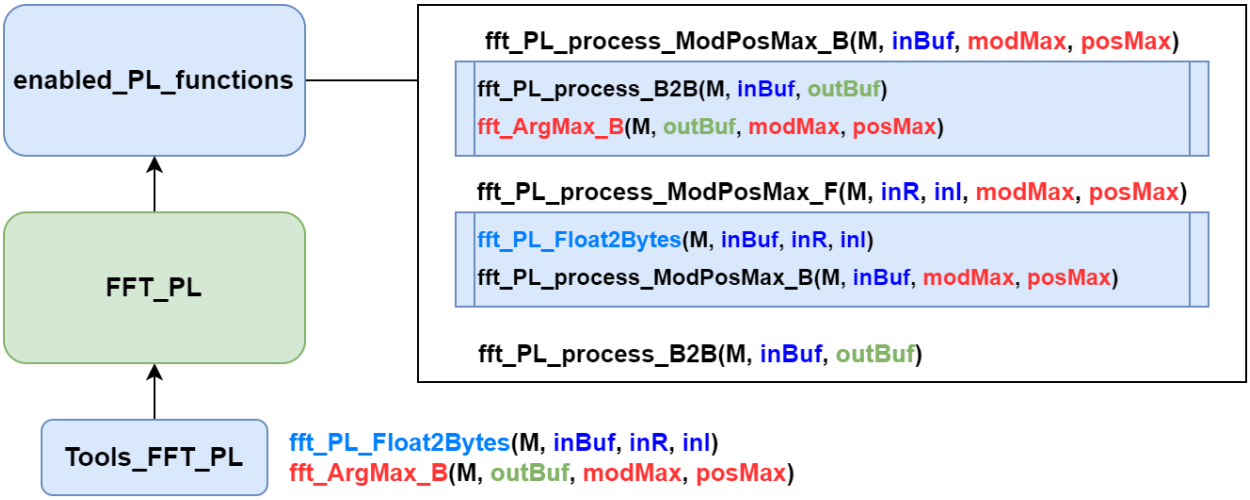
Avec un gain important sur la conversion flottant vers octet (74% plus rapide) et un gain similaire sur la conversion octets vers flottant (44% plus rapide), le calcul PL flottant-flottant est plus rapide que le calcul PS.

Maintenant que nous avons optimisé les méthodes PL, nous pouvons résumer la performance de l'IP FFT et ses méthodes de conversion associées dans le tableau suivant :

Opération (M = 1024) (µs)	Méthode PL Sans conversions	Méthode PL Mode SIMD	Méthode PL Mode 32 bits	Méthode PL Mode 8 bits	Méthode PS
Conversion flottant → octets	0	**21,224**	**70,727**	**81,826**	0
Calcul de la FFT	94,883	94,883	94,891	97,507	146,762
Conversion octets → flottant	0	**22,786**	**23,513**	**40,514**	0
Total (flottant → flottant)	**94,883**	**144,074**	189,610	220,707	**146,762**
Débit équivalent (FFT/ms)	**10,54**	**7,20**	5,27	4,53	**6,94**

Comme observé précédemment, les instructions SIMD permettent aux méthodes PL d'exploiter l'IP FFT sufisamment vite pour dépasser les méthodes PS de la classe « fft ». Dans cette configuration, nous pouvons attester qu'utiliser l'IP FFT va remplir son rôle primaire : décharger le processeur *ARM* d'opérations coûteuses, tout en offrant une performance similaire au calcul purement PS. Cependant, le potentiel réel de l'IP FFT ne peut pas être exploité, à cause des conversions coûteuses mentionnées maintes fois, mais pas que.

L'IP FFT actuelle charge $2 \times M$ valeurs, réalise une FFT sur M valeurs complexes, puis stocke les $2 \times M$ valeurs de sortie en mémoire. La détermination du module maximal (et son indice) est ensuite faite par le processeur, qui parcourt les $2 \times M$ valeurs de sortie. Cela demande plus d'accès mémoire et de conversions de format, qui coûtent toujours plus de temps processeur. Les méthodes fft_PL_process_ModPosMax_F et fft_PL_process_ModPosMax_B agissent comme des surcouches des méthodes PL précédentes :



Les deux méthodes PL appellent la méthode PS fft_ArgMax_B, dont le rôle est de lire le buffer de sortie *outBuf* puis de calculer le module maximal *modMax* et son indice *posMax*. Comme les méthodes de conversion décrites dans Tools_FFT_PL, fft_ArgMax_B a été optimisée successivement, d'abord avec un pointeur 32 bits puis des instructions SIMD. Le même schéma d'optimisation a été suivi par la méthode fft_PL_generator, et sont activables par l'utilisation des signaux FFT_PL_32BIT et FFT_PL_SIMD respectivement.

En utilisant les instructions SIMD, la performance des méthodes PL et de la méthode PS est la suivante :

```
---------------------------------- Maximum modulus and argument calculation performance ----------------------------------
Bytes-input PL calculated [2500] FFT maximums for M=[1024]      ¦        [0.156756] ms/ModMax    ¦        [6.379336] ModMax/ms
Float-input PL calculated [2500] FFT maximums for M=[1024]      ¦        [0.178045] ms/ModMax    ¦        [5.616572] ModMax/ms
Float-input PS calculated [2500] FFT maximums for M=[1024]      ¦        [0.163706] ms/ModMax    ¦        [6.108503] ModMax/ms
```

La méthode PL de calcul utilisant des valeurs à virgule flottante en entrée est plus lente que la méthode PS associée. Cela est dû en partie à l'écriture de la méthode PL fft_ArgMax_B, qui convertit l'ensemble des valeurs d'entrée à virgule fixe en nombres à virgule flottante, tandis qu'aucune conversion n'est nécessaire sur la méthode PS. Mais réaliser ces calculs sur l'IP elle-même permettrait non seulement d'accélérer l'opération, mais surtout de réduire le nombre de valeurs de sortie de $2 \times M$ à seulement 2.

7. Conclusion

En termes de performance pure, l'IP FFT remplit son rôle parfaitement : près de deux fois plus rapide que la méthode associée de la classe « fft », son potentiel d'accélération est certain. Et après optimisation des méthodes de conversion d'entrée (flottant → octets) et de sortie (octets → flottant), elle reste aussi rapide (voire plus rapide) que la méthode PS. En tous cas, le processeur a désormais moins d'opérations à réaliser.

Mais un nombre de limitations viennent diminuer sa performance théorique. D'abord, le besoin de passer entre virgules fixes et flottantes demande un grand nombre d'opérations lourdes. Cela ne sera plus un problème lorsque le Récepteur n'utilisera que des méthodes en virgule fixe, mais pour l'heure, elles occupent plus de 50% du temps d'exécution de la méthode PL flottant-flottant. Enfin, l'IP n'a été testée que pour *SF* 10, alors qu'elle pourrait peut-être supporter des *SF* inférieurs. *SF* 11 et 12 demanderont des modifications de l'IP.

De plus, la méthode de calcul du module maximal et son indice n'est pas encore optimale, et le calcul réalisé en PS reste plus rapide que celui réalisé en PL. Ce problème peut être résolu de deux manières : réécrire la méthode PL fft_ArgMax_B, qui permettrait de prendre avantage de l'arithmétique à virgule fixe (et d'éliminer les conversions flottant → octets), ou transformer l'IP FFT pour que ce calcul soit réalisé dans l'accélérateur matériel, accélérant le calcul mais, surtout, réduisant grandement le nombre de valeurs reçues depuis l'IP.

Néanmoins, l'étape la plus critique à la suite de ce projet a été accomplie : l'IP elle-même a été prise en main, comprise, utilisée telle qu'elle grâce au code hérité de Hugues, puis exploitée plus efficacement en employant nos propres méthodes. Les optimisations réalisées durant cette étude n'auraient pas été possible sans comprendre le format exact des données et les spécificités techniques de l'IP. Maintenant que le code C++ a été établit, des modifications substantielles devraient être possibles sans trop de difficultés.

L'ensemble des codes sources modifiés dans cette partie sont présent dans le répertoire GitHub suivant (autorisation nécessaire, contactez Alexandre Arrivé). Le répertoire AcceleratorPL contient les sources haut-niveau (enabled_PL_functions, enabled_PL_param, etc.), le répertoire AcceleratorApprox avec les sources de l'IP dichotomique, et enfin le répertoire AcceleratorFFT avec les sources de l'IP FFT. D'autres fichiers seront ajoutés en parallèle (usage de deux contrôleurs *DMA*, meilleure méthode fft_ArgMax_B, etc.) si le temps ne manque pas.

Quoi qu'il en soit, la montée en compétences demandée par ce projet a été fulgurante : conception d'accélérateurs matériels en *HLS*, intégration d'une simple IP sur *Vivado*, exploitation en Co-design (PS-PL) sur *Vitis*. Puis étude d'une IP complexe écrite par un inconnu, modification et optimisation des méthodes PL en C++ permettant son exploitation, et analyse des résultats. Sans oublier la découverte du SIMD, les heures passées à comprendre le flot de données entre processeur et IP, les mécaniques de conversion de type, etc. Enrichissant.

X. Conclusion

Au cours de ce projet nous avons été capables de prendre en main un projet complexe, riche, et très fortement intéressant.

Nous avons ainsi pu implémenter en C++ un algorithme de détection d'interférences pour un récepteur LoRa avec succès. En parallèle du développement de cet algorithme, nous avons mis en place toute une stratégie d'implémentation pour vérifier la fiabilité de nos résultats. Cette stratégie étant composée à la fois d'une vérification automatique, et d'une visualisation sur Matlab.

Nous avons également pu découvrir et approfondir nos connaissances en conception conjointe avec la suite d'outil de Xilinx : Vitis, Vitis HLS et Vivado. Nous avons ainsi appris à développer et à implémenter des accélérateurs dans des exercices de conception conjointe.

A la suite de ces exercices, deux accélérateurs ont étés implémentés sur un récepteur LoRa pour la plateforme NanoNAASC.

Le premier accélérateur appelé « approx_PL » a été corrigé afin de fonctionner pour chaque SF. De nombreuses améliorations sur différents aspects ont étés réalisés. Ces améliorations concernent l'écriture en mémoire alignée par le CPU, la réception des données par le FPGA, ainsi que la manière dont ces données sont traitées. Toutes ces améliorations permettent un gain important de performances temporelles.

Le second accélérateur appelé « FFT_PL » a été implémenté et a nécessité beaucoup de travail qui lui ont permis de fonctionner. Les premières versions étaient peu efficaces, et elles étaient mêmes plus lentes que le processeur. Cependant, après une meilleure compréhension du fonctionnement de l'IP, son implémentation en utilisant des écritures en mémoire SIMD a permis à cet accélérateur d'avoir de meilleurs performances que le CPU. De nombreuses modifications futures sont également possibles pour le rendre encore meilleur dans son utilisation par le CPU.

Finalement, nous pouvons dire que ce projet est un succès, puisque nous avons pu mener à bien l'ensemble des tâches qui nous ont étés données.

Comme note finale, nous voulons féliciter Alexandre qui a produit un travail colossal et qui a réussi à atteindre ses objectifs d'implémentation de l'accélérateur FFT_PL avec beaucoup d'investissement.

Nous souhaitons aussi vous remercier pour ce très bon projet et ce très bon encadrement, et nous espérons vous revoir bientôt,

Valentin, Clarisse, Alexandre, Matthieu