Architectures logicielles pour la radio flexible : intégration d'unités de calcul hétérogènes

Pierre-Henri Horrein

▶ **To cite this version:**

Pierre-Henri Horrein. Architectures logicielles pour la radio flexible : intégration d'unités de calcul hétérogènes. Electronique. Université de Grenoble, 2012. Français. NNT : . tel-00668338

HAL Id: tel-00668338

https://theses.hal.science/tel-00668338

Submitted on 6 Jun 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Pierre-Henri Horrein

Thèse dirigée par **Frédéric Pétrot**
et encadrée par **Christine Hennebert**

préparée au sein **du CEA/LETI et du TIMA**
et de **l'École Doctorale Mathématiques, Sciences et Techniques de l'Ingénieur, Informatique**

Architectures logicielles pour la radio flexible : intégration d'unités de calcul hétérogènes

Thèse soutenue publiquement le **10 janvier 2012**,
devant le jury composé de :

M. Olivier Sentieys
Professeur des Universités, ENSSAT, Président
M. Tanguy Risset
Professeur des Universités, INSA Lyon, Rapporteur
M. Christophe Moy
Professeur, Supélec Rennes, Rapporteur
M. Jean-Luc Danger
Directeur d'études, Télécom ParisTech, Examinateur
M. Franck Rousseau
Maître de Conférences, Grenoble INP, Examinateur
M. Frédéric Pétrot
Professeur des Universités, Grenoble INP, Directeur de thèse
Mme Christine Hennebert
Docteur, CEA, Encadrante

Remerciements

La thèse est un travail personnel, mais beaucoup de monde y participe. J'aimerais citer dans ces remerciements toutes les personnes qui ont rendu ce travail possible. Cependant, elles sont nombreuses, et je demande pardon par avance à ceux que je ne pourrai pas citer. J'aimerais tout d'abord remercier mes deux rapporteurs, Tanguy Risset et Christophe Moy d'avoir accepté de s'acquitter de cette tâche consommatrice en temps. Leurs remarques judicieuses sur mon travail avant et pendant la soutenance m'ont forcé à creuser et à élargir mon champ de vision. Je remercie également les membres du jury, Olivier Sentieys, Franck Rousseau, et Jean-Luc Danger d'avoir fait le déplacement jusqu'à Grenoble pour ceux qui n'y étaient pas, et d'avoir pris le temps de s'intéresser à mon travail.

J'ai eu la chance d'être soutenu avant et durant cette thèse par mon directeur de thèse, Frédéric Pétrot. Même si la thèse est un objectif depuis longtemps, il m'a montré ce qu'était la recherche et m'a remotivé (sans le savoir) lors de mon premier stage avec lui en 2007. Je l'ai parfois trop sollicité, mais il a toujours répondu présent, même quand le temps lui manquait. De même, Christine Hennebert m'a encadré au CEA durant cette thèse, et je l'en remercie. Elle a su juguler ma tendance à papillonner, et m'a permis de rester sur des rails plus ou moins droits malgré les difficultés, et ça n'a pas été évident.

J'ai pu m'appuyer sur deux laboratoires durant ma thèse. Du côté du CEA, l'équipe LASP, mon équipe de résidence devenue ANP au cours de ma thèse, m'a chaleureusement accueilli dès mon arrivée. Certaines personnes ont cependant eu plus d'influence que d'autres. Dimitri Kténas, bien qu'extérieur à l'équipe m'a offert la possibilité de travailler sur son simulateur et m'a permis d'avancer dans la direction que j'avais prise à l'époque. Xavier Popon m'a consacré du temps pour le support sur la carte Magnet. Dominique Noguet m'a soutenu dans mes démarches administratives et m'a judicieusement conseillé sur les directions à prendre tout au long de ma présence au CEA. Manuel Pezzin, par ses conseils, ses avis, et sa bonne humeur m'a grandement aidé à surmonter mes moments de petite forme. Florian Pebay-Peyroula m'a accueilli dans son bureau. J'ai beaucoup apprécié nos discussions, qu'elles soient scientifiques ou non, et j'espère pouvoir continuer à travailler avec lui. Pour nos discussions (pas forcément professionnelles) tard le soir, je remercie également Roselin. Finalement, une bonne ambiance au travail est primordiale pour avancer et l'équipe P'tit-Déj a fortement contribué à cette ambiance. Je remercie donc Alexandre, Yanis, Cécile, Mathilde, Jessie, Philippe, et Stéphanie (et Manuel et Florian, une fois de plus !), ainsi que Claire, arrivée après le P'tit-Déj. Ces remerciements dépassent bien sûr le cadre strictement professionnel.

Même si dans les faits, je n'étais pas réellement dans le laboratoire TIMA, je me suis toujours senti bienvenu dans l'équipe SLS, pour les quelques réunions de groupe auxquelles j'ai participé, ou juste pour un thé. Les personnes du groupe que j'aimerais remercier individuellement, en plus de mon directeur de thèse, sont plus des amis que des collègues. Donc, au TIMA et pas seulement, je remercie tout particulièrement Olivier Muller. Son soutien scientifique, son oreille attentive, ses conseils avisés (sans oublier son chat !) m'ont été d'un grand secours. Merci également de m'avoir donné l'opportunité de participer au projet C. Cette expérience, bien qu'épuisante, aura été bénéfique. Nicolas Fournel et Damien Hedde m'ont supporté à la maison durant ces dernières années (et il faut du courage !). Nos soirées (nuits/week-ends) Vorbis, nos discussions vidéoludiques, nos randonnées, et finalement les conseils de Nicolas et le parallélisme temporel de la thèse de Damien et de la mienne ont été un soutien bienvenu.

Finalement, même sans aucun lien avec le travail, certaines personnes ont participé au bon

déroulement de cette thèse. Pour cela, je remercie Catherine, qui m'a supporté et soutenu durant toute cette dernière année de thèse, malgré les difficultés de sa propre thèse. Céline, pour son soutien inconditionnel et les épreuves traversées ensemble, sera toujours importante. Merci finalement à Guillaume et Juliette qui restent proches même s'ils sont loins, et à Sylvain, moins loin et aussi proche.

Et bien entendu, pour m'avoir permis d'en arriver là et m'avoir soutenu et entouré quels que soient mes choix, ne les oublions pas : je n'aurais pas pu y arriver sans le soutien permanent de mes parents, de mes frères, et de ma famille. Je n'ai malheureusement pas la place de tous les citer, mais je tiens à les remercier chaleureusement.

Table des matières

1 Introduction **1**

2 Problématique **5**

2.1 Contexte de la thèse . . . 6
2.1.1 Modèle réseau . . . 6
2.1.2 Chaîne de communication . . . 6
2.2 Réseaux flexibles . . . 7
2.2.1 Radio classique . . . 7
2.2.2 Radio reconfigurable . . . 8
2.2.2.1 Vision générale . . . 8
2.2.2.2 Software Radio . . . 8
2.2.2.3 Software-Defined Radio . . . 9
2.2.3 Radio flexible . . . 10
2.2.4 Radio cognitive . . . 10
2.2.4.1 Terminal de radio cognitive . . . 10
2.2.4.2 Radio opportuniste . . . 11
2.2.5 Interactions radio flexible/radio cognitive . . . 11
2.2.5.1 Différences de terminologie . . . 11
2.2.5.2 Architecture d'un terminal radio . . . 11
2.2.6 Conclusion . . . 12
2.3 Radio flexible : de multiples possibilités . . . 12
2.3.1 Scénarios de reconfiguration . . . 12
2.3.1.1 Adaptation . . . 12
2.3.1.2 Évolution des normes . . . 13
2.3.1.3 Multimode . . . 13
2.3.2 De multiples cibles de reconfiguration . . . 13
2.3.2.1 Cibles logicielles . . . 13
2.3.2.2 Matériel reconfigurable . . . 14
2.3.3 Conclusion sur la radio flexible . . . 15
2.4 Choix de l'implantation : évolution . . . 15
2.4.1 Performance de la radio logicielle . . . 15
2.4.2 Essor du GPGPU . . . 16
2.4.3 Problèmes du GPGPU pour la radio logicielle . . . 16
2.5 Environnement de reconfiguration unique . . . 16
2.5.1 Objectifs . . . 16
2.5.2 Gestion de la reconfigurabilité . . . 17
2.5.3 Intégration de la radio flexible . . . 18
2.5.4 Gestion de la reconfiguration . . . 18
2.5.5 Conclusion : besoins d'un environnement flexible . . . 19
2.6 Conclusion générale . . . 19

3 État de l'art **21**
3.1 Représentation d'une application radio . . . 22
3.1.1 Reconfiguration . . . 22
3.1.1.1 Définition générale . . . 22
3.1.1.2 Cas de la radio logicielle . . . 22
3.1.2 Représentation des applications . . . 22
3.2 Plateformes de radio logicielle . . . 23
3.2.1 Solutions à base de processeurs . . . 23
3.2.2 Intégration des GPU . . . 24
3.2.3 Utilisation de FPGA . . . 24
3.2.4 Solutions hybrides . . . 25
3.2.5 Matériel reconfigurable et adaptabilité . . . 25
3.2.5.1 Paramétrisation et opérateurs communs . . . 25
3.2.5.2 ASIC et SOC . . . 25
3.2.6 Conclusion sur les plateformes . . . 26
3.3 Environnements . . . 26
3.3.1 Objectifs . . . 26
3.3.2 Exécution hétérogène . . . 26
3.3.2.1 Accélérateurs matériels . . . 27
3.3.2.2 Intégration des FPGA . . . 27
3.3.2.3 Exécution hétérogène GPP/DSP . . . 27
3.3.3 Environnements complets . . . 28
3.3.3.1 SCA . . . 28
3.3.3.2 RMA . . . 29
3.3.3.3 GNURadio . . . 29
3.3.3.4 P-HAL/Aloe . . . 30
3.3.3.5 MVR : Machine Virtuelle Radio . . . 31
3.3.3.6 Surfer . . . 31
3.3.4 Conclusion . . . 32
3.4 Conclusion du chapitre . . . 32

4 Intégration du GPU dans la radio logicielle **35**
4.1 Introduction : le GPGPU . . . 36
4.1.1 Origine . . . 36
4.1.2 Principe . . . 36
4.1.2.1 Architecture d'une plateforme OpenCL . . . 36
4.1.2.2 Format d'une application . . . 37
4.1.3 Contraintes et objectif . . . 38
4.2 Utilisation optimisée . . . 39
4.2.1 Opérations basiques . . . 39
4.2.2 Opérations complexes : première approche . . . 40
4.2.3 Proposition : maximisation de l'utilisation . . . 41
4.2.3.1 Parallélisation à grain fin . . . 41
4.2.3.2 Parallélisation à gros grain . . . 42
4.2.3.3 Comparaison . . . 42
4.3 Ordonnancement . . . 43
4.3.1 Efficacité . . . 43
4.3.2 Latence . . . 43
4.3.3 Mémoire . . . 45
4.4 Intégration distribuée . . . 45

4.4.1 Présentation 45
4.4.2 Communications 45
4.4.2.1 Intégration transparente 46
4.4.2.2 Buffer OpenCL 46
4.5 Intégration centralisée 47
4.5.1 Présentation 47
4.5.2 Communications 48
4.5.3 En pratique 48
4.6 Résultats 49
4.6.1 Synthèse des contributions 49
4.6.2 Opérations implantées 49
4.6.2.1 FFT 50
4.6.2.2 *Demapping* 50
4.6.2.3 IIR 51
4.6.3 Protocole d'expérimentation 52
4.6.4 Résultats des opérations unitaires 53
4.6.4.1 Influence du seuil 53
4.6.4.2 Influence de la taille des vecteurs 55
4.6.5 Résultats pour des applications multitâches 56
4.7 Perspectives et conclusion 57
4.7.1 Discussion : portabilité vers l'embarqué 57
4.7.2 Conclusion 58

5 Définition d'un environnement pour la radio flexible 61
5.1 Introduction 62
5.1.1 Contexte 62
5.1.2 Objectifs 62
5.1.2.1 Support des unités d'exécution 63
5.1.2.2 Application, adaptabilité et reconfigurabilité 63
5.2 Architecture générale 64
5.2.1 Présentation générale de l'environnement 64
5.2.2 R-HAL et traducteur 65
5.2.2.1 Représentation et contrôle de la plateforme 65
5.2.2.2 Traduction 65
5.2.2.3 Gestion des applications 66
5.2.3 Couche protocolaire 66
5.3 Gestion des applications 67
5.3.1 Présentation : flot de développement d'une application avec FRK 67
5.3.2 Représentation d'une application 68
5.3.2.1 *Waveform* 68
5.3.2.2 *Configuration Instance* 68
5.3.3 Traduction et implantation de l'application 68
5.3.3.1 Opérations 69
5.3.3.2 Transferts de données 70
5.3.3.3 Chargement d'une application 71
5.3.4 Gestion de l'exécution 71
5.3.4.1 Ordonnancement 71
5.3.4.2 Adaptabilité : modification dynamique de la CI 72
5.4 Implantation des cibles 73
5.4.1 TaME et extensions 73

5.4.1.1 Structure . . . 73
5.4.1.2 Interface : API principale . . . 73
5.4.2 Cible logicielle . . . 74
5.4.2.1 Définition . . . 74
5.4.2.2 Transferts de données . . . 74
5.4.2.3 Ordonnancement . . . 74
5.5 Intégration des coprocesseurs . . . 75
5.5.1 Définition et postulat . . . 75
5.5.2 Représentations . . . 76
5.5.2.1 Coprocesseur . . . 76
5.5.2.2 Opérations . . . 76
5.5.3 Fonctionnement de la cible coprocesseur . . . 77
5.5.3.1 Structure . . . 77
5.5.3.2 *Config Switch* . . . 77
5.5.3.3 Ordonnancement . . . 78
5.5.3.4 Transferts de données . . . 78
5.5.3.5 Exemple : le codeur . . . 79
5.6 FRK en pratique . . . 80
5.6.1 État de FRK . . . 80
5.6.2 Implantation . . . 81
5.6.2.1 Linux . . . 81
5.6.2.2 RTEMS . . . 81
5.6.3 Exemple du codeur convolutif . . . 81
5.6.3.1 Présentation du programme . . . 81
5.6.3.2 Exécution logicielle . . . 82
5.6.3.3 Exécution matérielle : une seule application . . . 83
5.6.3.4 Exécution matérielle : deux applications . . . 84
5.6.4 Exemple complet : IEEE 802.11 . . . 87
5.6.4.1 Présentation du programme et de la plateforme . . . 87
5.6.4.2 Fonctionnement BPSK . . . 88
5.6.4.3 Demande d'adaptation . . . 88
5.6.5 Quelques chiffres . . . 89
5.7 Conclusion . . . 89

6 Conclusion 91

A Résultats complets pour le GPU 95

B Parallélisation d'un filtre IIR 99
B.1 Principe . . . 99
B.2 Démonstration . . . 99
B.3 Conclusion . . . 102

C Présentation générale de l'interface de FRK 103
C.1 OPM et communications . . . 103
C.1.1 Structures . . . 103
C.1.1.1 Opération . . . 103
C.1.1.2 Waveform . . . 104
C.1.1.3 CI . . . 104
C.1.2 Fonctions . . . 105
C.1.2.1 Opérations . . . 105

C.1.2.2 Création de la waveform . . . 105
C.1.2.3 Traduction . . . 105
C.2 Interface R-HAL . . . 105
C.2.1 Structures . . . 106
C.2.1.1 Transferts de données . . . 106
C.2.1.2 Opérations . . . 106
C.2.2 Fonctions . . . 107
C.3 TaME . . . 107
C.3.1 Structures . . . 108
C.3.1.1 Structure d'un TaME . . . 108
C.3.1.2 Opérations . . . 108
C.3.1.3 Transferts de données . . . 109
C.3.2 Fonctions . . . 110
C.3.2.1 CI . . . 110
C.3.2.2 Opérations . . . 110
C.3.2.3 Transferts de données . . . 110
C.4 *Monitoring* . . . 111
C.5 Résumé et conclusion . . . 112

D Intégration de matériel dédié dans FRK 113
D.1 Utilisation de MAGNET . . . 113
D.1.1 Plateforme MAGNET . . . 113
D.1.2 Représentation . . . 113
D.2 Conclusion . . . 116

Publications 117

Bibliographie 119

Table des figures

2.1 Modèle OSI . . . 6
2.2 Exemple de chaîne de communication type pour un réseau sans fil . . . 7
2.3 Représentation d'un récepteur superhétérodyne . . . 8
2.4 Représentation d'un récepteur idéal de radio logicielle . . . 9
2.5 Représentation d'un récepteur réalisable de SDR . . . 10
2.6 Cycle simplifié d'un terminal intelligent [MM99] . . . 10
2.7 Interactions entre radio flexible et radio cognitive . . . 12
2.8 Représentation du rapport flexibilité/performance de différentes solutions d'implantation . . . 14
2.9 Architecture de plateforme flexible générique . . . 17

3.1 Représentation et reconfiguration hiérarchique [DMLP05] . . . 23
3.2 Architecture de l'environnement SCA . . . 28
3.3 Architecture générale de GNURadio . . . 30
3.4 Architecture générale de P-HAL/Aloe . . . 31

4.1 Représentation d'une plateforme OpenCL . . . 37
4.2 Architecture d'une application OpenCL . . . 38
4.3 Opération de radio logicielle . . . 39
4.4 Opération de radio logicielle sur GPU : approche classique . . . 40
4.5 Opération de radio logicielle sur GPU : approche proposée . . . 42
4.6 Débit échantillons en MS/s pour des FFT de 128 points pour différents seuils . . 44
4.7 Implantation distribuée . . . 46
4.8 Utilisation de *buffers* spécifiques pour les GPU . . . 47
4.9 Implantation centralisée . . . 48
4.10 Protocole d'expérimentation . . . 53
4.11 Débit échantillons en MS/s pour la FFT pour différents seuils, à taille de vecteur fixée . . . 54
4.12 Débit échantillons en MS/s pour le *demapping* pour différents seuils, à taille de vecteur fixée . . . 54
4.13 Débit échantillons en MS/s pour l'IIR pour différents seuils, à taille de vecteur fixée 55
4.14 Débit échantillons en MS/s en fonction de N, pour un seuil optimal calculé expérimentalement . . . 56
4.15 Résultats pour une séquence d'opérations, pour un seuil optimal calculé expérimentalement . . . 57

5.1 Architecture de plateforme flexible générique . . . 62
5.2 Intégration de *Flexible Radio Kernel* (FRK) dans une architecture logicielle classique 64
5.3 Intégration de FRK dans une architecture logicielle classique . . . 65
5.4 Architecture de la PL . . . 66
5.5 Étapes de chargement d'une application FRK . . . 67
5.6 Traduction d'une opération en utilisant l'OPM . . . 69
5.7 Architecture général d'un contrôleur de cibles . . . 73
5.8 Représentation d'une tâche pour la cible logicielle . . . 75
5.9 Codeur convolutif matériel paramétrable . . . 76

5.10 Instance pour le codeur convolutif 77
5.11 Implantation de la cible pour le codeur 77
5.12 Automate de sélection d'une configuration pour le TaME coprocesseur 79
5.13 Plateforme MagNET 80
5.14 Application traduite pour une cible logicielle 83
5.15 FRK avec deux codeurs, cibles mixtes 84
5.16 FRK avec deux codeurs en matériel 85
5.17 Représentation de l'état du système 86
5.18 Application IEEE 802.11 simplifiée utilisée 87
5.19 Plateforme matérielle pour l'application IEEE 802.11 87

A.1 Résultats complets pour la FFT 95
A.2 Résultats complets pour le *demapping* 96
A.3 Résultats complets pour la IIR 97

C.1 Diagramme de classe des structures de FRK 112

Liste des acronymes

API *Application Programming Interface*

ASIC *Application Specific Integrated Circuit*

ASIP *Application Specific Integrated Processor*

BSP *Board Support Package*

CI *Configuration Instance*

CORBA *Common Object Request Broker Architecture*

CPU *Central Processing Unit*

DSP *Digital Signal Processor*

DVB *Digital Video Broadcasting*

FFT *Fast Fourier Transform*

FIFO *First In First Out*

FIR *Finite Impulse Response*

FRK *Flexible Radio Kernel*

FPGA *Field-Programmable Gate Array*

GPGPU *General Purpose computing on Graphics Processing Unit*

GPP *General Purpose Processor*

GPU *Graphics Processing Unit*

GSM *Global System for Mobile communications*

HAL *Harware Abstraction Layer*

KPN *Kahn Process Network*

LDPC *Low Density Parity Check*

MAC *Medium Access Control*

NoC *Network on Chip*

OFDM *Orthogonal Frequency Division Multiplexing*

OPM *Operation to Platform Mapping*

OSI *Open Systems Interconnection*

OpenCL *Open Computing Language*

PHY couche PHYsique

PL *Protocol Layer*

POSIX *Portable Operating System Interface for Unix*

QAM *Quadrature Amplitude Modulation*

QPSK *Quadrature Phase Shift Keying*

R-HAL *Radio Hardware Abstraction Layer*

RTEMS *Real-Time Executive for Multiprocessor Systems*

RVM *Radio Virtual Machine*

SCA *Software Communications Architecture*

SIMD *Single Instruction Multiple Data*

TaME *TArget Management Element*

UMTS *Universal Mobile Telecommunication System*

WCDMA *Wideband Code Division Multiple Access*

Chapitre 1

Introduction

L'INFORMATIQUE d'aujourd'hui est connectée à tous les niveaux. Les applications sont de plus en plus externalisées, et il devient difficile de trouver des programmes informatiques n'utilisant pas les réseaux.

Allant de paire avec les applications, la conception même des plateformes matérielles intègre les réseaux. Que ce soit les serveurs, les stations de travail, les ordinateurs portables, les téléphones, . . . La plupart des systèmes utilisent aujourd'hui plusieurs méthodes d'accès aux réseaux. L'évolution des téléphones portables est représentative de cette connectivité accrue. Ils permettent de se connecter au réseau téléphonique [1], peuvent utiliser le Bluetooth pour se connecter à des périphériques ou à un ordinateur, et ont également souvent accès à un réseau WiFi. Tous ces réseaux sont des réseaux sans fil, qui utilisent les ondes électromagnétiques comme support. Ils évoluent régulièrement, de nouveaux voient le jour, certains ne s'utilisent plus.

Contexte : la radio logicielle

Intégrer une norme réseau dans une plateforme matérielle requiert l'intégration de plusieurs éléments :

- une antenne adaptée à l'onde utilisée pour la norme ;
- un système de modulation et de démodulation, qui permet de moduler le signal en bande de base sur la porteuse à l'émission, et d'effectuer l'opération inverse pour la réception. Ce système est généralement en partie analogique et en partie numérique ;
- une chaine d'opérations, appelée couche PHYsique (PHY), qui transforme à l'émission la suite de bits à envoyer en un signal en bande de base selon des caractéristiques propres à la norme. A la réception, une chaine d'opérations duale effectue le traitement inverse afin de restituer la séquence de bits émise ;
- un protocole, appelé couche *Medium Access Control* (MAC), qui définit quelle suite de bits sera envoyée à quel moment sur quel canal. La couche MAC reçoit les données du réseau et les transfère à la couche physique.

Les *Application Specific Integrated Circuit* (ASIC) sont communément utilisés pour supporter une norme de communication dans un terminal radio. Un terminal radio contient autant de périphériques matériels qu'il y a de réseaux auxquels il est susceptible de se connecter. L'utilisation de périphériques matériels permet actuellement d'atteindre les performances requises en termes de taille, de consommation et de débit.

Alors que la multiplication des normes supportées par un simple système génère une augmentation de la surface de silicium requis par les ASIC, que de nouvelles normes arrivent sur le marché régulièrement, que toutes ces normes sont sujettes à de nombreuses évolutions et mises à jour, la notion de flexibilité s'invite dans la réflexion concernant la conception des futurs systèmes communicants. On ne veut pas seulement implanter des réseaux de manière performante, et en limitant la consommation d'énergie. On veut pouvoir faire évoluer ces réseaux, et les implanter de manière intelligente. Cette flexibilité peut s'appuyer sur différents types de circuits :

1. ce qui reste tout de même leur fonction première

logiciel sur processeurs (généralistes ou non), matériel paramétrable, *Field-Programmable Gate Array* (FPGA), ...

La notion de flexibilité recouvre plusieurs aspects. Elle permet de gérer la coexistence, voire la coopération entre unités de calcul hétérogènes. Mais, au-delà de cette possibilité offerte aux futurs systèmes, elle permet aussi d'offrir aux systèmes une forme d'intelligence dans leurs capacités d'adaptation à leur environnement. Ces capacités d'adaptation peuvent servir pour différents objectifs :

- changer les paramètres intrinsèques d'un codeur pour s'adapter aux défaillances du canal et réduire les erreurs ;
- changer de norme en fonction de la distance à l'émetteur, du débit requis ;
- éteindre des éléments qui ne servent pas et les ré-allumer dès que nécessaire ;
- utiliser une puce unique pour exécuter différentes normes, potentiellement en concurrence (multimode).

Cette liste n'est pas exhaustive.

Si l'utilisateur peut souhaiter mettre à jour certaines caractéristiques de son système en fonction de ses besoins, il ne veut pas que sa communication soit coupée parce que le codeur change de paramètres ou que le système a détecté qu'il peut communiquer avec une norme plus économe en énergie. La capacité du système à s'adapter doit donc être intégrée tout en conservant la qualité du service offerte à l'utilisateur. La notion de flexibilité prend l'allure d'un défi pour les systèmes à venir.

Problématique générale et contributions

La flexibilité ultime en informatique reste le logiciel. Cependant, la radio logicielle reste expérimentale, utilisable uniquement avec des processeurs puissants, des plateformes dédiées, ou des normes requérant peu de calculs. On peut également se demander si c'est vraiment un objectif à atteindre. La recherche ne s'effectue plus réellement au niveau de la radio logicielle mais du développement des processeurs, puisqu'on peut difficilement réduire le nombre minimal d'opérations requises pour une norme. En attendant l'émergence de processeurs suffisament puissants qui n'arriveront peut être pas, d'autres solutions peuvent être utilisées qui permettent déjà une certaine flexibilité. Elles se basent sur l'utilisation d'unités de calcul hétérogènes. On peut par exemple envisager l'utilisation de processeurs spécialisés, que ce soit des *Application Specific Integrated Processor* (ASIP) ou des *Digital Signal Processor* (DSP). Nous étudions dans cette thèse une unité assez récente pour ce type de calculs, le *Graphics Processing Unit* (GPU). Ces unités proposent une puissance de calcul énorme (de l'ordre du TFlop/s pour les plus puissants !), mais sont basées sur une architecture compliquée à exploiter. Nous verrons comment on peut en bénéficier.

Mais des unités matérielles sont également utilisables. Ces unités sont conçues pour réaliser une opération donnée, mais sont paramétrables pour pouvoir modifier leur fonctionnement. Certains coprocesseurs *Fast Fourier Transform* (FFT), par exemple, peuvent calculer des transformées pour différents nombres de points. On peut aussi définir des encodeurs ou des décodeurs canal programmables. De plus en plus d'architectures matérielles voient le jour qui permettent une certaine flexiblité. Ces architectures sont hétérogènes, c'est-à-dire qu'elles utilisent des unités différentes. Elles ne permettent pas de tout faire, mais sont tout de mêmes capables de balayer des ensembles assez large de possibilités, en utilisant le plus possible les accélérateurs, et en compensant en logiciel les éléments non réalisables en matériel. Le problème des plateformes hétérogènes résident dans la difficulté à les utiliser. Une plateforme logicielle est dans un sens simple à programmer, même si le système qui permet de la contrôler est complexe. On fournit du code, et quelle que soit la plateforme qui l'exécutera, on pourra conserver le même code. Les possibilités de plateformes hétérogènes sont beaucoup plus nombreuses, parce qu'on peut mettre les accélérateurs que l'on veut, pour les opérations que l'on veut. Il y a une infinité de possibilités. Nous nous intéressons à

ces plateformes hétérogènes dans cette thèse, en proposant un environnement capable d'abstraire n'importe quelle unité, et en proposant une méthode originale de gestion des unités matérielles paramétrables.

Plan de la thèse

Cette thèse est ordonnée en 6 chapitres. Le premier chapitre est cette introduction.

Le chapitre 2 présente la problématique de la radio flexible et de cette thèse en général et les points particuliers sur lesquels portent nos travaux.

Le chapitre 3 est dédié à l'état de l'art. Nous présentons plusieurs travaux réalisés dans le domaine de la radio flexible, qui ont alimenté notre réflexion. Nous nous attardons sur les plateformes radio hétérogènes étudiées par d'autres équipes de recherche.

Le chapitre 4 est consacré à l'utilisation du GPGPU pour la radio flexible. La conception d'une application radio à base de GPU et son intégration dans une chaine de communication logicielle sont étudiées théoriquement. Des expérimentations sont menées et les résultats sont présentés et analysés.

Le chapitre 5 détaille un nouvel environnement de radio flexible réalisé pendant cette thèse. L'environnement et ses enjeux sont présentés avant d'aborder des aspects précis comme la gestion des unités d'exécution hétérogènes, l'intégration d'unités matérielles ou l'impact du contrôle sur les applications. Des cas pratiques d'utilisation de l'environnement sont ensuite proposés.

Finalement, les conclusions générales de cette thèse sont données dans le dernier chapitre. Celui-ci s'ouvre sur des perspectives de compléments, d'améliorations ou d'évolution des travaux effectués, et du domaine en général.

Quatre chapitres d'annexe sont également joints. L'annexe A contient les résultats numériques complets pour l'utilisation du GPU dans la radio flexible.

On donne la démonstration dans l'annexe B de la correction d'une implantation parallèle d'un filtre récursif.

L'annexe C présente de manière plus détaillée l'interface de l'environnement défini dans le chapitre 5.

Finalement, l'annexe D donne des détails d'implantation de FRK pour l'utilisation avec du matériel dédié.

Chapitre 2

Problématique

DANS ce chapitre, nous abordons différents problèmes liés à la radio flexible. Comme nous avons pu le voir dans l'introduction, la notion de flexibilité est à prendre en compte lors de la conception d'un terminal radio. Cette flexibilité n'est cependant pas exempte d'inconvénients, en particulier au niveau de l'intégration. Nous donnons ici une définition plus précise de la radio flexible, et nous discutons ensuite les axes adressés dans ces travaux de thèse.

2.1 Contexte de la thèse **6**
- 2.1.1 Modèle réseau 6
- 2.1.2 Chaîne de communication 6

2.2 Réseaux flexibles **7**
- 2.2.1 Radio classique 7
- 2.2.2 Radio reconfigurable 8
- 2.2.3 Radio flexible 10
- 2.2.4 Radio cognitive 10
- 2.2.5 Interactions radio flexible/radio cognitive 11
- 2.2.6 Conclusion 12

2.3 Radio flexible : de multiples possibilités **12**
- 2.3.1 Scénarios de reconfiguration 12
- 2.3.2 De multiples cibles de reconfiguration 13
- 2.3.3 Conclusion sur la radio flexible 15

2.4 Choix de l'implantation : évolution **15**
- 2.4.1 Performance de la radio logicielle 15
- 2.4.2 Essor du GPGPU 16
- 2.4.3 Problèmes du GPGPU pour la radio logicielle 16

2.5 Environnement de reconfiguration unique **16**
- 2.5.1 Objectifs 16
- 2.5.2 Gestion de la reconfigurabilité 17
- 2.5.3 Intégration de la radio flexible 18
- 2.5.4 Gestion de la reconfiguration 18
- 2.5.5 Conclusion : besoins d'un environnement flexible 19

2.6 Conclusion générale **19**

2.1 Contexte de la thèse : les réseaux sans fil

Les travaux réalisés durant cette thèse se situent à la frontière entre les réseaux et l'architecture des systèmes. Afin de pouvoir présenter la problématique que nous adressons, il convient de définir correctement l'architecture générale d'un terminal radio.

2.1.1 Modèle réseau

Tout d'abord, les normes réseaux actuelles sont divisées selon un modèle en couche, le modèle *Open Systems Interconnection* (OSI).

Le modèle OSI, présenté dans la figure 2.1, est un modèle basé sur 7 couches distinctes, chacune des couches étant spécifiée pour remplir un objectif bien précis. Dans un terminal réseau, chacun de ces niveaux est capable de dialoguer avec son équivalent dans un autre terminal. Un niveau donné reçoit des requêtes de la couche directement au dessus, et émet des requêtes vers la couche directement en dessous.

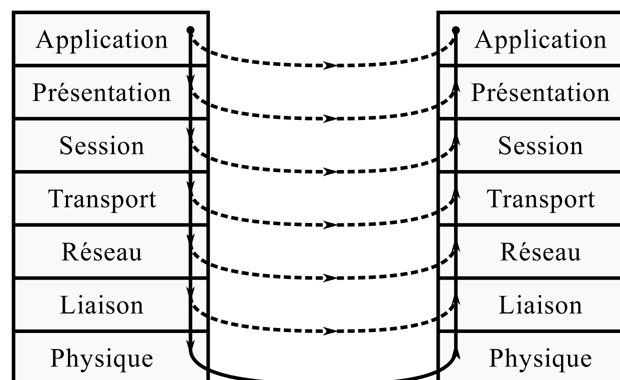


FIGURE 2.1 – Modèle OSI

On s'intéresse principalement aux deux premières couches du réseau, la couche physique (appelée PHY), et la couche de liaison de données (*data link layer*). La première couche est en contact direct avec le *medium* utilisé pour la transmission des données. Dans le cadre de la radio, ce *medium* est le domaine des ondes électromagnétiques.

La seconde couche est conçue pour permettre le transfert entre deux terminaux du réseau. Elle se divise en deux sous-couches, la couche *Medium Access Control* (MAC), qui permet de gérer effectivement l'accès au milieu, surtout dans le cas d'accès multiples, et la couche *Logical Link Controller* (LLC), qui gère normalement le contrôle d'erreur en plus d'autres attributions. On regroupera par abus de langage sous la même appellation "MAC" l'ensemble des éléments de la couche 2.

2.1.2 Chaîne de communication

La partie physique d'un réseau est généralement représentée comme une chaîne de communication, permettant de transposer l'information du domaine numérique au domaine analogique en utilisant des techniques de modulation. Cette chaîne de communication correspond à ce que nous appellerons dans ce mémoire une application radio. Une chaîne type extrêmement simplifiée est présentée dans la figure 2.2.

La chaîne de communication est donc idéalement représentée sous la forme d'un graphe. Chaque nœud représente une tâche et les arêtes qui relient les nœuds représentent les communications (données) entre les tâches. Ce graphe peut-être assimilé à un *pipeline*. On peut utiliser

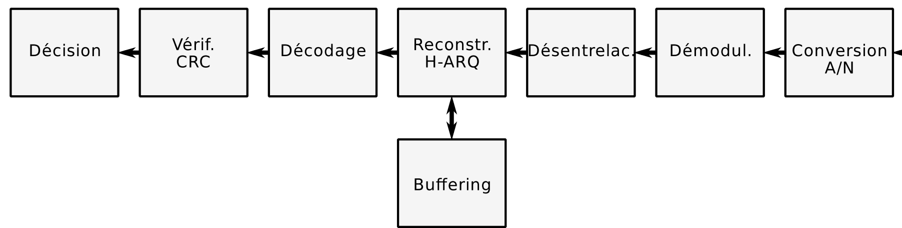


FIGURE 2.2 – Exemple de chaîne de communication type pour un réseau sans fil

le modèle de programmation *Synchronous Data Flow* pour représenter cette application, les ratios entre les entrées consommées et les sorties produites pour chaque tâche étant connus.

2.2 Vers des réseaux de plus en plus flexible

La flexibilité devient une notion incontournable des réseaux sans fil [Tut02]. Le terme de flexibilité est utilisé, dans ce contexte, pour caractériser un élément du réseau qui n'est pas figé dans un mode de fonctionnement. Cet élément peut être le réseau lui-même, ou bien un terminal réseau, ou encore un élément de ce terminal. Depuis la formalisation de la radio logicielle en 1992 [Mit92], la notion de flexibilité dans la radio a pris de l'ampleur et est maintenant devenue un des principaux domaines de recherche associés aux liaisons sans fil.

Cependant, cette notion de flexibilité est floue. Une multitude de termes se rapportent plus ou moins à la radio flexible, il est par conséquent difficile d'en donner une définition précise. Nous recensons dans cette partie les termes les plus utilisés, afin de clarifier la réflexion autour des travaux réalisés dans le cadre de cette thèse. Nous employons la terminologie utilisée dans [DZD+05].

2.2.1 Radio classique

On définit un terminal radio comme un appareil capable d'émettre et de recevoir des données modulées sur des ondes électromagnétiques du domaine radio-fréquentiel (RF). On définit une chaîne de communication radio comme la séquence d'actions à effectuer pour transmettre ou recevoir les données. La chaîne de transmission prend donc en entrée un flux de données à transmettre, et fournit en sortie une onde électromagnétique. La chaîne de réception prend en entrée l'onde électromagnétique, et fournit en sortie un flux de données, qui correspond aux données transmises par l'émetteur.

Les domaines fréquentiels et d'implantation pour un récepteur superhétérodyne usuel sont présentés en figure 2.3. Le domaine RF est la sortie directe de l'antenne. Le récepteur effectue couramment un filtrage et une amplification, qui sont plus simplement réalisés en analogique. Les récepteurs superhétérodynes utilisent une fréquence intermédiaire (FI), qui s'obtient en mélangeant le signal RF à un signal issu d'un oscillateur local. Ce signal intermédiaire est également filtré, puis démodulé afin d'obtenir le signal en bande de base (BB). Ces opérations sont couramment réalisées en analogique également. Le signal bande de base est ensuite échantillonné, afin de réaliser les opérations suivantes en numériques (synchronisation, demapping, désentrelacement, correction d'erreur, . . .).

Dans une implantation traditionnelle d'une chaîne de communication, tous les éléments de la chaîne sont implantés à l'aide de matériel dédié sur ASIC. Cette solution présente le meilleur rapport consommation/performance/surface. Elle limite également la difficulté d'intégration dans une plateforme complète, puisque les communications ne sont pas la seule fonctionnalité attendue d'un système informatique moderne. Il n'y a en effet pas besoin de rajouter de connaissance de la

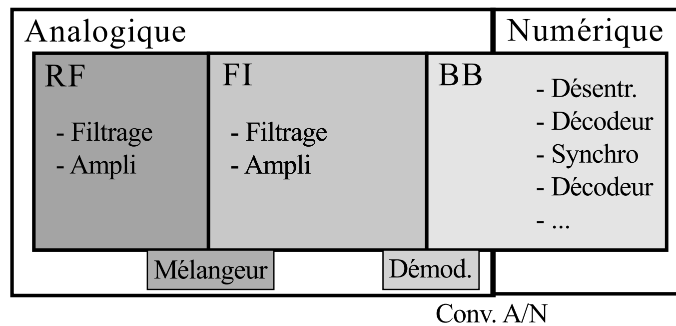


FIGURE 2.3 – Représentation d'un récepteur superhétérodyne : implantation et fréquences

chaîne de communication dans le système, uniquement de gérer un périphérique.

La multiplication des normes réseaux remet en cause la pertinence de cette approche classique car la conception de terminaux "multistandards", c'est-à-dire qui supportent plusieurs normes de communication, imposerait la multiplication d'implantations dédiées. Il y a donc une limite rapidement atteinte quant au coût de la multiplication des implantations matérielles dédiées.

2.2.2 Radio reconfigurable

2.2.2.1 Vision générale

Un terminal radio reconfigurable est un terminal qui utilise le même matériel pour exécuter plusieurs chaînes de communication différentes (les applications radio). On distingue ici deux propriétés :

- l'adaptabilité [DZD+05][GC97][KH00], qui s'applique plutôt à la chaîne de communication, et qui lui permet de s'adapter à l'environnement ambiant en faisant varier un ensemble de valeurs connues, comme par exemple le taux de codage ou la constellation de la modulation ;
- la reconfigurabilité [DZD+05][KMR00], qui est une propriété du terminal, et qui définit sa capacité à modifier en profondeur la structure de l'application ou des éléments qui la composent.

Ces deux propriétés sont bien sûr non exclusives. Lorsqu'on parle de radio reconfigurable, on s'intéresse donc à des implantations de terminaux radio qui permettent une modification profonde de la chaîne de communication.

2.2.2.2 Software Radio

La radio logicielle ("software radio" en anglais) est une mise en œuvre de la radio reconfigurable, dans laquelle les éléments qui composent une chaîne de communication sont implantés en logiciel, et sont donc exécutés par des processeurs génériques [Mit92][Jon05][Tut02].

La radio logicielle représente la vision idéale de la radio reconfigurable. L'antenne est directement reliée à un échantillonneur (convertisseur analogique numérique), qui peut fonctionner à une fréquence suffisante pour respecter le critère de Shannon-Nyquist, c'est-à-dire

$$f_s \geq 2 \times f_c$$

avec f_c la fréquence de l'onde porteuse, et f_s la fréquence d'échantillonnage. Les modifications de domaines par rapport à la radio classique sont présentés sur la figure 2.4. Il n'y a plus de fréquence intermédiaire, toutes les opérations sont réalisées directement en numérique, et sur un processeur générique (*General Purpose Processor* (GPP)).

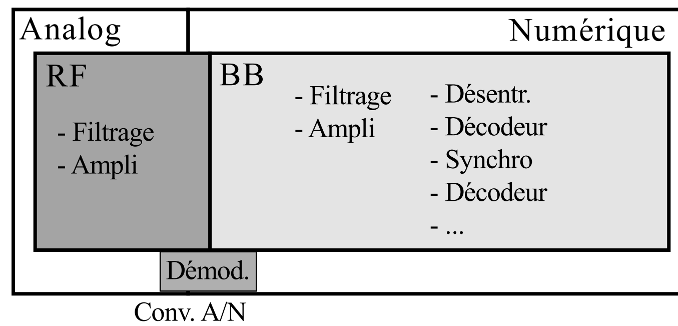


FIGURE 2.4 – Représentation d'un récepteur idéal de radio logicielle

Cependant, dans des systèmes radio, on travaille sur des fréquences dont l'ordre de grandeur varie entre la dizaine de mégahertz (10^7 Hz), et la dizaine de gigahertz (10^{10} Hz). L'état de la technique sur les échantillonneurs commerciaux permet actuellement d'atteindre, à un prix élevé, des fréquences de l'ordre de 3 Géch/s. À titre d'exemple, les convertisseurs de National Semiconductor, sur 8 bits, à 3 Géch/s (référence : ADC083000CIYB-ND), se vendent aux alentours de 800 dollars l'unité (en juin 2011). La cible n'est donc pas l'électronique grand public.

Au-delà du problème de l'échantillonnage, si on prend la norme WiFi qui opère à 2.4 GHz, traiter 4.8×10^9 échantillons par seconde avec des opérations de traitement du signal complexe est un défi intéressant mais difficile à relever avec les capacités de traitement actuelles. D'après une étude des normes *Global System for Mobile communications* (GSM) et *Universal Mobile Telecommunication System* (UMTS), [Alh10] estime les besoins en calcul pour le GSM (porteuse autour de 900 MHz) à 10 GIPS, et pour l'UMTS (même porteuse) à 100 GIPS. On notera que les études divergent quant à cette complexité. Ainsi, [TR08] estime à 100 MIPS le GSM, et à 6 GIPS le *Wideband Code Division Multiple Access* (WCDMA) (comparable à l'UMTS).

L'UMTS n'est donc même pas théoriquement réalisable sur un processeur Intel Xeon actuel, malgré ses 4 cœurs et 8 threads potentiels. Réaliser une plateforme de radio logicielle requiert la conception d'une architecture multiprocesseur intégrée à très haute performance. Ceci est réalisable, mais pose des problèmes de coûts, de consommation, de surface, et également de contrôle logiciel.

2.2.2.3 Software-Defined Radio

La radio logicielle est donc un concept, qui est atteignable en utilisant une plateforme dédiée, mais qui dans la pratique souffre de problèmes de consommation et d'encombrement qui limitent son développement commercial.

Afin de pallier ce problème de performance requise, le concept de *Software-Defined Radio* [Jon05] (radio logicielle restreinte [Alh10] en français) fait intervenir un traitement préliminaire de l'onde, afin de réduire la fréquence en entrée du système logiciel. La partie logicielle ne s'applique donc qu'à un sous-ensemble du terminal radio complet. Il y a également une généralisation du terme logiciel, qui ne désigne plus simplement l'exécution sur un processeur généralisé, mais l'utilisation d'une architecture qui permet la reconfiguration (GPP, DSP, FPGA, ou autres fonctions dédiées reconfigurables).

On présente sur la figure 2.5 les domaines correspondants à la SDR. La partie RF est gérée par une entité matérielle, qui se charge d'appliquer les filtres et de sélectionner uniquement la bande de fréquence souhaitée. La translation de fréquence vers une fréquence intermédiaire est également effectuée par ce *front-end* RF. Le signal analogique résultant est ensuite numérisé, puis transmis au logiciel SDR, qui est donc en charge du traitement numérique du signal. Ce logiciel peut s'exécuter sur un FPGA, sur un DSP, ou encore sur un processeur classique.

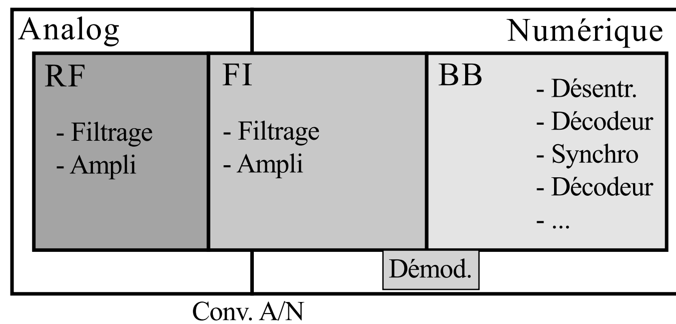


FIGURE 2.5 – Représentation d'un récepteur réalisable de SDR

2.2.3 Radio flexible

La radio flexible est un concept distinct de la radio reconfigurable, d'après [DZD+05]. C'est également un concept extrêmement flou. [DZD+05] définit la notion de flexibilité comme un ensemble de propriétés qui permettent une implantation évolutive de la radio, comme par exemple la reconfigurabilité, l'adaptabilité, la modularité, l'extensibilité,... D'après [PRR+03], un terminal présentant une seule de ces caractéristiques peut être considéré comme un terminal flexible. On peut également trouver le terme de radio logicielle étendue.

La flexibilité englobe de nombreux domaines. On s'intéresse principalement ici à la flexibilité de la couche physique, ainsi qu'à la mise à disposition de cette flexibilité dans les couches supérieures. Ainsi, un protocole de routage "flexible" sort du domaine abordé, alors que la prise en compte dans la couche MAC de la possibilité de changer la modulation, par exemple, en fait partie.

2.2.4 Radio cognitive

2.2.4.1 Terminal de radio cognitive

Finalement, la dernière notion liée à la radio flexible est la radio intelligente (cognitive radio) [MM99][Jon05]. Un terminal de radio cognitive est capable d'analyser son environnement et de réagir aux potentiels changements. Dans un monde idéal, un terminal de radio intelligente est un terminal capable de communiquer avec n'importe quelle autre entité, intelligente ou non, en s'adaptant aux modes de communications des autres entités. Il est également capable d'opérer dans des conditions de transmission difficiles, ou encore de s'intégrer dans des "trous" de communication d'autres entités (radio opportuniste). Il se base sur les capacités de la radio flexible, et permet d'optimiser l'utilisation du spectre.

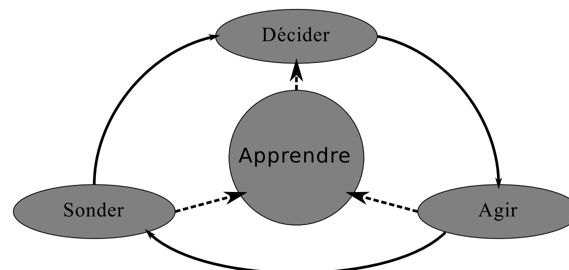


FIGURE 2.6 – Cycle simplifié d'un terminal intelligent [MM99]

Un terminal de radio cognitive peut être décrit de façon simplifiée comme sur la figure 2.6 [MM99]. Le terminal surveille en permanence son environnement. Cet environnement peut être

défini de différentes manières, selon l'application. En fonction du résultat de cette surveillance, le terminal décide quelle sera sa configuration, et transmet à l'unité de reconfiguration les modifications à effectuer. Tout ce cycle s'organise autour d'un apprentissage. La radio cognitive peut être utilisée dans différentes applications. Par exemple, elle peut permettre de créer un terminal radio qui peut se connecter à n'importe quelle norme radio à portée, afin de communiquer avec n'importe quel terminal. Mais l'une des utilisations principales du concept de radio cognitive est actuellement l'accès dynamique au spectre radio (*Dynamic Spectrum Access*, DSA).

2.2.4.2 Radio opportuniste

La radio opportuniste est un effort pour optimiser l'utilisation des bandes de fréquence, sous licence ou non. Un réseau opportuniste est un réseau secondaire, qui communique en présence d'autres réseaux (le réseau primaire étant le réseau officiel) de manière invisible pour ces autres réseaux.

La radio opportuniste temporelle s'attache à combler les trous dans une bande de fréquence donnée. Le projet Oracle [ora] est un exemple de ce type d'application. Il vise à créer un réseau secondaire permettant l'échange d'informations alors qu'un réseau primaire de type WiFi, utilisant le CSMA/CA, est présent.

La radio opportuniste fréquentielle ou spatiale cherche à utiliser les fréquences sous licences, mais non utilisées dans une zone donnée du fait de la répartition en cellules du réseau. L'exemple typique de cette application est l'utilisation des *TV White Space* (TVWS)[qos][cog]. La diffusion de la télévision se fait sur des bandes de fréquences sous licence. Cependant, afin d'éviter les interférences, le réseau de diffusion est divisé en cellules. Deux émetteurs adjacents ne se verront pas allouer la même bande de fréquence. Il existe donc des trous dans l'utilisation du spectre fréquentiel. Le passage au tout numérique pour la télévision est également une opportunité pour cet axe de recherche, avec la libération des bandes de fréquence précédemment utilisées par la télévision analogique.

2.2.5 Interactions radio flexible/radio cognitive

2.2.5.1 Différences de terminologie

Selon les études sur le sujet, on peut trouver différentes terminologie pour les différents domaines abordés précédemment. Ainsi, la radio flexible dans notre cas représente une implantation flexible d'une application radio, alors que dans d'autres études, elle peut représenter le domaine général de la radio évolutive, et donc inclure la radio cognitive. De même la radio logicielle peut également se décliner en radio logicielle étendue, et inclure ce que nous appelons la radio flexible.

2.2.5.2 Architecture d'un terminal radio

Un terminal radio évolutif peut donc être composé de deux éléments :

- la partie cognitive, qui se concentre sur la prise de décision, et sur la réponse à apporter aux modifications d'environnement ;
- une implantation flexible, qui exécute les applications radio, et offre la possibilité de modifier son fonctionnement.

Cette découpe en deux couches de l'implantation cohabite avec la découpe en couche du modèle OSI. On présente sur la figure 2.7 une architecture conceptuelle d'un terminal radio.

Le sondage effectué par la radio cognitive correspond en fait à un retour d'information fait par l'implantation de l'application radio. Cette implantation correspond au domaine de la radio flexible. Toutes les couches du modèle OSI peuvent être concernées. De même, la phase d'action du cycle de la radio cognitive correspond finalement à une modification de l'implantation, elle

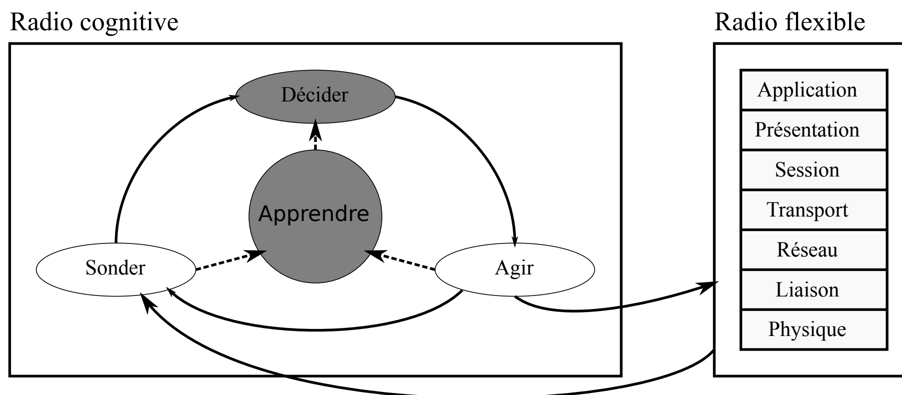


FIGURE 2.7 – Interactions entre radio flexible et radio cognitive

correspond donc à des ordres donnés à la partie radio flexible du terminal. On entrevoit déjà ici la nécessité de faciliter ces ordres et l'utilisation de la radio flexible.

2.2.6 Conclusion

Il existe donc plusieurs manières d'aborder la radio flexible. On approfondira dans cette étude les notions de reconfigurabilité et d'adaptabilité, et plus particulièrement leur intégration dans le contexte de la radio flexible et cognitive.

L'un des avantages majeurs de la radio flexible (et cognitive) réside dans le moment de la prise de décision. Alors que dans un système classique, la prise de décision se fait en amont de la spécification du réseau, dans la radio flexible, cette prise de décision est distribuée, à la fois dans le temps et entre les différents acteurs, de la spécification à l'exécution. Le système est ainsi capable de changer dynamiquement son fonctionnement pour s'adapter à son environnement.

Cependant, l'ajout de la flexibilité à un terminal ne se fait pas sans heurts. Un terminal de radio classique est simple en termes d'intégration et de contrôle. Il prend en entrée des données à transmettre, quelques paramètres pour le contrôle, et fonctionne en boîte noire. Plus un terminal devient flexible, plus ce contrôle et cette intégration se compliquent.

2.3 Radio flexible : de multiples possibilités

2.3.1 Scénarios de reconfiguration

Dans la radio flexible, la reconfiguration peut suivre différents scénarios [KM02][MBGS03] selon l'objectif recherché. Chaque scénario présente sa propre problématique.

2.3.1.1 Adaptation

Tout d'abord, l'adaptabilité permet de modifier le fonctionnement de la radio dans le but de l'adapter au réseau. Dans un tel scénario, un terminal, ou le réseau lui-même, intègre des algorithmes de prise de décision qui lui permettent, selon son environnement, de modifier son fonctionnement.

La complexité réside dans les critères et les algorithmes permettant de prendre les décisions d'adaptation. Le champ de la modification est généralement connu, et la transformation du terminal reste limitée à une gamme de fonctionnement finie et maîtrisée, contrôlable par un jeu de paramètres numériques (*cf.* 2.2.2.1).

2.3.1.2 Évolution des normes

Un deuxième scénario possible est le scénario de la mise à jour de la radio. La prise de décision n'est pas intégrée à la radio, mais est effectuée par des acteurs externes au système. En règle générale, ces acteurs sont les éléments responsables de la maintenance du système.

Il existe de nombreuses illustrations de ce cas de figure. Par exemple, dans le domaine de la téléphonie, ce scénario peut être utilisé pour toutes les évolutions mineures des normes, comme par exemple le passage de l'UMTS à l'HSPA, sans avoir à modifier toute l'infrastructure (et donc à changer les stations de base, éléments coûteux du réseau). Dans ce cas, la décision de mise à jour est prise par l'opérateur responsable de l'infrastructure à faire évoluer.

Ce scénario fait appel à la notion de reconfigurabilité, mais dans un cadre statique. Dans le cadre d'une mise à jour, on peut se permettre d'éteindre temporairement un système. La décision de reconfiguration est externe au système, et les algorithmes de prise de décision du système sont donc inexistants.

Par contre, le besoin de reconfigurabilité est plus large que pour l'adaptation. Dans le cadre de l'adaptation, seule une partie du système est modifiée, dans le respect de la spécification initiale des éléments le constituant. Cela peut être fait en cours de fonctionnement. Dans le cadre de la mise à jour, une nouvelle version des éléments, répondant à de nouvelles spécifications, peut être chargée, venant potentiellement modifier l'intégralité de la chaîne de communication.

2.3.1.3 Multimode

Le cas du multimode est le plus complexe et le plus délicat. Dans un scénario multimode, la reconfiguration a lieu afin de permettre l'utilisation de plusieurs réseaux différents en utilisant le même matériel.

L'exemple du *smartphone* est une bonne illustration du scénario. Ces appareils sont généralement équipés de plusieurs méthodes de connexion sans fil, les plus courantes étant la téléphonie (GSM, GPRS, EDGE, UMTS, HSPA), les réseaux locaux (WiFi) et les réseaux personnels (Bluetooth). La coexistence de ces différents réseaux se fait généralement par la coexistence de plusieurs ASIC figés, chacun étant spécifié pour exécuter une norme.

Afin d'éviter cette multiplication des ASIC, il est envisageable de mettre en place un composant générique, capable d'exécuter toutes ces normes, et de se reconfigurer selon les besoins de manière dynamique. Ce scénario fait appel à la reconfigurabilité, mais également à des algorithmes de prises de décision. La reconfiguration est temporellement beaucoup plus contrainte que dans le cas d'une évolution, et doit être faite dynamiquement et de façon transparente pour l'utilisateur (ou le système qui utilise les différentes chaînes de communication).

2.3.2 De multiples cibles de reconfiguration

La reconfigurabilité d'une plateforme peut s'atteindre de différentes manières, selon les besoins liés au scénario envisagé. Le problème du choix de la solution est intimement lié au compromis classique performance/flexibilité, comme présenté sur la figure 2.8 [Kha02][MGT01]. On parle ici de performance énergétique, c'est à dire du rapport entre la performance (en termes de capacité de calcul) et l'énergie dépensée pour atteindre cette performance.

2.3.2.1 Cibles logicielles

La cible logicielle est la plus flexible des possibilités, dans l'état actuel des connaissances. En effet, si on ne considère pas la performance et les difficultés pratiques de mise en place de la radio logicielle, un processeur est virtuellement capable d'exécuter n'importe quelle opération, tant qu'on lui fournit le programme qui la décrit.

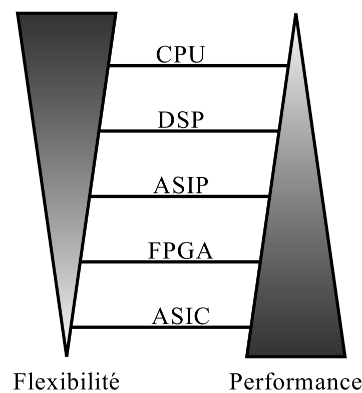


FIGURE 2.8 – Représentation du rapport flexibilité/performance de différentes solutions d'implantation

En pratique, il est difficile d'ignorer les aspects de performances et de consommation. Dans un système embarqué, la consommation d'un processeur est nettement supérieure à celle d'un ASIC. Les implantations utilisant des ASIC sont certes figées, mais extrêmement performant avec une consommation réduite. Les aspects de consommation mis à part, la question de la performance peut être abordée d'un autre point de vue : la séparation entre l'application elle-même et l'exécutant de cette application. Dans la radio traditionnelle, le concepteur de l'application conçoit le circuit qui permettra d'implanter une norme. Dans ce but, il recherche un compromis visant à réaliser un circuit de petite taille répondant aux exigences de débit imposées par l'application tout en tenant compte de la technologie du circuit et de l'ergonomie finale du produit. Dans la radio logicielle, le concepteur déploie l'application sur une cible dont la puissance de calcul est limitée et indiquée par le fondeur du processeur.

Cette limite externe de performance est compensée par la mise en place d'architectures à base de multiples processeurs hétérogènes (DSP), qui augmentent d'autant la complexité de mise en œuvre. En effet, le surcoût lié à la gestion de ces architectures, qui s'ajoute à la complexité de la gestion de la radio logicielle elle-même (chemins de données, contrôle, interactions avec les couches hautes, reconfiguration, ...), rendent ces implantations extrêmement lourdes. La complexité de la radio logicielle n'est pas liée à l'implantation d'algorithmes de traitement du signal qui sont déjà connus, mais à l'intégration de ces algorithmes et à la gestion dynamique de leur exécution. Ceci rejoint des problématiques d'architecture des systèmes intégrés.

2.3.2.2 Matériel reconfigurable

La définition d'accélérateurs matériels reconfigurables permet de répondre en partie au problème de performance de la radio logicielle. Cependant, la perte de généricité liée à l'utilisation de matériel est sensible.

Afin de définir du matériel reconfigurable efficace, c'est-à-dire permettant d'atteindre des performances suffisantes en gardant la place requise sous contrôle, il est nécessaire de trouver un compromis entre généricité et performance. Alors que les approches logicielles font le choix de la généricité "absolue", les solutions de matériels reconfigurables se basent sur des éléments ayant une fonctionnalité bien définie (par exemple, un opérateur FFT), mais qui peuvent modifier leur fonctionnement (par exemple, changer le nombre de points de la FFT).

À la différence des techniques logicielles (ou apparentées), on définit ici du matériel dédié, qui restera en place. Dans les solutions logicielles, on charge une fonction logicielle sur un processeur générique, qui va exécuter cette fonction. La reconfiguration est simplement un changement de fonction. Dans une solution à base de matériel reconfigurable, l'exécutant de la fonction, c'est à dire l'accélérateur matériel, est confondu avec la fonction, et la reconfiguration est donc limitée à

sa gamme d'exécution.

Ces approches font apparaître de nouveaux problèmes, en particulier des problèmes de contrôle du matériel. Il n'est pas envisageable, dans un cadre flexible, de supprimer complètement le logiciel. L'interaction entre le contrôle logiciel et le matériel est un point crucial des environnements à base de matériel reconfigurable. De plus le nombre de possibilités pour répondre à un besoin donné est tel qu'il n'y a pas d'uniformisation et qu'il est nécessaire de fonctionner au cas par cas pour le contrôle.

Finalement, il pourrait être envisageable de mettre en place des solutions mixtes à base de matériel reconfigurable et de logiciel. De telles méthodes combinent les avantages du logiciel et du matériel : tant que l'application reste dans le cadre du matériel défini, il est possible d'utiliser ce matériel. Mais si un besoin ponctuel non géré par le matériel apparaît, il est possible d'absorber cet écart en utilisant le logiciel. Ces approches sont cependant difficiles à mettre en place, à cause des transferts de données requis, et de la synchronisation.

2.3.3 Conclusion sur la radio flexible

Les possibilités d'utilisation et d'implantation de la radio flexible sont donc très vastes. L'objectif est d'obtenir des terminaux capables de s'adapter à leur environnement. Ceci passe en partie par la possibilité de reconfigurer le terminal. Les techniques de radio flexible étaient pressenties, au début des années 2000, comme les solutions miracles, qui seraient adoptées très largement dans un délai de 10 ans [Tut02]. Même si des progrès ont été faits, ces solutions restent encore majoritairement expérimentales.

Dans le domaine de la radio reconfigurable, les recherches se concentrent généralement sur la définition de composants permettant une reconfiguration et sur des architectures intégrant ces composants. Le problème réside dans le peu d'interopérabilité entre les différentes implantations. La décision d'utiliser la radio flexible pour une implantation commerciale impose le choix d'une solution. Ce choix est déterminant car le développement qui sera réalisé sur cette solution ne pourra pas être utilisé pour une autre méthode.

Au delà de ce problème du choix de l'architecture reconfigurable, se pose le problème de l'intégration de cette architecture avec les algorithmes de radio flexible ou cognitive. Il y a un besoin de faire coopérer les éléments qui gèrent ces algorithmes (généralement, des éléments logiciels rattachés à la couche MAC), aux éléments qui vont effectivement répercuter les décisions (la partie reconfigurable). La multiplication des possibilités posent également un souci dans ce cadre, puisque pour chaque méthode de reconfigurabilité, il faut revoir l'intégration.

2.4 Choix de l'implantation : évolution

La radio logicielle utilise donc des processeurs pour implanter la chaîne de communication de la radio. Cependant, cette solution d'implantation se heurte à de nombreux problèmes, en particulier des problèmes de performance et de gestion des différentes tâches.

2.4.1 Performance de la radio logicielle

Les chaînes de communication radio sont généralement implantées en matériel, dans un environnement non flexible. En effet, elles utilisent un modèle de *pipeline* de tâches particulièrement adapté à une implantation matérielle, et les opérations très coûteuses utilisées dans les normes sans fil rendent l'optimisation nécessaire pour atteindre les débits requis.

Alors que la radio logicielle est une solution optimale en termes de flexibilité, la performance n'est pas vraiment au rendez-vous. En effet, le modèle *pipeline* qui est utilisé ne se prête pas

vraiment aux processeurs, qui ne peuvent exécuter qu'une opération à la fois, et qui doivent donc rendre les calculs séquentiels.

Ceci détériore considérablement les performances en terme de débit atteignable des solutions de radio logicielle. L'utilisation de plateformes multiprocesseurs est une solution aux problèmes d'adéquation application/architecture, puisque l'implantation d'un pipeline devient possible, mais ne résout pas les problèmes d'optimisation.

Les processeurs à simple cœur ne se trouvent plus que dans les systèmes embarqués pour lesquels l'énergie consommée est un élément clé du système. Cette séquentialisation du *pipeline* n'est donc que partielle. De plus, la forte progression de la puissance de calcul des processeurs actuels permet également d'atteindre des débits intéressants. Cependant, d'autres solutions peuvent être envisagées, qui permettraient de libérer les processeurs pour son usage de base, tout en offrant potentiellement un meilleur débit accessible.

2.4.2 Essor du GPGPU

Le calcul général sur processeur graphique (*General Purpose computing on Graphics Processing Unit* (GPGPU)) est un domaine de recherche assez récent, lié à l'évolution des *Application Programming Interface* (API) graphiques. Alors que les opérations étaient très ciblées au départ, le besoin en généricité pour certains calculs est devenu de plus en plus important, ce qui a mené à une utilisabilité du GPU pour des calculs autres que des calculs purement graphique.

Un GPU est une architecture à base d'une multitude de cœurs de calculs dégénérés. Ces cœurs sont uniquement capable d'exécuter des opérations, mais pas de gérer toutes les tâches annexes d'un processeur habituel, c'est à dire la gestion d'un programme, les *branch*, et autres opérations non calculatoires.

La gestion de ces opérations annexes est déléguée à une entité séparée, qui contrôle les cœurs. C'est donc une architecture de type *Single Instruction Multiple Data* (SIMD) (ou SPMD selon les cas), capable d'exécuter une seule instruction ou programme sur une multitude de données.

Les GPU étant à l'origine dédiés au traitement de l'image, de nombreuses opérations de type traitement du signal sont proposées et optimisées.

2.4.3 Problèmes du GPGPU pour la radio logicielle

Il peut paraître intéressant d'utiliser le GPU pour augmenter encore le débit atteignable par des applications de radio logicielle. Cependant, le problème d'adéquation application/architecture est encore plus présent dans ce cas. Alors que la parallélisation des processeurs permet d'exécuter plusieurs étages du *pipeline* en parallèle, la parallélisation interne du GPU, de type SIMD, ne permet pas cette amélioration.

Si on prend l'exemple de la radio logicielle sur un ordinateur standard, avec deux processeurs à 4 cœurs, cette architecture permet en théorie d'exécuter en concurrence 8 étages du *pipeline*. Afin d'avoir un intérêt en termes de performances pures à utiliser un GPU, il faudrait que celui-ci exécute ces 8 étages séquentiellement plus rapidement, ce qui implique un gain moyen par étage supérieur à 8, ce qui est très élevé.

L'intégration d'applications de radio logicielle sur un GPU n'est donc pas intuitive, et il n'est pas garanti qu'un gain soit observable.

2.5 Environnement de reconfiguration unique

2.5.1 Objectifs

Le développement de la radio cognitive et la multiplication des plateformes cibles pour la radio logicielle créent de nouveaux besoins. Afin de permettre l'exécution efficace de la radio cognitive,

et l'utilisation optimale des plateformes, il est nécessaire d'intégrer les différentes techniques de radio flexible (matériel reconfigurable, logiciel sur GPP, DSP ou GPU, ...) dans un même environnement. Il est également nécessaire de faire collaborer ces solutions avec des algorithmes de prise de décision qui permettent de bénéficier au mieux de la reconfigurabilité pour le protocole. Cette collaboration pose le problème plus général de l'intégration des différents domaines de la radio flexible.

L'intelligence de la radio flexible est généralement implantée en logiciel, alors qu'il existe différentes manières d'implanter la partie opérative. Cette intégration peut dès lors être envisagée comme un simple programme logiciel, qui utilise les unités fonctionnelles matérielles ou logicielles en ayant une connaissance complète de ce qu'ils font. Si on prend l'exemple du H-ARQ, la partie protocolaire, qui prend la décision de choisir une certaine configuration du composant, répercuterait cette décision directement sur la partie PHY. Cependant, le moindre changement dans la plateforme, que ce soit sur l'environnement logiciel, ou sur la partie PHY de l'H-ARQ, impose d'implanter à nouveau le protocole.

Dans cette section, on s'intéresse aux différents problèmes à résoudre afin d'obtenir un environnement unifié et efficace pour la radio flexible. On utilise, à cette fin, la plateforme représentée dans la figure 2.9. On y trouve des éléments matériels reconfigurables, des éléments d'exécution logicielle (processeurs, spécialisés ou non), des éléments de mémorisation, et des entrées/sorties (E/S) classiques d'un système de ce type.

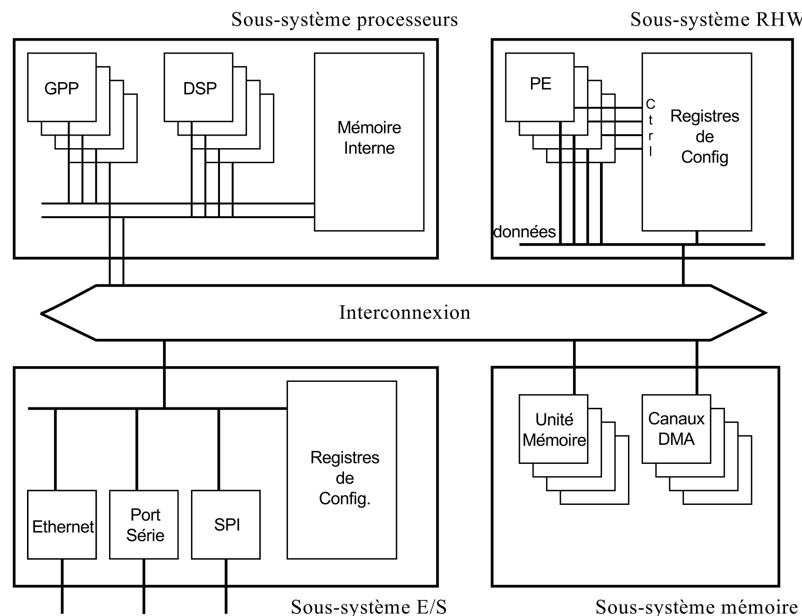


FIGURE 2.9 – Architecture de plateforme flexible générique

On envisage donc de mettre en place un environnement pour la radio flexible capable de s'exécuter sur cette plateforme générique, et de s'adapter aux modifications de la plateforme.

2.5.2 Gestion de la reconfigurabilité

La reconfigurabilité est une propriété liée à l'implantation [KMR00]. Lorsqu'un système possède cette propriété, son fonctionnement est susceptible d'être modifié.

Cette propriété de reconfigurabilité s'obtient de différentes manières, comme évoqué en section 2.2. La multiplication des implantations reconfigurables, qu'elles soient logicielles ou matérielles ou les deux, est le premier point à gérer lors de l'intégration des différents éléments de la radio flexible dans un environnement unique.

La mise en place de solutions reconfigurables passe souvent par une phase de sélection du type de reconfigurabilité souhaité, en fonction des avantages des différentes méthodes. Une fois ce type choisi, on intègre les éléments de reconfigurabilité soit au niveau du système d'exploitation (pour les éléments matériels), soit directement au niveau de l'application (pour les éléments logiciels et matériels).

Ce manque d'abstraction nuit à la flexibilité, car il empêche d'intégrer des solutions qui pourraient être avantageuses en termes de reconfigurabilité, et limite également le développement algorithmique et protocolaire, puisqu'il faut à chaque fois adapter l'implantation de l'algorithme à la plateforme que l'on utilise. De manière générale, la partie protocolaire du réseau n'a pas besoin de connaître les détails du fonctionnement des éléments reconfigurables. Que ceux-ci soient logiciels ou matériels, le seul objectif pour le protocole est d'être capable de signaler le mode de fonctionnement qui convient, quel que soit la méthode utilisée pour l'implanter.

La gestion de la reconfigurabilité au niveau de l'environnement est donc un point clé d'un environnement unifié. Cette gestion soulève plusieurs interrogations. Tout d'abord, les couches hautes ont besoin d'une abstraction des opérateurs nécessaires à la radio, qu'il convient de définir. Ensuite, les mécanismes de répercussion de modifications depuis la couche haute jusqu'à l'élément permettant effectivement de réaliser l'opération ne sont pas immédiats. Finalement, en plus de la gestion de la reconfigurabilité au niveau de la fonctionnalité, il faut aussi se poser la question de la reconfigurabilité des connexions entre les différents éléments.

2.5.3 Intégration de la radio flexible

Les environnements existant actuellement sont principalement des environnements de reconfiguration spécifiquement conçus pour une solution. Cependant, afin de permettre l'utilisation de la radio flexible dans les meilleures conditions possibles, il est intéressant de se pencher sur l'opportunité d'étendre ces environnements reconfigurables à des environnements flexibles, c'est-à-dire intégrant les algorithmes de prise de décision propres à la radio flexible (ou par extension à la radio cognitive).

Les algorithmes dits "cross-layer" qui tendent à augmenter la coopération entre les couches traditionnelles du réseau, sont à mettre sur le même plan que les algorithmes flexibles. Ils soulèvent un problème plus large, puisqu'il peut également être nécessaire de modifier le fonctionnement de la couche MAC.

Cette intégration n'est donc ni évidente ni immédiate. Il faut, pour la rendre effective, résoudre le problème de la place des algorithmes propres à la radio flexible dans le réseau, c'est-à-dire de leurs interactions avec les couches MAC et PHY, du réseau.

2.5.4 Gestion de la reconfiguration

La reconfiguration se gère à différents niveaux [KM02], en fonction de la granularité souhaitée, et des connaissances du niveau considéré. Au niveau de la gestion d'un composant matériel unique, la reconfiguration est un processus purement système, alors qu'au niveau d'un terminal radio complet, la reconfiguration impose la connaissance des normes réseaux, et de l'environnement.

La gestion de la reconfiguration couvre plusieurs points :

- une représentation de l'état d'un système pour les couches de haut niveau. Ce point est primordial afin de permettre l'intégration des couches hautes évoquée précédemment. Les algorithmes de prise de décision, qui vont décider de l'opportunité et de la teneur d'une reconfiguration, ont besoin d'une représentation de l'état d'un système uniformisé, et indépendante de la plateforme ;
- un mécanisme de traduction de cette représentation en une représentation compréhensible pour une plateforme donnée. Celui-ci s'apparente à une compilation d'une configuration sur une plateforme donnée.

– une gestion des états possibles, qui inclut un mécanisme de changement d'état. Ces mécanismes sont mis en œuvre à l'exécution, et peuvent avoir à supporter des contraintes temporelles fortes.

2.5.5 Conclusion : besoins d'un environnement flexible

La radio flexible est une technique d'implantation de terminaux radio qui permet d'envisager une meilleure coopération des éléments du réseau, et donc d'atteindre une meilleure utilisation des ressources. Cependant, elle englobe plusieurs solutions, à base de logiciel ou de matériel reconfigurable, qui peuvent s'exécuter sur différentes cibles.

Il devient donc nécessaire d'unifier ces solutions dans un même environnement, afin de permettre un vrai développement de la radio flexible. Un tel environnement permettrait en effet de rassembler de manière homogène les différents domaines impliqués dans la flexibilité, en les isolant les uns par rapport aux autres et donc en leur masquant les difficultés et spécificités des autres domaines.

Afin de concevoir cet environnement, il faut se pencher sur les différents éléments qui composent la radio flexible, et sur les interactions entre ces éléments.

Des ébauches d'environnement existent actuellement, mais qui ne permettent pas d'intégrer n'importe quelle solution, et surtout qui n'intègrent pas des éléments des couches hautes, pourtant indissociables de la radio flexible.

2.6 Conclusion générale : objectifs des travaux

En résumé, ces travaux se placent dans le contexte de la radio flexible. Deux axes principaux sont étudiés, l'un portant sur l'utilisation du GPU dans un contexte de radio logicielle, afin d'améliorer les performances, et l'autre axe traitant de l'unification des différents domaines et des diverses solutions de la radio flexible. Ces deux axes se rejoignent en un seul objectif, permettre l'intégration d'unités hétérogènes pour une application de radio flexible.

Plus précisément, nous tenterons de répondre aux questions suivantes :

– Comment utiliser de manière efficace le GPGPU pour la radio logicielle ? Les GPU proposent une architecture assez peu adaptée aux application de radio logicielle, cependant leur puissance potentielle de calcul est énorme, et pouvoir les utiliser efficacement permettrait d'envisager des débits qui sont pour l'instant une utopie ;
– Comment intégrer les opérations sur GPU dans une application radio ? Au delà de l'utilisation efficace théorique, il est important de pouvoir intégrer efficacement ces opérations dans une vraie application, qui devra probablement faire appel à d'autres unités ;
– Quelle solution adopter pour utiliser les accélérateurs matériels dans une application de radio flexible ? Ces accélérateurs peuvent offrir un certain niveau de flexibilité tout en permettant d'atteindre un meilleur débit qu'une solution logicielle, avec un coût énergétique moindre. Intégrer ces accélérateurs peut donc permettre d'envisager des plateformes flexibles embarquées ;
– Au delà de ces considérations d'intégration d'unités spécifiques, le nombre de possibilités pour implanter un terminal radio a considérablement augmenté ces dernières années. La structure en couches d'une norme réseau impose cependant aux couches de haut niveau de connaître les couches de bas niveau disponibles. La multiplication du nombre de solutions d'implantation est donc un frein à la recherche. Dès lors, le problème de l'intégration de ces différentes solutions devient primordiale. Deux problèmes se posent alors, en plus des questions évoquées précédemment :
  – quelle architecture définir pour permettre l'exécution d'une application sur ces architectures hétérogènes ?

– comment décrire et traduire une application radio, afin de permettre une certaine généricité dans la description, et de s'adapter à n'importe quelle plateforme qui l'exécutera ?

Chapitre 3

État de l'art

Dans ce chapitre, nous présentons un ensemble non exhaustif de solutions liées à la radio flexible, et plus particulièrement à la radio reconfigurable.

Étant donnée l'activité intensive de recherche autour des problématiques de radio cognitive, la littérature sur ce sujet est abondante. Elle aborde le domaine sous plusieurs angles, qui vont des algorithmes de prise de décision propres à la radio flexible et cognitive, à des implantations reconfigurables d'éléments bien précis. Nous avons fait le choix de présenter particulièrement deux aspects de la radio flexible, qui ont fait l'objet de nos recherches durant cette thèse.

Dans un premier temps, nous présentons des techniques d'exécution de la radio logicielle. Ces techniques concernent en particulier des plateformes et différentes approches de la reconfigurabilité dans la radio logicielle, et sont abordées dans la section 3.2. Dans un second temps, nous nous intéressons au contrôle de l'exécution et de la reconfiguration de ces plateformes, principalement d'un point de vue système.

3.1 Représentation d'une application radio . . . 22
- 3.1.1 Reconfiguration . . . 22
- 3.1.2 Représentation des applications . . . 22

3.2 Plateformes de radio logicielle . . . 23
- 3.2.1 Solutions à base de processeurs . . . 23
- 3.2.2 Intégration des GPU . . . 24
- 3.2.3 Utilisation de FPGA . . . 24
- 3.2.4 Solutions hybrides . . . 25
- 3.2.5 Matériel reconfigurable et adaptabilité . . . 25
- 3.2.6 Conclusion sur les plateformes . . . 26

3.3 Environnements . . . 26
- 3.3.1 Objectifs . . . 26
- 3.3.2 Exécution hétérogène . . . 26
- 3.3.3 Environnements complets . . . 28
- 3.3.4 Conclusion . . . 32

3.4 Conclusion du chapitre . . . 32

3.1 Représentation d'une application radio

3.1.1 Reconfiguration

3.1.1.1 Définition générale

On définit la reconfiguration comme un ensemble de 3 éléments [KM02] :
- une configuration initiale, avant la reconfiguration ;
- une configuration finale, après la modification ;
- un mécanisme de changement d'état, qui joue le rôle de protocole de reconfiguration.

On appelle configuration l'état dans lequel se trouve le système considéré. Cette configuration a une représentation, qui permet d'enregistrer l'état. Le mécanisme de changement d'état dépend de la cible considérée.

Prenons par exemple un noyau de système d'exploitation, le processeur étant la cible de la reconfiguration, alors la configuration du système est la valeur de ses registres, la représentation est un tableau de registres, comportant les valeurs correspondantes au programme en cours, et le mécanisme est la commutation de contexte. Si la cible de la reconfiguration est le couple processeur/mémoire, alors la configuration est représentée par l'état de la mémoire et par les registres du processeur, un changement de configuration pouvant faire intervenir un chargement de programme dans la mémoire.

3.1.1.2 Cas de la radio logicielle

Dans le cas spécifique de la radio logicielle, ou de la radio reconfigurable en général, la reconfiguration est un processus hiérarchique [DMLP05]. En effet, la cible de la reconfiguration peut se situer à différents niveaux d'abstractions.

Le premier niveau de reconfiguration est le plus haut niveau. C'est la partie cognitive de la radio, qui se situe au niveau de la spécification de la norme. On ne s'intéresse pas à l'exécution de la norme, ni aux spécificités de l'implantation, mais uniquement à ses fonctionnalités. Ce niveau utilise donc une représentation abstraite des opérations possibles. Par exemple, une opération de filtrage *Finite Impulse Response* (FIR), ou une opération de transformée FFT, mais sans connaissance du mode d'implantation de l'opération.

Ces opérations sont fournies par le second niveau, que nous appelons niveau fonctionnel. Ce niveau offre au niveau supérieur un ensemble d'opérations, et permet d'associer une implantation sur la plateforme courante. Il possède donc une connaissance des spécificités du système sur lequel s'exécute la norme.

Le dernier niveau s'intéresse aux éléments reconfigurables. Il n'y a plus de notion de fonctionnalités dans ce cadre, uniquement des éléments qui sont à configurer de manière à s'adapter aux ordres du second niveau. Un élément fonctionnel du second niveau peut avoir plusieurs implantation dans le dernier niveau. Par exemple, on peut implanter la même fonctionnalité sur un DSP et sur un GPP.

Cette hiérarchisation, issue de [DMLP05], est présentée dans la figure 3.1. Nous distinguons dans cette étude la reconfiguration et la représentation. Les deux sont très fortement liés. [DMLP05] ne détaille que la représentation.

3.1.2 Représentation des applications

Les applications radio sont appelées *waveform*. Celles-ci sont couramment représentées en utilisant un graphe, dans lequel les nœuds représentent les opérations, et les arêtes les transferts de données [DDL10]. Cette vision en graphe de l'application se prête bien à l'utilisation du modèle de programmation composant. Chaque bloc est décrit en fonction de ses entrées/sorties, et de

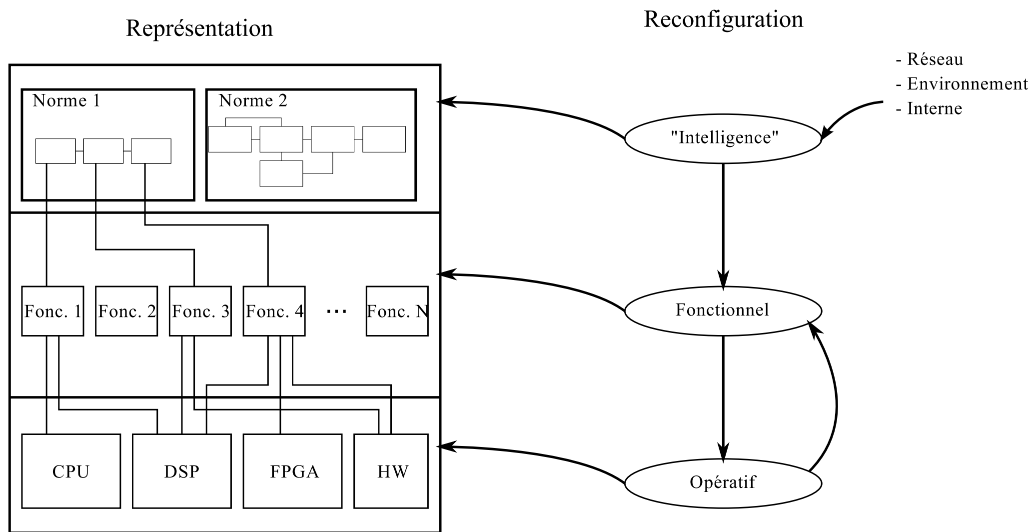


FIGURE 3.1 – Représentation et reconfiguration hiérarchique [DMLP05]

l'opération qui sera exécutée dans ce bloc. On peut ensuite connecter les différentes opérations en utilisant des FIFO.

On trouve différentes philosophies d'implantation des *waveforms*. On distingue principalement les implantations directes, comme celle de GNURadio [gnub], et les implantations indirectes comme *Software Communications Architecture* (SCA) [sca01]. Dans les implantations directes, la *waveform* est directement exécutable telle qu'elle. GNURadio utilise du C++ ou du Python pour écrire ses *waveforms*, qui sont donc directement exécutée. Au contraire, SCA utilise un langage de description pour décrire ses *waveforms*, qui sont ensuite utilisées par l'environnement pour être exécutées.

3.2 Plateformes de radio logicielle

La radio flexible est clairement dépendante de capacité d'un système à modifier son mode de fonctionnement.

D'après la définition donnée en 2.2.2.1, la reconfigurabilité est une propriété qui définit la capacité d'un système à modifier son fonctionnement au-delà d'un simple changement de valeur numérique. La reconfigurabilité est à inclure dans le processus de conception d'un terminal radio [KMR00].

Cette section s'attache à présenter certaines plateformes et approches permettant la reconfigurabilité, d'un point de vue architecture. On ne cherche pas à définir quelles techniques sont les plus efficaces, uniquement à balayer le plus largement possible les possibilités, ceci afin de pouvoir les intégrer dans un environnement unifié, comme expliqué dans le chapitre précédent.

3.2.1 Solutions à base de processeurs

La reconfigurabilité dans le logiciel est intrinsèque [KMR00]. En effet, il est très simple d'implanter plusieurs applications radio, et de passer de l'une à l'autre en déchargeant et rechargeant les codes compilés correspondant. La difficulté des approches logicielles réside dans le contrôle de ces chargements et déchargements, et dans la performance limitée des processeurs comparée à des solutions matérielles dédiées.

On définit généralement comme logiciel, dans la radio flexible, tout élément dont l'exécutant est séparé de la fonction qui va être exécutée. Ainsi, on peut déployer l'application sur une cible

comme un GPP, un processeur dédié comme un DSP ou un GPU, ou un FPGA, les trois cibles principales de la radio logicielle. Le choix de la cible est généralement dicté par le type d'application et le contexte d'exécution. Ainsi, les normes réseau qui requièrent beaucoup de calculs auront probablement besoin soit de nombreux processeurs génériques, soit d'utiliser des DSP. Dans un contexte embarqué, on préférera généralement les DSP ou les FPGA, qui offrent un meilleur rapport performance/énergie que les GPP.

La radio logicielle peut facilement être développée pour des plateformes qui ne sont pas spécifiquement dédiées à cet usage. Par exemple, le projet GNURadio [gnub] peut être déployé sur des architectures PC classiques, ou sur des architectures embarquées comme par exemple la BeagleBoard [bea10][gnua]. On trouve également beaucoup d'architectures développées spécifiquement pour exécuter des applications de radio logicielle. Ces plateformes peuvent être basées sur des DSP [Ram07][CAL] (avec l'aide d'accélérateurs matériels), des FPGA [HHP+09][MSA06][ASM+07]ou bien un mélange des deux [DMLP05]. Les plateformes à base de FPGA sont abordées dans la section suivante.

D'un point de vue applicatif, l'utilisation de GPP pour l'application transforme celle-ci en application usuelle, même si elle utilise un modèle particulier. Elle est compilée puis exécutée, et la reconfiguration de la chaîne de communication se fait en lançant un nouveau processus. L'intégration des DSP est plus complexe, mais reste dans le domaine du développement logiciel.

3.2.2 Intégration des GPU

Avec le développement de l'usage du GPU pour le calcul scientifique (GPGPU), la question de l'utilisation des GPU pour exécuter des applications de radio logicielle a été le sujet de plusieurs études. D'un point de vue architectural, les plateformes GPU peuvent aller de la carte graphique d'un ordinateur de bureau classique à des architectures plus fouillées. Par exemple, l'étude [KHC10] propose une architecture matérielle et logicielle qui permet de bénéficier de la puissance de calcul du GPU. Elle propose également une implantation complète du protocole WiMAX en utilisant le GPU.

D'un point de vue applicatif, l'étude de l'utilisation du GPU peut se faire avec deux objectifs distincts :

- concevoir une application spécifique ;
- utiliser le GPU dans le cadre d'un environnement.

Dans le premier cas, on se place dans un cadre d'optimisation classique. Il n'y pas de séparation entre l'application et l'environnement. On trouve une réalisation d'un filtre polyphase dans [HSM+08], d'un récepteur FM dans [SH09], d'un détecteur MIMO dans [WSGC11] et [WEL11], de décodeurs Turbo Code [WSC10] ou encore *Low Density Parity Check* (LDPC) [JCS11].

Dans le second cas, le problème est un peu différent puisqu'on s'intéresse à l'intégration d'opérations dans un système.. On définit des opérations primaires, qui pourront ensuite être utilisées pour définir une application radio. Les opérations sont donc potentiellement plus petites, et surtout, la gestion de la communication entre les différentes opérations pose problème. Les différentes études réalisées sur GNURadio avec l'environnement CUDA [NVI11] semblent donner des résultats limités [arc11]. L'intégration sous SCA [Mil10] semble plus efficace, cependant, les tailles de vecteurs abordées sont très grandes, ce qui ne correspond pas à un cas pratique d'utilisation de la radio logicielle. Une représentation des applications ainsi qu'une étude théorique de l'utilisation des GPU est proposée dans [PZB+11]. L'utilisation des GPU est donc une question ouverte, généralement considérée comme peu adaptée aux applications radio.

3.2.3 Utilisation de FPGA

Les FPGA sont la cible de choix pour les applications de radio logicielle. Ils offrent en effet un bon mélange entre flexibilité (utilisation d'un langage de description) et efficacité (proche de

celle d'un ASIC).

Dans un cadre de reconfiguration statique, c'est-à-dire quand la radio peut être mise hors ligne durant la reconfiguration, la réalisation d'une application de radio logicielle [MSA06][ASM+07][HHP+09] se penche principalement sur la définition des briques de base matérielles et sur la structure de l'application.

Cependant, le manque de dynamisme des solutions FPGA ne permet pas de réaliser efficacement des terminaux multimodes. Le développement des technologies de reconfiguration partielles [Xil10] permet en partie de résoudre ce problème [DPML07][HCS11], mais le temps de reconfiguration limite encore grandement l'efficacité de cette technologie.

3.2.4 Solutions hybrides

Finalement, l'utilisation des 3 technologies (GPP, DSP/GPU, FPGA) présente des avantages et des inconvénients. L'une des solutions envisagées pour compenser les inconvénients et l'utilisation de plateforme hybride, utilisant les trois technologies. Une étude d'une telle plateforme est proposée dans [TR08]. Elle présente une architecture matérielle mixte GPP/DSP/FPGA, ainsi qu'un environnement pour développer sur une telle architecture. Cette étude reste superficielle, il n'y a pas d'automatisation du calcul, juste une répartition et un interfaçage entre le DSP et le FPGA, et un OS temps-réel qui s'exécute sur le GPP.

D'autres études sur des fonctionnement hybrides existent. On citera en particulier [DMLP05], qui s'intéresse à une architecture de calcul hybride DSP/FPGA.

3.2.5 Matériel reconfigurable et adaptabilité

Alors que les implantations de la radio logicielle sur des cibles telles que les GPP ou les FPGA utilisent une dissociation entre la fonction à implanter et l'élément reconfigurable qui permettra de l'implanter, il est également possible d'utiliser une approche ne réalisant pas cette dissociation. Dans ce cas, le matériel utilisé est indissociable de la fonction implantée, ce qui sous-entend généralement une implantation figée.

Si on se concentre sur l'adaptabilité, qui est l'un des critères retenus dans [DZD+05] pour définir la radio flexible, il n'est pas nécessaire d'utiliser une cible logicielle pour avoir cette propriété. Un opérateur FFT, par exemple, peut être adaptable en ajoutant la possibilité de changer le nombre de points. Un modulateur peut être adaptable si la constellation n'est pas figée.

3.2.5.1 Paramétrisation et opérateurs communs

La paramétrisation est une approche d'implantation des terminaux de radio logicielles [1] qui consiste à utiliser un jeu de paramètres pour définir le système [Jon02]. Une application, c'est-à-dire une norme réseau, peut dès lors être représentée comme un ensemble de paramètres, et reconfigurer le système consiste en un changement de paramètres actifs.

La technique des opérateurs communs [APR+11] est un dérivé de cette technique. Elle vise à extraire de différentes normes potentiellement exécutable sur une plateforme des opérateurs "indivisibles", qui représentent les opérations de base d'une norme radio. Ces opérateurs communs peuvent ensuite être implantés sur n'importe quelle cible, logicielle ou matérielle (ASIC). C'est la mise en commun de ces opérateurs qui définit quelle sera la fonctionnalité réellement implantée.

3.2.5.2 ASIC et SOC

L'utilisation d'ASIC adaptables pour implanter un terminal reconfigurable est également une matéhode qui a été envisagée. LeoCore [LNT+09] est un processeur bande de base conçu à partir

1. cette approche est beaucoup plus générale, mais nous nous concentrons ici sur la radio logicielle

d'un NoC, qui embarque tous les éléments d'un système de transmission radio usuel, et qui utilise une modification des chemins entre ces éléments pour se reconfigurer.

Il existe un autre système à base de NoC, Magali [NKM+09]. Ce système utilise un coeur GPP et des coeurs DSP pour la partie flexible ainsi que des accélérateurs matériels figés pour certaines opérations comme l'H-ARQ, le codage, ou encore la modulation. La propriété d'adaptabilité est suffisante pour permettre l'utilisation des ces opérateurs dans plusieurs normes, des DSP sont disponibles pour les cas où le matériel ne suffit pas.

3.2.6 Conclusion sur les plateformes

L'intérêt croissant pour la recherche sur les techniques de radio cognitive (et donc flexible) a conduit à la multiplication des techniques employées pour permettre l'exécution des normes, ou la reconfiguration.

Concernant les plateformes pour l'exécution de la radio reconfigurable, le nombre de possibilités est presque infini. On trouve des plateformes à base de GPP uniquement (ordinateurs de bureau), ou de DSP, ou encore de FPGA. On trouve également des mélanges de toutes ces solutions. Des approches pour le GPU commencent également à voir le jour dans le cadre d'applications spécifiques.

D'autres approches à base de NoC existent. Ces techniques permettent d'utiliser des ASIC adaptables, plus efficaces que les solutions logicielles comme les DSP ou les FPGA, et de connecter ces ASIC à l'aide d'un NoC. Ceci permet de modifier les séquences d'actions à effectuer.

On se retrouve donc face à une multitude de techniques possibles, avec pour chacune, une méthode de gestion différente. Cette multiplication des solutions appelle donc une unification du contrôle, qui peut être fournie des environnements de reconfiguration.

3.3 Environnements

3.3.1 Objectifs

L'implantation d'une norme de radio flexible est donc actuellement réalisable sur de nombreuses plateformes. Cependant, ces plateformes ne servent qu'à exécuter l'application qui est déployée. Le contrôle de l'exécution de cette application, ainsi que sa reconfiguration, doivent également être prises en compte.

Les environnements de radio logicielle sont développés pour satisfaire ces contraintes. Ils servent d'interface entre la représentation de l'application et son exécution réelle, et prennent donc en compte les spécificités de la plateforme. Ils offrent également une interface de reconfiguration aux couches supérieures.

Nous présentons ici plusieurs environnements de radio logicielle, avec chacun leurs spécificités.

3.3.2 Exécution hétérogène

Au-delà de la radio logicielle, beaucoup d'études se sont penchés sur les possibilités d'exécution utilisant des éléments hétérogènes. On s'intéresse ici à ces possibilités, qui peuvent être utilisées dans le cadre de la radio logicielle. Lorsqu'on intègre des éléments de calcul dans un système, deux points sont à prendre en compte :

- le contrôle de l'élément, qui permet de définir ce qu'il va faire, de le synchroniser, et de modifier son fonctionnement le cas échéant ;
- les communications qui permettent d'amener à l'élément les données sur lequel il va devoir travailler.

3.3.2.1 Accélérateurs matériels

Dans le cas des accélérateurs matériels ou des coprocesseurs, le contrôle dépend du type d'accélérateurs. Si celui-ci est intégré au processeur, comme dans le cas des coprocesseurs du MIPS [MIP92] ou du ARM [ARM05], alors le contrôle se fait en utilisant des instructions spécifiques dans un programme.

Si on utilise un accélérateur matériel connecté à un bus, avec des registres, on utilise généralement une interface périphérique telle que définie dans la norme *Portable Operating System Interface for Unix* (POSIX) [fIT03].

Le transfert de données entre le monde logiciel (mémoire vive) et les coprocesseurs est un problème plus complexe [HPA01]. L'utilisation de FIFO est la méthode classique, qu'elles soient logicielles ou matérielles.

3.3.2.2 Intégration des FPGA

On distingue ici deux types d'utilisation du FPGA. Dans un premier cas, le FPGA est utilisé comme un ASIC, mais peut être modifié si besoin est. C'est le cas classique d'utilisation, dans lequel le fait que ce soit un FPGA qui est utilisé est secondaire. Une fois l'application terminée, elle n'est jamais modifiée. Dans ce cas, le contrôle s'apparente à un contrôle d'accélérateur matériel.

Dans le second cas, le FPGA peut être reconfiguré partiellement durant l'exécution. L'intégration des éléments potentiellement reconfigurables est alors problématique. On se heurte en fait à un problème majeur d'ordonnancement, qui peut se résumer ainsi : quand et quoi doit-on reconfigurer ?

Une solution possible est de voir le FPGA comme un processeur [ANJ+04]. Dans ce contexte, les éléments à reconfigurer deviennent l'équivalent des programmes logiciels. Ils sont décrits avec un langage (VHDL, SystemC, ...) puis synthétisés et placés/routés pour la cible (au lieu d'être compilé pour un processeur). Il convient dès lors de les intégrer comme des tâches à part entière dans le système, avec une partie fixe qui remplace les structures de processus des systèmes d'exploitation classiques. On trouve dans [ANJ+04] les extensions nécessaires du système d'exploitation. Cette approche est reprise dans [MHV+09][LP08][WCW+09].

3.3.2.3 Exécution hétérogène GPP/DSP

L'intégration des DSP dans un système utilise le domaine plus général des systèmes hétérogènes. Utiliser des processeurs différents dans une même plateforme peut se révéler utile, puisque chaque processeur a ses inconvénients qui peuvent être contrebalancés par les avantages des autres. Cependant, le développement d'une application devient rapidement complexe. Si on passe sur les difficultés dans le choix du processeur à utiliser pour chaque tâche, il reste à gérer l'intégration de processeurs ayant chacun leur propre interface d'accès à la mémoire, leur propre jeu d'instruction, ou pire encore, leur propre "boutisme" (endianness).

On distingue alors deux approches pour gérer cette intégration. Il est possible d'utiliser une approche distribuée, basée sur un système d'exploitation (OS) global [Gue10], qui intègre une représentation de la plateforme, et qui définit les méthodes à utiliser pour chaque type de processeur. L'approche proposée dans [Gue10] s'appuie sur un flot de génération.

Il est également possible d'utiliser une approche centralisée, dans laquelle un GPP est utilisé comme contrôleur, et donne les ordres et les données aux autres processeurs. La spécification OpenCL [ope10] est un exemple de cette approche. L'environnement OpenCL définit une architecture basée sur un hôte qui exécute toutes les opérations de contrôle, et sur des éléments de calcul qui sont commandés par l'hôte. Les échanges entre l'hôte et les éléments sont basés sur des FIFO. La définition d'une tâche implique l'écriture d'un noyau (*kernel*) qui définit le calcul. L'application OpenCL crée des contextes d'exécution lors de son initialisation. Ces contextes sont

en fait des associations entre des queues de contrôle, et un ou plusieurs éléments de calcul. Il y a ensuite une compilation des *kernel* pour les contextes sur lesquels ils seront exécutés. Ainsi, si un *kernel* peut s'exécuter sur un DSP et sur un GPP, il sera compilé pour ces deux entités. Lors de l'exécution, le programme compilé associé au contexte est employé.

L'architecture OpenCL peut également être utilisée pour les calculs sur GPU. En revanche, à cause des spécificités du processeur graphique, il est difficile de mettre en oeuvre l'approche distribuée.

3.3.3 Environnements complets

3.3.3.1 SCA

Le premier (et principal) environnement complet de radio logicielle est la norme proposée par le Joint Tactical Radio System (JTRS) Joint Program Office (JPO). Cette norme baptisée SCA [sca01][SMM05][JXJQ09] est issue du domaine militaire. Elle est maintenant maintenue par le Wireless Innovation Forum. La norme spécifie une interface pour définir une *waveform* (*cf.*3.1.2). L'implantation effective de cette *waveform* est ensuite libre. On notera que SCA n'est pas un environnement à proprement parler, mais plutôt une spécification. Il y a donc une certaine liberté dans l'implantation, et donc une variation possible entre différentes implantations.

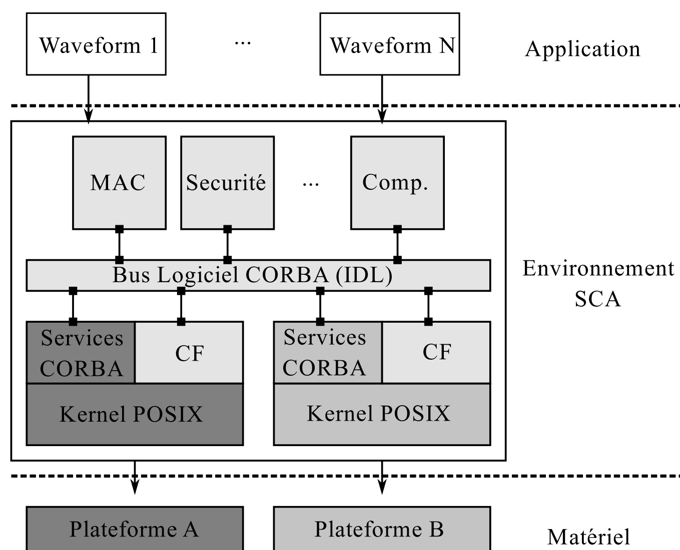


Figure 3.2 – Architecture de l'environnement SCA

L'architecture de SCA est présentée sur la figure 3.2. Tout s'articule autour du *Core Framework* (CF), l'environnement central, représenté en gris clair. Le CF définit toutes les interfaces pour les différents éléments de SCA. La norme utilise une approche orientée objet, et donne donc les définitions à différents niveaux. Ainsi, par exemple, l'élément *Resource* est une interface pour la configuration des composants logiciels. Cet élément sert en fait de base à une multitude d'autres éléments, comme par exemple les *NetworkResource*, qui gère des entités réseaux. D'autres types peuvent également héritée de cet élément.

Les différents types de matériels sont supportés par l'élément *Device*. Cet élément se décline en plusieurs sous-types, selon le type de matériel considéré. On trouve principalement le type *LoadableDevice* qui représente un périphérique sur lequel on peut charger ou décharger du logiciel. Les FPGA et les DSP rentrent dans cette catégorie de *Device*. Tout élément matériel doit être encapsulé dans un élément logiciel de type *Device*.

Toute la norme SCA s'articule autour de la norme *Common Object Request Broker Architecture* (CORBA) [OMG08], ceci afin de permettre la communication entre les différents composants. Il existe une implantation libre de SCA, baptisée OSSIE [Tec], basée sur l'implantation OmniORB de CORBA.

L'environnement repose sur un système d'exploitation existant, qui doit respecter la norme POSIX. Ce système est représenté en gris foncé sur la figure. Il est possible d'avoir deux plateformes différentes, grâce à l'utilisation du bus logiciel CORBA.

De notre point de vue, SCA et ses implantations souffrent de deux problèmes principaux :

- le manque de dynamisme de la reconfiguration [GMSG08]. La norme a été conçue pour permettre une configuration efficace, et pour abstraire les spécificités du matériel. Cependant, elle ne permet que difficilement le passage dynamiquement d'une application SCA à une autre. Diverses études s'attachent à résoudre ce problème. Dans [CDPR10], plusieurs approches sont expérimentées, basées sur un aiguilleur de données [Cor08]. Les résultats montrent que la solution fonctionne bien, mais elle reste une rustine pour ajouter une fonctionnalité absente de la conception.
- la forte connotation logicielle de la norme. Il n'est que peu question d'utiliser des accélérateurs matériels ou autres matériels dédiés dans la norme SCA. Une étude de l'intégration de matériel spécialisé dans SCA est proposée dans [KB05].

La littérature montre également que SCA est largement adopté dans le domaine. On trouve par example [DBRC05] ou [BZZ08], qui présente des exemples d'utilisation de la norme.

3.3.3.2 RMA

Reconfiguration Management Architecture (RMA) [MGT01] est une architecture de gestion de la reconfiguration qui s'attaque principalement au problème de la topologie d'un réseau qui intègre des terminaux reconfigurables. C'est le plus haut niveau de la hiérarchie de reconfiguration présentée en 3.1.1.2.

RMA est une architecture scindée en deux parties. La première partie, Configuration Control Part (CCP), est ancrée dans le réseau. Elle gère l'authentification du terminal, la connexion à un dépôt de logiciel (qui enregistre les configurations), et les interactions avec le reste du monde. Cette partie gère entre autres la récupération du logiciel, la virtualisation de la configuration, et le monitoring.

La seconde partie est centrée sur le terminal. On distingue la partie configuration de la partie radio. La partie configuration (Configuration Management Part, CMP), sert d'interface avec la CCP, et implémente le gestionnaire principal de configuration, qui interagit avec tous les autres éléments. L'interface entre la CMP, et la partie radio (Radio Module Part, RMP) se fait au travers des contrôleurs de reconfiguration (Reconfiguration Module Controller, RMC). La configuration du terminal est représentée dans RMA par un *tag-file*, qui est un fichier XML.

RMA est une proposition relativement ancienne, qui est aujourd'hui supplantée par les autres solutions, en particulier par SCA.

3.3.3.3 GNURadio

GNURadio [gnub] est un environnement de radio logicielle défini afin de permettre l'exécution et la configuration d'une application radio couche basse sur un GPP. Il est basé sur une vision "bloc" de la *waveform*, avec une modélisation *Synchronous Data Flow* de l'application. C'est un environnement abouti, encore en développement, et très actif.

La figure 3.3 est une représentation de l'architecture générale de GNURadio. Il y a une séparation claire entre la partie calculatoire de l'application et la partie contrôle. GNURadio définit trois grands types d'objets :

- les blocs, qui servent à représenter les opérations ;

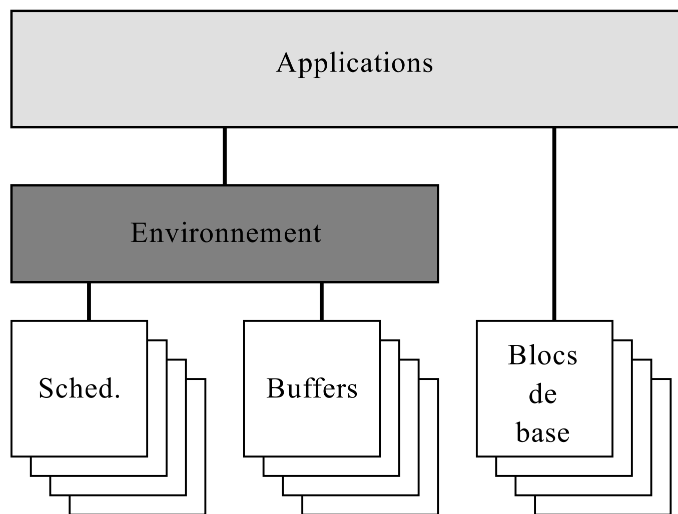


FIGURE 3.3 – Architecture générale de GNURadio

– les *buffers* qui permettent d'assurer les communications entre les différents blocs ;
– l'ordonnanceur (*scheduler*) qui spécifie comment vont s'exécuter les blocs.

GNURadio souffre des mêmes problèmes que SCA : le projet est conçu pour une application logicielle sur GPP uniquement, et il souffre d'un manque de dynamisme dans la reconfiguration. Cependant, il ne s'appuie pas sur CORBA, et limite donc le surcoût de communication, et les dépendances avec le logiciel. GNURadio est un *framework* plus qu'un environnement de reconfiguration : il permet d'écrire rapidement une application radio en utilisant des blocs déjà connus, mais la gestion de la reconfiguration est superficielle. L'accent est mis sur l'exécution d'une application.

3.3.3.4 P-HAL/Aloe

P-HAL [RGMF05] et son successeur Aloe [GMSG08] sont deux environnements similaires qui proposent une interface d'abstraction de la plateforme, afin de permettre à une application de s'exécuter sur plusieurs plateformes utilisant P-HAL.

P-HAL est basé sur une architecture qui se découpe en 3 éléments, comme représenté sur la figure 3.4 :

– une représentation de l'application (du type *waveform*) et une traduction de cette représentation en tâches compréhensibles par le système ;
– une couche d'"intergiciel", qui sert d'interface entre l'application et les exécutants ;
– les éléments d'exécution, qui se séparent en une couche logicielle, et une couche matérielle.

L'objectif de P-HAL est comparable à celui de SCA. Cependant, il n'y a pas d'utilisation de CORBA, ce qui réduit les dépendances de l'application. Il devient envisageable de déployer P-HAL sur une architecture ne supportant pas CORBA. Le bus logiciel est ainsi remplacé par la couche de *middleware* qui permet de masquer les spécificités du matériel. Chaque plateforme est ainsi vue comme une plateforme virtuelle P-HAL. La couche d'intergiciel permet d'ajouter encore à l'abstraction afin de transformer les plateformes multiples en une plateforme unique virtuelle.

Aloe [GMSG08][GMBG10][alo] est une évolution de P-HAL. Cette évolution est conçue pour permettre la radio cognitive, et ajoute en particulier une gestion temps réel des ressources de calcul, qui passe par un *monitoring* précis et efficace du système. Cet ajout est réalisé en conservant les développements précédents de P-HAL, mais en ajoutant des services de contrôle du système [GMSG08].

Á l'inverse de GNURadio, P-HAL/Aloe est un gestionnaire de reconfiguration, l'accent est

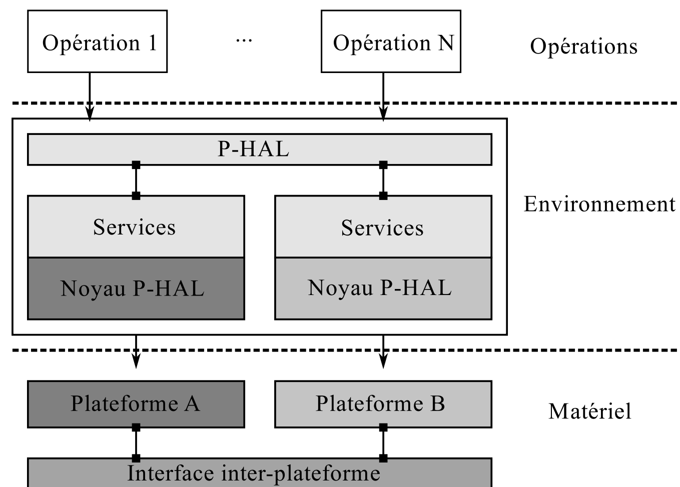


FIGURE 3.4 – Architecture générale de P-HAL/Aloe

mis sur le contrôle de la plateforme, et sur l'interface offerte aux autres éléments d'un système de communication.

3.3.3.5 MVR : Machine Virtuelle Radio

Une autre approche, proposée en 2000 par Gudaitis et Mitola (d'après [Abd10], le document n'étant pas disponible) se base sur l'utilisation d'une machine virtuelle radio. L'objectif de cette méthode, baptisée *Radio Virtual Machine* (RVM) est de définir une machine virtuelle du même type que celle de Java, c'est-à-dire en utilisant une approche à composants et en se basant sur du *bytecode* exécutable par toute plateforme supportant la machine virtuelle.

Une utilisation de cette approche est proposée dans [ARFD09] et [ARFM10]. Une implantation d'une RVM basée sur Lua est proposée, ainsi qu'un cas pratique d'utilisation sur la plateforme Magali.

Cette approche basée sur une machine virtuelle présente de nombreux avantages. Elle permet en particulier une vraie abstraction de l'application radio. Elle permet également une reconfiguration simplifiée d'un terminal, puisqu'il suffit de transférer le *bytecode* correspondant à une norme puis à l'exécuter. Cependant, malgré le réel intérêt que présente cette approche pour la flexibilité, certains points ne sont pas complètement satisfaisants, en particulier le surcoût associé à la machine virtuelle, et la difficulté de gestion d'un composant matériel. Ces travaux restent malgré tout les plus proches de ce que nous proposons dans cette thèse.

3.3.3.6 Surfer

Le dernier environnement présenté ici est Surfer [DDL10][DLD11]. C'est un environnement relativement récent, présenté pour la première fois en 2010. Son architecture est classique.

L'objectif principal de *Surfer* est de minimiser au maximum le surcoût lié au contrôle de l'exécution des blocs, tout en optimisant au maximum le calcul des données. Il utilise pour cela un schéma d'ordonnancement distribué basé sur des seuils dans les *buffers* d'entrée/sortie. Les surcoûts, qui sont généralement centralisés dans une tâche de contrôle, sont ainsi répartis entre les différents blocs. Un autre objectif principal de *Surfer* réside dans le peu de dépendances avec d'autres éléments. L'environnement est basé sur une couche d'interface avec le système d'exploitation, qui doit être adaptée en cas de changement de système. On note ici la différence avec SCA, qui requiert l'utilisation d'un système POSIX. C'est la même approche que dans P-HAL. La

collecte de statistiques en interne de chaque bloc (dans le cadre de la distribution du contrôle) fait également partie des objectifs.

A partir de cet environnement, une approche de la reconfiguration dynamique est proposée dans [DLD11]. L'objectif est ici de permettre le délestage des processeurs afin d'offrir une qualité de service donnée à certaines communications. Afin d'y parvenir, plusieurs implantations de la même fonctionnalité sont proposées dans *Surfer*. Un superviseur est implanté, qui se sert des capacités de collecte de statistiques des blocs afin de détecter s'il y a besoin de changer de moyen de calcul. Par exemple, une application s'exécute sur une plateforme qui contient un GPP et un GPU. Le GPP étant suffisant pour exécuter cette application, il n'y a pas de raisons de lancer deux périphériques. Cependant, si une application externe est lancée sur ce GPP, il risque de ne plus suffire. Le superviseur délestera alors le GPP en lançant le calcul sur le GPU. L'application reviendra sur le GPP une fois celui-ci allégé. Sans prendre en compte le superviseur, la méthode de reconfiguration dynamique est assez proche de [CDPR10] sur SCA.

3.3.4 Conclusion

Il existe beaucoup d'études sur des environnements de radio logicielle. Les quatre principaux présentés ici, SCA, GNURadio, P-HAL/Aloe et Surfer présentent tous des intérêts différents. SCA est principalement intéressant parce qu'il est normatif, et extrêmement répandu. Cependant, il définit le *framework*, sans spécifier l'environnement d'exécution proprement dit. De plus, l'utilisation de CORBA est pénalisante, les problèmes de latence dans les interfaces étant connus. Cette dépendance a d'ailleurs été supprimée de la dernière version de la norme [SCA11], de même que la dépendance à POSIX. Elle reste cependant présente dans les implantations. Par contre, SCA fournit un bon support des matériels hétérogènes. GNURadio, bien qu'intéressant dans sa conception, et extrêmement riche en termes de développement, d'évolutions, et dans sa bibliothèque d'éléments de calcul, est exclusivement développé pour GPP. Le support d'autres éléments d'exécution n'est pas prévu, hormis les DSP (BeagleBoard). Il est également difficile à prendre en main. La réalisation d'une application simple est aisée, mais la moindre modification demande de connaître tous les détails, et la marche à franchir est haute. Finalement, la reconfiguration n'est pas réellement abordée dans l'environnement, ce qui le rend insuffisant pour nos besoins. Il reste cependant un environnement intéressant en particulier pour le prototypage et la recherche.

P-HAL/Aloe est actuellement l'environnement le plus proche de nos attentes. Il est conçu pour unifier des plateformes hétérogènes sous une même architecture. La possibilité de collecter dynamiquement des statistiques offrent des perspectives intéressantes. Il est également disponible sous licence libre. Cependant, la reconfiguration des *waveforms* n'est pas dynamique, et le support des *waveforms* multiples n'est pas disponible.

Surfer semble intéressant, cependant, il n'est pas disponible sous licence libre (et ne le sera pas prochainement d'après le concepteur principal), et malgré l'intérêt qu'il présente, il en reste à ses débuts.

Finalement, tous ces environnements souffrent de deux défauts majeurs :

- la gestion de l'adaptabilité n'est pas étudiée ;
- le support du matériel spécifique n'est pas abordé.

Malgré leur intérêt certain, ils ne répondent pas complètement à la problématique de cette thèse.

3.4 Conclusion du chapitre

La radio flexible est un domaine très large. Il englobe toutes les techniques d'implantation de terminaux radios qui ne sont pas figées, en particulier des solutions adaptables par modification de

paramètres numériques, et des solutions reconfigurables par changement de la structure même de l'application.

Plusieurs méthodes existent pour apporter des éléments de réponse à cette problématique. La radio logicielle utilise des cibles génériques, comme les processeurs (GPP, DSP) ou les FPGA. L'utilisation des GPU a également été envisagée, mais les résultats ne sont pas à la hauteur des attentes. Certaines méthodes utilisent une conception particulière des éléments matériels dédiés, qui peuvent exécuter différentes opérations selon l'agencement utilisé. Finalement, il est également possible de multiplier les composants dédiés, autant qu'il y a de normes à implanter (méthode "Velcro" [Alh10]).

La multiplication des implantations nuit au développement des algorithmes qui tirent parti de la flexibilité. La nécessité de prendre en compte toutes les implantations possibles est un frein à leur déploiement. L'utilisation des environnements de radio flexible permet de contrebalancer en partie ce problème, en offrant une interface générique aux couches supérieures. Cependant, les environnements actuels présentent des lacunes qui empêchent d'atteindre ce but d'abstraction de la méthode utilisée pour la flexibilité :

- les environnements sont développés la plupart du temps pour une solution précise, et non pour prendre en compte toutes les implantations matérielles ;
- les environnements existants ne prennent pas systématiquement en compte la reconfiguration dynamique de la chaîne de communication ;
- l'intégration des couches hautes, et donc des algorithmes de la radio flexible et cognitive, n'est pas faite dans l'environnement ;
- la gestion du matériel dédiée n'est quasiment pas abordée.

Chapitre 4

Intégration du GPU dans la radio logicielle

Nous abordons dans ce chapitre l'utilisation du GPU dans le cadre de la radio logicielle. Comme évoqué dans le chapitre 2, le GPGPU est un domaine qui permet d'obtenir une puissance de calcul énorme avec un coût financier limité. Cependant, appliqué au domaine de la radio logicielle, les études précédentes montrent des résultats décevants. Nous cherchons donc à répondre à la question suivante dans ce chapitre : comment peut-on tirer profit du GPGPU dans le cadre d'un environnement de radio logicielle ? On s'intéresse dans cette étude aux trois points suivants :

- la conception d'une opération SDR sur le GPU ;
- l'ordonnancement ;
- les transferts de données entre les opérations.

4.1 Introduction : le GPGPU **36**
- 4.1.1 Origine 36
- 4.1.2 Principe 36
- 4.1.3 Contraintes et objectif 38

4.2 Utilisation optimisée **39**
- 4.2.1 Opérations basiques 39
- 4.2.2 Opérations complexes : première approche 40
- 4.2.3 Proposition : maximisation de l'utilisation 41

4.3 Ordonnancement **43**
- 4.3.1 Efficacité 43
- 4.3.2 Latence 43
- 4.3.3 Mémoire 45

4.4 Intégration distribuée **45**
- 4.4.1 Présentation 45
- 4.4.2 Communications 45

4.5 Intégration centralisée **47**
- 4.5.1 Présentation 47
- 4.5.2 Communications 48
- 4.5.3 En pratique 48

4.6 Résultats **49**
- 4.6.1 Synthèse des contributions 49
- 4.6.2 Opérations implantées 49
- 4.6.3 Protocole d'expérimentation 52
- 4.6.4 Résultats des opérations unitaires 53
- 4.6.5 Résultats pour des applications multitâches 56

4.7 Perspectives et conclusion **57**
- 4.7.1 Discussion : portabilité vers l'embarqué 57
- 4.7.2 Conclusion 58

4.1 Introduction : le GPGPU

4.1.1 Origine

Les évolutions successives des API graphiques, de DirectX en particulier, ont poussé vers une généricité de plus en plus grande des processeurs graphiques (GPU). D'unités extrêmement spécialisées, ces processeurs sont devenus de plus en plus génériques afin de satisfaire les contraintes de plus en plus larges des opérations de traitement graphique. Logiquement, ce gain en généricité a permis une ouverture de l'architecture traditionnellement limitée aux API graphiques. Cette ouverture a permis l'émergence du GPGPU.

Le GPGPU permet d'utiliser le processeur graphique pour effectuer des calculs "classiques", en utilisant un environnement adapté. Le GPU ne peut pas être utilisé dans toutes les applications du fait de son architecture SIMD qui ne permet aucun contrôle dépendant des données dans un programme. Cependant, bon nombre de traitement de données sont compatibles avec cette architecture. Le GPU permet alors un gain en performance non négligeable.

Ces environnements étant extrêmement dépendants du matériel utilisé, chaque constructeur fournit son propre *framework* (CUDA pour NVidia, Stream pour ATI). Afin d'unifier ces environnements, et de les élargir à d'autres architectures que les GPU, le Khronos Group [1], qui avait déjà normalisé l'OpenGL, propose l'environnement de programmation Open Computing Language, OpenCL [ope10]. Cet environnement permet d'utiliser des unités de calcul hétérogènes, gérées par un hôte. Même si on s'intéressera principalement à l'OpenCL pour les GPU, la norme prévoit la possibilité d'utiliser des DSP ou des GPP, en conservant le même modèle de programmation.

Bien que les spécifications soient déjà disponibles, il n'existe pas encore à notre connaissance d'implantation complète d'OpenCL. Chaque constructeur propose sa propre implantation, qui se trouve être une surcouche de l'environnement propriétaire. Ceci nuit à l'objectif final d'OpenCL, dans le sens où il n'est pas possible d'utiliser réellement des éléments hétérogènes. Un système avec un GPU ATI et un GPU NVidia ne pourra pas utiliser ces deux GPU dans une même application.

4.1.2 Principe

Afin de bien poser le cadre de l'étude, nous présentons succinctement OpenCL, en particulier l'architecture d'une plateforme OpenCL et sa signification sur un GPU, et le format d'une application OpenCL.

4.1.2.1 Architecture d'une plateforme OpenCL

Une plateforme OpenCL est une plateforme générique utilisée pour représenter tous les types d'unités de calcul utilisables avec la norme. La figure 4.1 est une représentation de cette plateforme. Les traits pleins représentent les données, les pointillés montrent le contrôle.

L'architecture est divisée en deux parties, l'hôte et les *computing devices*. L'hôte est un GPP, qui est utilisé pour le contrôle de l'application. Il se charge :
- de l'initialisation de l'environnement OpenCL, avec l'allocation des structures ;
- de l'allocation des zones mémoires requises pour l'application, sur la mémoire hôte et sur la mémoire des *devices* ;
- du lancement des différentes applications sur les *devices*.

Un *device* est un "périphérique de calcul" asservi à l'hôte. Il peut être un GPP, un DSP, ou encore un GPU. Un *device* est subdivisé en *computing units* (unités de calculs), qui sont elles même séparées en *processing elements* (élément de traitement).

1. On trouve parmi les membres du Khronos Group les principaux acteurs du domaine : AMD/ATI, NVidia, Intel, ARM, Texas Instruments et STMicroelectronics en particulier

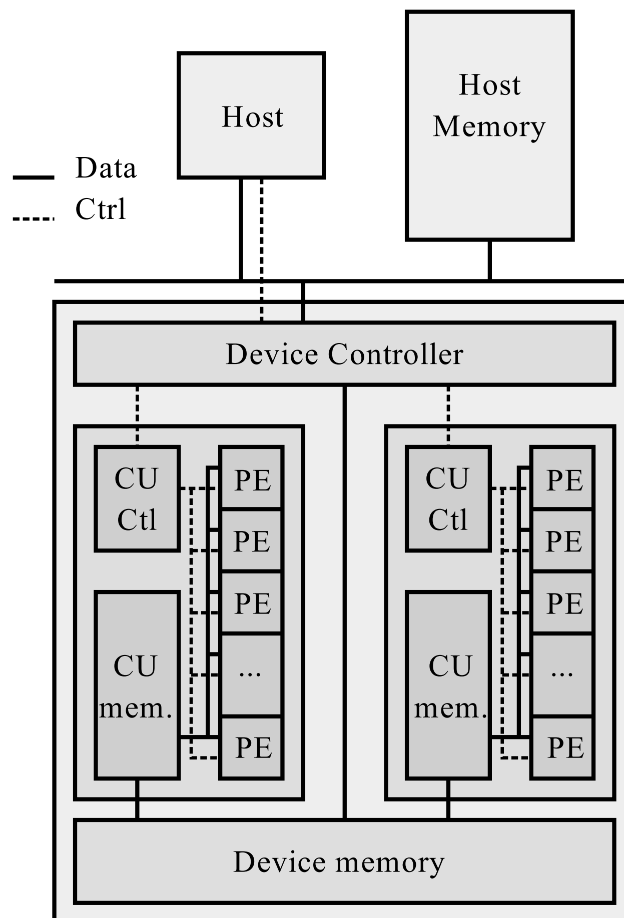


FIGURE 4.1 – Représentation d'une plateforme OpenCL

Dans notre étude, nous nous intéressons uniquement au GPU, et plus particulièrement aux GPU NVidia, qui ont servi pour nos expérimentations. Un GPU NVidia est basé sur la même architecture. Les PE sont appelés *cores*, les CU sont appelés *MultiProcessors*, et les *devices* sont les GPU complets. Les PE sont des coeurs de calculs dégénérés, dans lesquels tous les éléments de contrôle de l'application ont été enlevés. Un PE ne peut pas gérer son propre programme. Un programme s'exécute au niveau de l'unité, qui contrôle elle-même l'instruction qui sera exécutée. Tous les PE exécutent donc la même instruction au même moment, mais sur plusieurs données différentes (SIMD). Les GPUs sont particulièrement bien adaptés pour le calcul vectoriel.

4.1.2.2 Format d'une application

L'écriture d'une application OpenCL passe par deux étapes : l'écriture du *kernel*, qui spécifie le calcul qui sera réalisé, et l'écriture de l'application hôte, qui permet d'utiliser l'environnement OpenCL.

La figure 4.2 représente l'architecture générale d'une application hôte OpenCL. On effectue un calcul sur un vecteur. Un *kernel* représente l'application unitaire qui s'applique sur chaque élément du vecteur. Il peut être fourni par l'utilisateur, ou se baser sur une bibliothèque. Les *cl_devices* servent à représenter le périphérique physique. Les pilotes qui seront utilisés pour ces périphériques sont masqués à l'utilisateur, seul l'environnement d'exécution (*runtime*) d'OpenCL a connaissance de ces spécificités.

Le contexte est l'élément d'exécution de base pour l'utilisateur. Il regroupe un ou plusieurs périphériques sous une même structure, ainsi qu'une FIFO de contrôle. Les communications entre le

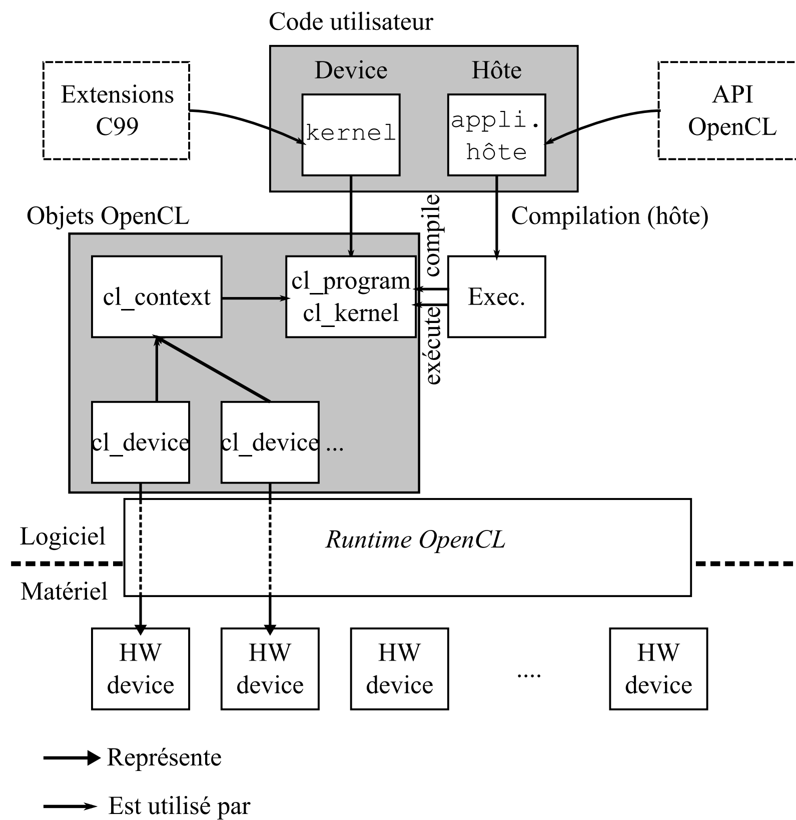


FIGURE 4.2 – Architecture d'une application OpenCL

programme OpenCL et l'environnement ont lieu exclusivement en utilisant ces FIFO. Les *kernels* sont compilés en programmes durant l'initialisation. Une compilation a lieu pour chaque contexte sur lequel le programme s'exécutera. Ces *kernels* sont ensuite exécutés pour chaque élément du vecteur par les PE, chaque instance du *kernel* est appelée *thread*.

4.1.3 Contraintes et objectif

L'objectif de ce chapitre est d'étudier les différentes possibilités d'implantation de la radio logicielle pour un processeur graphique. La radio logicielle a un haut niveau de généricité. L'utilisation du GPU pour ce type d'application est donc plus complexe que pour une application spécifique. Une application spécifique peut se concevoir en fonction du GPU pour avoir une exécution optimisée. Dans le cas de la radio logicielle, l'optimisation devient plus complexe.

Une application de radio logicielle est une succession d'opérations appliquée à des échantillons. Ces échantillons proviennent du convertisseur analogique numérique branché sur la sortie de l'antenne. Les échantillons sont généralement organisés en trame. Par exemple, l'utilisation de l'OFDM comme technique de modulation impose l'utilisation de vecteurs d'échantillons. Les opérations sont appliquées sur ces vecteurs, l'exemple typique restant la transformée de Fourier. Il est toujours possible d'extraire une unité de base sur laquelle s'applique l'opération, cette unité pouvant aller de l'échantillon unique (une valeur) à un paquet complet en sortie de la chaîne de communication.

On utilisera dans ce chapitre GNURadio comme environnement de radio logicielle, mais l'étude est facilement transposable à d'autres environnements.

4.2 Méthode pour une utilisation optimisée

Dans cette section, nous présentons trois méthodes pour utiliser le GPU dans un environnement de radio logicielle.

4.2.1 Opérations basiques

On parle ici d'opérations basiques pour définir les opérations qui sont effectuées échantillon par échantillon, sans notion de bloc. Ce sont des opérations généralement simples, utilisés pour les transmissions non organisées, c'est-à-dire sans séparation en blocs ou en trames. Le passage d'une opération à une autre est géré, de manière conceptuelle, en utilisant des FIFO [gnub]. Une opération, dans ce contexte, est donc un bloc logiciel qui lit une FIFO en entrée, effectue son opération sur les données lues, puis écrit le résultat sur une FIFO en sortie. Cette FIFO sera elle-même lue par l'opération suivante de l'application de radio logicielle. Ceci est représenté sur la figure 4.3.

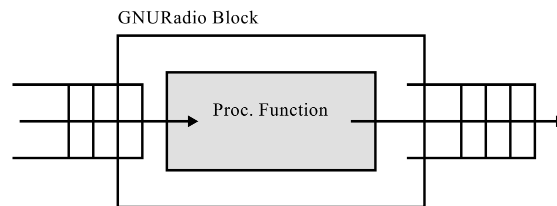


FIGURE 4.3 – Opération de radio logicielle

L'amplification est un bon exemple d'opération basique, dont l'implantation est présentée dans l'algorithme 1. L'objectif de l'amplification est de multiplier l'échantillon par un facteur *amp* connu. On remarque que même si la transmission n'est pas organisée en bloc, l'opération s'effectue sur un tableau, et utilise donc une boucle. Ceci est dû au fonctionnement des environnements de radio logicielle qui ne transfèrent pas les données échantillon par échantillon pour des raisons évidentes de performance. SCA utilise une taille d'échantillon, et travaille donc sur des tailles constantes, GNURadio utilise des FIFO, et travaille donc sur les données disponibles.

Algorithm 1 Opération d'amplification

```
for all sample in input do
    output[sample] ← input[sample] * amp
end for
```

Utiliser le GPU pour la radio logicielle dans ces conditions est relativement simple. On trouve dans [Mil10] une étude complète de l'intégration du GPU dans SCA, pour 5 opérations basiques dans le cas d'un traitement non organisé : une démodulation AM, une démodulation FM, un amplificateur, une décimation et une interpolation. L'approche utilisée est alors logique. On définit comme *kernel* pour le GPU le corps de la boucle. Le CPU n'est plus utilisé pour le calcul, mais uniquement pour transférer les données au GPU et lancer le *kernel* sur l'ensemble des échantillons à l'entrée. La figure 4.4 représente la modification dans l'opération. Le code correspondant pour l'amplification, sans les initialisations, est donné dans l'algorithme 2.

L'approche classique [Mil10] donne de bons résultats dans une transmission non organisée, à condition de travailler sur suffisamment d'échantillons en même temps pour maximiser l'utilisation du GPU. D'après [Mil10], le nombre d'échantillons minimum pour observer un gain en temps de calcul pour le GPU par rapport au CPU est de 1024.

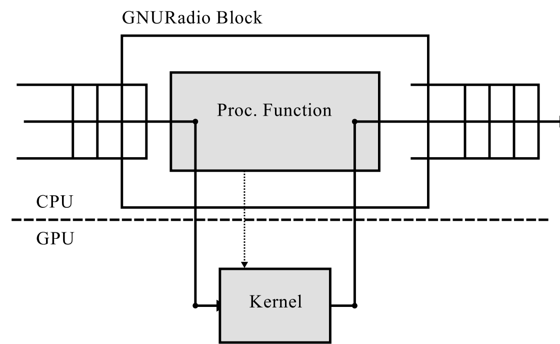


FIGURE 4.4 – Opération de radio logicielle sur GPU : approche classique

Algorithm 2 Opération d'amplification sur le GPU

```
procedure EXECUTE                                                        ▹ Code hôte
    Copier N échantillons vers le GPU
    Définir les arguments in, out, amp
    Exécuter AMPLIFICATION_KERNEL(in, out, amp) sur les N échantillons
    Copier les N échantillons résultats depuis le GPU
end procedure
procedure AMPLIFICATION_KERNEL(input, output, amp)                       ▹ Code GPU
    id ← id du thread
    output[id] = input[id] * amp ;
end procedure
```

4.2.2 Opérations complexes : première approche

On appelle opérations complexes les opérations pour lesquelles le traitement échantillon par échantillon ne fonctionne pas. Ces opérations s'effectuent donc sur des vecteurs d'échantillons. Un bon exemple d'opération complexe est la FFT qui est utilisée entre autres pour la modulation OFDM. Une FFT prend en entrée un vecteur et renvoie un vecteur de même taille. La FFT se calcule selon la fonction suivante, avec X le vecteur de sortie, x le vecteur d'entrée, et N la taille du vecteur :

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2j\pi k \frac{n}{N}}, k = 0 \dots N-1$$

Pour effectuer efficacement une opération complexe sur le GPU, il convient de chercher une implantation SIMD. Si on reprend l'exemple de la FFT, en utilisant l'algorithme radix-2, on obtient une implantation efficace sur GPU [bea]. Le *kernel* est alors relativement simple, puisqu'il implante un papillon de FFT entre deux éléments. On lance le *kernel* $\log_2(N)$ fois, sur $\frac{N}{2}$ éléments.

L'approche utilisée ici ressemble donc à celle utilisée pour les opérations basiques, même si l'unité de base n'est plus réellement les échantillons. Cependant, cette approche présente deux inconvénients majeurs. Tout d'abord, d'après [bea] et nos propres expérimentations, on n'observe un gain en temps de calcul pour le GPU par rapport au CPU que pour N très grand ($N > 2^{18}$ d'après [bea], $N > 2^{15}$ d'après nos expérimentations). Il est très peu probable d'avoir besoin d'une telle taille de vecteurs dans une application radio, ce qui rend l'utilisation du GPU avec une telle approche inutile.

Lorsque l'opération est réalisée sur un "petit" vecteur, le CPU est efficace et le GPU n'apporte donc pas de gain. L'utilisation de l'environnement génère un surcoût, et les cœurs de calcul du GPU sont moins efficaces que le CPU. Ceci explique en partie la grande taille de vecteur nécessaire. De plus, le coût de transfert des données vers le GPU impose qu'il y ait un vrai gain sur le

calcul proprement dit. La bande passante entre le CPU et le GPU tourne autour de 8 Go/s théorique (GPU connecté sur un bus PCI-Express) sans cache. Les transferts s'effectuent en parallèle du calcul sur le CPU, alors qu'ils doivent être séquentialisés sur le GPU.

4.2.3 Proposition : maximisation de l'utilisation

Les applications radio ont une particularité qui n'est pas exploitée dans les implantations actuelles sur GPU : elles traitent des données en permanence. Si on utilise le GPU pour une application de calcul scientifique comme par exemple MatLab, on ne sait pas par avance combien on aura d'échantillons à traiter. L'optimisation se fait donc sur une opération unique : on cherche à coller du mieux possible au modèle SIMD, et à minimiser le nombre d'opérations. Cependant, dans une application radio, on connaît le nombre d'échantillons : c'est une application de *streaming* (flux), ce qui implique qu'il y a virtuellement un nombre infini d'échantillons à traiter. Tant que le réseau est actif, il y a des données.

Cette propriété peut être utilisée pour adapter la stratégie précédente de deux manières différentes.

4.2.3.1 Parallélisation à grain fin

La première méthode permettant de maximiser l'utilisation du GPU est une extension de la méthode précédente. La maximisation de l'utilisation s'obtient en lançant plusieurs opérations ensemble, augmentant ainsi le nombre de *threads* à exécuter. Ces *threads* permettent toujours d'effectuer une sous-partie de l'opération, mais plusieurs opérations sont lancées en même temps.

On joue ici sur l'ordonnancement. Au lieu d'exécuter les données une par une, ou quand elles sont disponibles, on force une tâche à attendre jusqu'à ce qu'un certain nombre d'échantillons soit disponible. On présente le code correspondant pour le CPU dans l'algorithme 3. it_{max} est la dernière itération pour l'algorithme de FFT utilisé. Les vecteurs sont de taille N.

Algorithm 3 Parallélisation à grain fin

```
procedure EXECUTE                                              ▷ Code hôte
    Calculer it_max
    Calculer T_opt
    Attendre les T_opt vecteurs
    Copier N * T_opt échantillons vers le GPU
    while it < it_max do
        Définir les arguments in, out, it
        Exécuter FFT_R2_KERNEL(in, out, it) sur les T_opt vecteurs
        it ← it × 2
    end while
    Copier les N * T_opt échantillons résultats depuis le GPU
end procedure
procedure FFT_R2_KERNEL(input, output, itération)              ▷ Code GPU
    PAPILLON_RADIX2(input, output, itération)
end procedure
```

Tout réside donc dans le calcul de T_{opt}, qui représente le seuil d'exécution, c'est-à-dire le nombre d'opérations lancées en parallèle, et donc le nombre de vecteurs qui seront traités par une exécution. On précise ce paramètre dans la section 4.3. Cette méthode est une extension de la méthode classique.

4.2.3.2 Parallélisation à gros grain

Dans cette étude, nous proposons une nouvelle méthode qui modifie l'unité de base pour le calcul. Un *thread* traite intégralement une opération, et non plus une sous-partie. L'élément de base traité par un *kernel* OpenCL n'est donc plus l'échantillon, ou une sous-partie de vecteur, c'est la trame complète de l'application radio. Ceci est représenté sur la figure 4.5, et décrit par l'algorithme 4

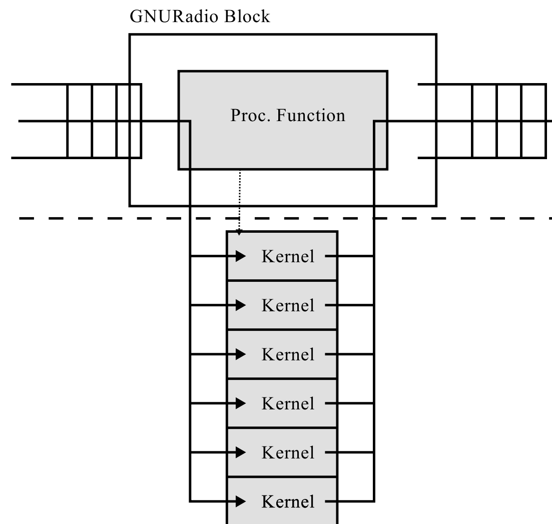


FIGURE 4.5 – Opération de radio logicielle sur GPU : approche proposée

Algorithm 4 Parallélisation à gros grain

```
procedure EXECUTE                                              ▷ Code hôte
    Calculer T_opt
    Attendre les T_opt vecteurs
    Copier T_opt vecteurs vers le GPU
    Définir les arguments in, out
    Exécuter FFT_KERNEL(in, out) sur les T_opt vecteurs
    Copier les T_opt vecteurs résultats depuis le GPU
end procedure
procedure FFT_KERNEL(input, output)                            ▷ Code GPU
    output ←FFT(input)
end procedure
```

Les deux méthodes de maximisation de l'utilisation ont le même objectif et utilisent toutes les deux un calcul fait sur plusieurs vecteurs en même temps. Cependant, même si elles permettent toutes les deux un gain conséquent par rapport au CPU, même pour des vecteurs de petite taille, les avantages et inconvénients ne sont pas les mêmes.

4.2.3.3 Comparaison

La première méthode est plus compliquée à déployer, et utilise de petits noyaux. Il est en effet nécessaire de trouver un algorithme qui se prête à une décomposition, avec des petits éléments indépendants. Ceci n'est pas toujours possible, comme par exemple pour un filtre IIR. L'utilisation des petits noyaux est nécessaire pour bien utiliser le GPU sur des calculs uniques afin de pouvoir utiliser tous les cœurs de calcul. Cependant, elle pénalise les performances, en augmentant le

surcoût lié au contrôle des *threads*. De plus, si on reprend la FFT comme exemple, le calcul se fait en plusieurs itérations, l'hôte est donc beaucoup sollicité. Par contre, cette méthode est plus modulaire, comme on le verra dans la section sur l'ordonnancement, et permet donc une réduction de la mémoire nécessaire. On peut également s'attendre à ce qu'elle soit plus efficace pour des seuils plus petits.

La méthode à gros grains est plus simple à déployer. Comme le *kernel* représente l'opération complète, il n'est pas nécessaire de trouver une implantation adaptée. De même, il est possible d'utiliser cette méthode pour des opérations ne permettant pas une adaptation au modèle SIMD, comme pour le filtre IIR. Tant que ces opérations sont appliquées sur les blocs, sans transfert de données d'un bloc à l'autre (IIR par bloc indépendant), il est possible d'utiliser le GPU efficacement, ce qui n'est pas le cas avec l'autre méthode. Les noyaux sont également plus gros, ce qui limite le surcoût de contrôle et permet de soulager l'hôte. Cependant, le besoin en mémoire est important, et la méthode est très peu modulaire, comme le montrent les expérimentations et la section sur l'ordonnancement. Elle requiert également des seuils élevés (pour être efficace, il faut lancer au minimum un calcul par PE).

4.3 Ordonnancement

Les deux méthodes proposées sont toutes les deux basées sur un seuil, qui permet de définir combien de vecteurs vont être traités par le GPU en même temps. Ce seuil influe sur différents paramètres.

4.3.1 Efficacité

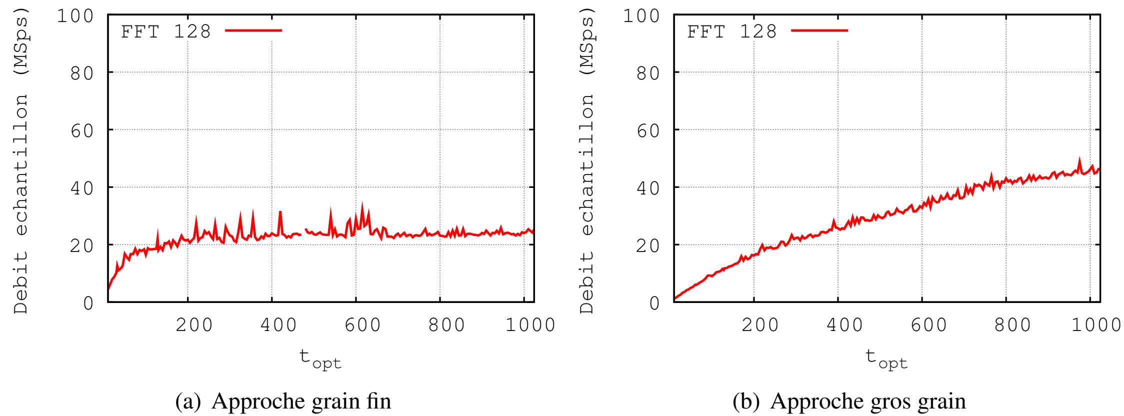
Le premier paramètre est l'efficacité de l'opération. On parle ici d'efficacité en termes de performances brutes. Afin de relativiser l'effet du surcoût de contrôle du GPU, il faut utiliser le GPU au maximum. Logiquement, augmenter le seuil permettrait donc d'améliorer les performances du GPU. Cependant, il faut nuancer cette affirmation. Afin d'étudier l'influence du facteur T_{opt} sur la performance globale du système, nous avons mis en place un test sur la FFT pour les deux approches précédentes. Il s'agit de calculer un nombre donné de FFT en faisant varier T_{opt}, et de regarder le temps nécessaire. Pour des raisons d'occupation mémoire, T_{opt} reste inférieur à 1024. Le GPU utilisé pour le test est constitué de 4 *devices*, chacun utilisant 32 *PE*, ce qui donne 128 PE. La valeur optimale de T_{opt} est dépendante du GPU. Les tests suivants sont réalisés en utilisant le GPU entier.

La figure 4.6 présente les résultats de ce test. On a donc le débit de traitement en million d'échantillons par seconde en fonction du seuil utilisé. Ce débit prend en compte le transfert des données vers le GPU. La courbe sans les transferts est similaire, même si les valeurs sont plus petites (facteur 10). Dans les deux cas, on distingue une forte augmentation du débit au début, puis une augmentation moins marquée avant une certaine stagnation. Il ne sert donc à rien d'augmenter indéfiniment le seuil. L'existence d'un seuil optimal est clairement visible sur ces résultats, d'où le nom T_{opt}.

Le seuil optimal de l'approche à grain fin, visible sur la figure 4.6(a), est plus faible que celui de l'approche à gros grain, sur la figure 4.6(b). Cependant, l'approche à gros grain permet d'avoir un temps d'exécution plus court.

4.3.2 Latence

Le seuil influence aussi la latence. L'utilisation des approches proposées génère une plus forte latence que dans le cas d'une exécution "normale", du fait de l'accumulation des vecteurs avant l'exécution. Cette latence peut se révéler problématique dans certains cas. Par exemple, en



(a) Approche grain fin

(b) Approche gros grain

FIGURE 4.6 – Débit échantillons en MS/s pour des FFT de 128 points pour différents seuils

présence d'un protocole de retransmission des paquets erronés, il faut connaître rapidement le résultat de la réception afin de pouvoir faire la demande de retransmission (ou transmettre l'acquittement à l'émetteur). Il y a donc une contrainte temporelle sur le décodage de la transmission. Une latence trop importante pour le traitement de certains échantillons peut générer une perturbation de la transmission. À l'opposé, la latence ne pose pas réellement de problèmes pour une application de *streaming* vidéo par exemple, tant que le débit est supporté.

Il est assez aisé de calculer la latence induite par l'utilisation du seuil dans le cadre d'une application radio. Elle dépend en effet de la fréquence d'échantillonnage et du seuil. Si on doit attendre d'avoir T_{opt} vecteurs de taille N pour lancer un calcul, et que les échantillons arrivent avec une fréquence f_s, on obtient la formule suivante pour la latence L.

$$L = \frac{T_{opt}N}{f_s}$$

La latence L ainsi obtenue est le temps nécessaire pour obtenir le nombre de paquets requis. Pour obtenir la latence globale du système, il faut ajouter le temps requis par chaque opération.

Notons tout de même qu'augmenter la fréquence d'échantillonnage pour diminuer la latence ne fonctionne pas. f_s représente bien ici la fréquence des échantillons à traiter en entrée de la chaîne. Cette chaîne est dimensionnée pour prendre en compte la fréquence d'échantillonnage, donc si on augmente la fréquence d'échantillonnage, N augmentera également. De même, si on décide de travailler sur des vecteurs plus petits, pour que le temps de calcul ne devienne pas prépondérant, il faudra augmenter T_{opt}, ce n'est donc pas non plus une solution viable.

Afin de limiter l'impact des deux approches sur la latence, on détaille deux solutions possibles. Ces deux solutions présupposent une connaissance de la structure de la transmission. Le principe de base est de tabler sur la répartition en paquets, et sur la structure du calcul GPU, qui permet de calculer en parallèle les paquets. Ainsi la multiplication du nombre de paquets à calculer n'augmentent pas le temps d'une instance du calcul. Afin de diminuer la latence, il peut être nécessaire de diminuer le seuil, ce qui diminuerait l'efficacité.

La première solution utilise une connaissance des échantillons prioritaires. Ainsi, si on sait que durant la réception, il y a un paquet prioritaire à intervalle régulier comme un *beacon* par exemple, il est envisageable de fixer le moment du calcul en fonction de cet intervalle. Si le reste de la transmission est non prioritaire, il est envisageable de tout accumuler dans des files, en attendant que les éléments prioritaires arrivent dans cette file. L'arrivée de ces éléments déclenchent automatiquement le calcul, afin de garantir une latence minimale pour ceux-ci.

La seconde solution utilise la séparation en paquets de la transmission. Ainsi, si on connait le temps de transmission d'un paquet, on peut caler T_{opt} sur le nombre d'échantillons correspondants.

Ainsi, les données qui une fois traitées auraient dues être mises en attente en attendant la fin du paquet sont toutes traitées en même temps, ce qui garantit une latence minimale pour le paquet.

Dans le cas où la transmission n'est pas structurée, par exemple dans les périodes d'attente de paquet (corrélation/synchronisation), il n'y a pas de solution élégante. Il convient de définir une latence acceptable pour n'importe quel échantillon, et de calculer T_{opt} en conséquence, à partir du débit d'échantillon en entrée du récepteur.

4.3.3 Mémoire

Finalement, il y a un lien entre T_{opt} et l'espace mémoire requis par l'application. Ainsi, plus on augmente le seuil, plus la mémoire nécessaire pour exécuter les différentes opérations sera importante. Il faut dès lors adapter T_{opt} aux caractéristiques du système. La mémoire n'est cependant prédominante dans le choix de T_{opt} que dans certains cas bien particuliers comme le *streaming* ou l'embarqué, la latence acceptable étant généralement plus limitante.

4.4 Intégration distribuée

Deux implantations de l'ordonnanceur ont été envisagées, l'une distribuée et l'autre centralisée. L'implantation distribuée a été mise en place dans l'environnement de radio logicielle GNURadio, son fonctionnement se prêtant bien à ce type d'implantation. L'implantation centralisée, plus complexe à mettre en place, mais plus prometteuse, est également détaillée. Elle a également été mise en place dans GNURadio, même si une intégration complète de cette méthode est difficile dans cet environnement. Les approches proposées de réduction de latence n'ont pas été implantées, l'environnement GNURadio n'étant pas adapté. L'intégration centralisée est présentée dans la section suivante (4.5).

4.4.1 Présentation

L'implantation distribuée utilise une approche "opération par opération", similaire à celle utilisée dans GNURadio. C'est l'implantation la plus simple à mettre en œuvre.

L'environnement OpenCL est instancié et géré dans une classe spécifique, qui contient les différents contextes, et qui permet de mettre en place les différentes queues de commande requises. Chaque opération qui doit être implantée nécessite un bloc GNURadio pour l'interface avec le reste de l'application et un *kernel* correspondant à l'application.

L'ordonnanceur est implanté de manière distribuée, dans chacun des blocs GNURadio. C'est une implantation pseudo-statique. Il est possible de faire évoluer l'ordonnancement dans le temps, mais de manière simpliste, à cause de l'absence d'information sur le reste de l'application. Le paramètre T_{opt} est défini dans le bloc lors de son instanciation.

Cette implantation permet de mélanger différents modes d'exécution entre GPU et CPU. Elle peut utiliser indifféremment les différentes méthodes de communication définies précédemment.

Dans la pratique, cette approche est assez simple à mettre en œuvre. Chacune des opérations à implanter prend la forme d'un nouveau bloc GNURadio dans l'API. Ce bloc a la même interface qu'un bloc classique, avec en plus comme attribut la classe spécifique à l'OpenCL, qui contient toutes les structures propres. Ces structures doivent pour la plupart être uniques pour l'application, cette classe est donc instanciée une seule fois. L'utilisateur qui veut utiliser des blocs GPU peut utiliser tous les blocs définis, de la même manière qu'il utiliserait des blocs classiques.

4.4.2 Communications

Les applications SDR en général, et GNURadio en particulier, sont basées sur une communication entre blocs de type FIFO. Dans le cas d'une implantation sur CPU, ces FIFO sont logicielles,

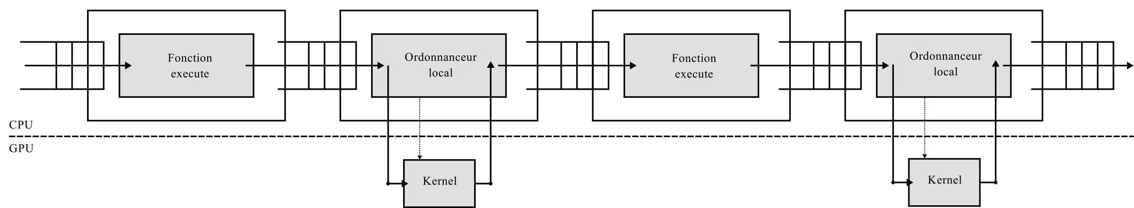


FIGURE 4.7 – Implantation distribuée

implantées en utilisant un *buffer* tournant, et des indices. Dans cette étude, l'utilisation du GPU modifie le problème. Il devient nécessaire de faire transiter les données entre les deux domaines, et la méthode utilisée est différente. Les performances sont également à prendre en compte, les transferts de données vers et depuis le CPU étant un goulot d'étranglement de l'utilisation du GPU. Deux approches pour effectuer ces transferts sont étudiées.

4.4.2.1 Intégration transparente

La première solution est la plus évidente. Elle est utilisée dans [Mil10]. Elle permet de rendre l'utilisation du bloc GPU transparente pour le reste du système, en gérant le transfert dans le bloc. C'est l'approche représentée sur les figures 4.4 et 4.7. L'intégration du bloc dans le domaine CPU est forte, les données transitant dans les FIFO logicielles du CPU.

Cette méthode permet d'alterner des blocs CPU et GPU facilement étant donnée que les données résident dans le CPU. En laissant au bloc la responsabilité de gérer sa mémoire GPU, elle permet de diminuer l'empreinte sur la mémoire du GPU, la mémoire étant allouée au plus proche de ce qui est requis. Du point de vue du développeur, l'approche est aisée à implanter, puisqu'elle ne nécessite pas de modifications de la structure de GNURadio. Du point de vue de l'utilisateur, les blocs sont interchangeables et il est donc possible d'utiliser l'un ou l'autre domaine indifféremment [2].

Cependant, c'est également l'intégration qui est potentiellement la moins efficace. Utiliser des alternances de GPU et de CPU pour bénéficier des implantations les plus efficaces n'est pas forcément optimal. Les coûts de transfert d'un domaine à l'autre sont tels, comparés aux coûts de l'exécution, qu'il peut être préférable de n'utiliser qu'un seul domaine. L'inconvénient majeur dans cette approche est que même si deux opérations successives ont lieu sur le GPU, les données vont devoir repasser par le CPU entre ces deux opérations. La perte de performance est difficilement acceptable.

4.4.2.2 Buffer OpenCL

Afin d'éviter cette perte de performance, il faut définir un nouveau type de *buffer*, adapté à l'utilisation du GPU.

La première idée pour réaliser un tel *buffer* est d'utiliser une fonctionnalité spécifiée dans l'API OpenCL : l'utilisation d'une zone dans la mémoire de l'hôte comme base pour un *buffer* OpenCL. Les données sont ensuite transférées selon les besoins dans la mémoire du GPU de manière transparente pour l'utilisateur. Cependant, cette solution offre des résultats plus que décevants, et fait toujours transiter les données par le CPU. De plus, le *runtime* OpenCL est pour l'instant une boîte noire, et il n'est donc pas possible de savoir exactement comment cette fonctionnalité est gérée. Nos expérimentations montrent des variations importantes de l'efficacité de la méthode. Une autre fonction d'OpenCL, qui permet de *mapper* un *buffer* GPU dans le plan d'adressage mémoire du CPU, a également dû être abandonnée, à cause de résultats aléatoires.

2. du point de vue de la communication au moins. L'interface du bloc peut être modifiée par des paramètres propres à l'implantation OpenCL

On en revient donc à la définition complète d'un buffer OpenCL. Le GPU est une entité de calcul et non de contrôle. De plus, le CPU reste responsable du contrôle de l'application radio complète, il a donc besoin d'informations sur ce qu'il se passe dans cette application, et les transferts entre GPU et CPU sont difficiles. L'implantation du contrôle d'un buffer tournant dans le GPU est donc exclue.

Par contre, nous pouvons utiliser une conception qui utilise les deux domaines, avec le contrôle dans l'hôte (le CPU), et les données physiques dans le GPU. Des blocs de transferts explicites des données entre les deux domaines sont proposés. Cette approche est présentée dans la figure 4.8. Le contrôle est représenté en pointillés, les données réelles en trait plein.

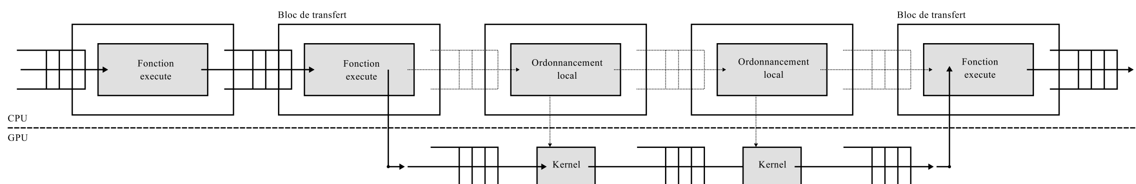


FIGURE 4.8 – Utilisation de *buffers* spécifiques pour les GPU

On utilise en fait une FIFO "fantôme" qui réside dans le CPU, et qui permet de refléter ce qu'il se passe dans le GPU. Cette FIFO de contrôle est une FIFO d'indices. Lors de l'initialisation, la FIFO est remplie avec ses indices, et la mémoire est allouée dans le GPU. Lors de l'utilisation de ces FIFO, le bloc GNURadio lit la FIFO de contrôle et en déduit l'indice correspondant dans la FIFO OpenCL. Cet indice est transmis en tant que paramètre du *kernel*. Une fois le calcul effectué, le bloc retourne le nombre de valeurs produites, ce qui dans GNURadio permet de maintenir les FIFO à jour. Il n'y a par contre aucune données réellement écrites dans la mémoire de l'hôte. Les blocs de transferts permettent de faire l'interface entre ces deux types de FIFO. Les données sont réellement lues en entrée, puis transférées vers le GPU. En sortie, la FIFO de contrôle est mise à jour en conséquence. Les opérations inverses sont effectuées dans l'autre sens du transfert.

Cette approche a un inconvénient majeur : elle consomme de la ressource mémoire "inutile", puisque ne contenant pas réellement de données. De manière générale, l'implantation de FIFO consomme également plus de mémoire que l'implantation de simple *buffers* temporaires comme pour l'approche précédente. Cependant, elle permet d'éviter les transferts inutiles, tout en conservant une vision de l'état des FIFO pour l'ordonnanceur de GNURadio. Les performances sont donc améliorées.

4.5 Intégration centralisée

4.5.1 Présentation

L'implantation centralisée est plus proche de l'architecture OpenCL. Le contrôle du GPU étant centralisé sur l'hôte, il paraît préférable d'utiliser une implantation centralisée pour intégrer le GPU dans GNURadio.

Le principe de cette approche est de regrouper dans un seul et même bloc GNURadio toute une séquence d'opérations réalisées sur GPU. Ce bloc est responsable d'ordonnancer les *kernels* et de gérer les communications entre les différentes opérations, comme présenté sur la figure 4.9.

Cette approche a trois grands avantages :

– elle limite le besoin en mémoire. Puisqu'un seul bloc gère toute une séquence d'opérations GPU, il n'est plus forcément nécessaire d'avoir des FIFO entre ces opérations. Les seules FIFO seront en entrée et en sortie de ce bloc. On n'utilisera donc que la mémoire requise en fonction du T_{opt} maximal ;

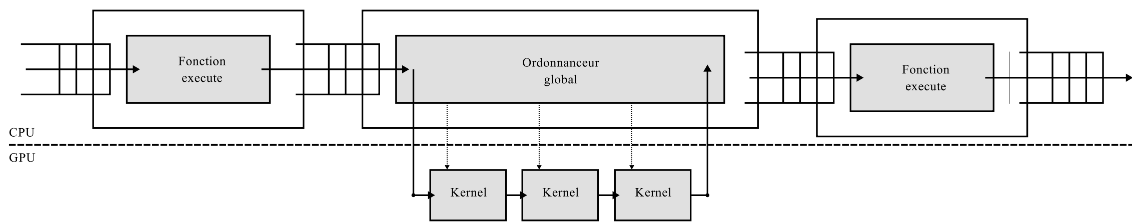


FIGURE 4.9 – Implantation centralisée

– elle permet d'améliorer encore l'utilisation du GPU. La plupart des GPU récents sont basés sur des *units* indépendants (les *multiprocessors* de NVidia en sont l'exemple type). Pour atteindre l'utilisation maximale, on a remarqué dans la section 4.3.1 que T_{opt} pouvait être très grand. Centraliser le contrôle peut permettre de lancer en parallèle différentes tâches, qui seront donc exécutées de manière concurrente. Le GPU peut donc potentiellement être utilisé correctement même avec un seuil plus petit. Cet avantage n'est pertinent que pour les GPU récents, qui permettent l'utilisation complètement indépendante des *units* (*cf.* le *compute capability* de NVidia, [NVI10]) ;
– plus généralement, elle permet d'utiliser des ordonnancements plus souples, puisqu'il y a une connaissance de l'état de toutes les opérations dans le bloc qu'il n'y avait pas avec l'approche distribuée.

Cette approche est cependant moins transparente pour l'utilisateur que l'approche distribuée, étant donné qu'elle se distingue de l'approche GNURadio. En l'état actuel de l'implantation, l'utilisateur doit spécifier quelles opérations seront effectuées dans le bloc GNURadio. Il est envisageable d'automatiser cette répartition. Cette approche n'a également d'intérêt que si beaucoup d'opérations sont réalisées en séquence. Cependant, cette limitation est un faux problème, puisque le coût d'un transfert mémoire limite les passages d'un domaine à l'autre.

4.5.2 Communications

Les communications de l'intégration centralisée sont assurées par des FIFO logicielles classiques de GNURadio en entrée et en sortie du bloc complet, et par des *buffers* entre les différentes opérations GPU. La taille des blocs peut être calculée au plus juste. On connait en effet le ratio entrées/sorties, et le seuil de calcul qui sera utilisé, c'est-à-dire le nombre d'éléments qui seront traités par une occurrence de l'opération.

L'approche centralisée permet donc de supprimer le problème des communications, même si le calcul des tailles n'est pas forcément évident.

4.5.3 En pratique

Dans la pratique, l'implantation centralisée consiste en une réalisation d'un bloc GNURadio qui réalisera plusieurs opérations, et qui s'intégrera normalement dans une application. Ce bloc sert d'interface avec le GPU. Il contient ses différentes structures, et une représentation de l'application GPU, c'est à dire de l'ensemble des opérations qui seront effectuées, ainsi que des connexions entre ces opérations. La fonction qui sera exécutée lors de l'appel de ce bloc dans l'application sera en fait l'ordonnanceur, qui va lire les données en entrée, s'occuper du transfert vers le GPU, puis exécuter son application interne, avant de transférer à nouveau le résultat vers le CPU pour l'écrire en sortie.

Implanter l'approche centralisée revient à ajouter la possibilité de créer un bloc à partir d'opérations GPU. Les mécanismes utilisés ressemblent fortement aux mécanismes qui permettent de créer une application dans GNURadio. Une opération GPU est représentée à l'aide des

paramètres suivants :
- un *kernel* qui représente l'opération à effectuer par le GPU ;
- une fonction pour définir les arguments de l'opération et pour lancer une exécution du *kernel* ;
- un ratio entrées/sorties qui définit le rapport entre le nombre d'entrées et le nombre de sorties ;
- un seuil optimal T_{opt}.

Dans la gestion centralisée, on calcule dans la phase d'initialisation un ordonnancement, avec les tailles de *buffers* requises entre les différentes opérations GPU, et le nombre d'exécution. Cet ordonnancement est ensuite utilisé durant l'exécution pour lancer les noyaux. On propose un algorithme permettant d'initialiser l'ordonnancement. L'objectif de l'ordonnanceur est de définir :
- le seuil pour chacune des opérations considérées ;
- la taille des *buffers* entre les blocs ;
- la séquence de *kernels* à exécuter.

Dans la première version présentée dans l'algorithme tout le GPU est utilisé pour réaliser une opération. En présence d'un GPU compatible avec le multitâche, il est possible d'appliquer une autre modification, qui permet potentiellement de réduire l'empreinte mémoire et la latence sans diminuer le débit. Si le GPU est constitué de *units* indépendants, on peut lancer plusieurs opérations en parallèle sur le GPU. Cette approche a plusieurs avantages. Les *units* étant plus petits, ils ont besoin d'un seuil moins élevé pour être utilisé de manière optimale. Le temps d'exécution est évidemment plus long, mais comme les tâches sont exécutées en parallèle, on peut espérer garder le même débit, mais en diminuant la latence et la mémoire requise.

4.6 Résultats

4.6.1 Synthèse des contributions

Avant d'examiner le résultat des différentes méthodes proposées, résumons l'ensemble des propositions. L'objectif principal est d'étudier les possibilités d'utilisation du GPU pour la radio logicielle. On s'intéresse pour l'instant à une exécution sans limitation de consommation électrique, et sur des plateformes qui peuvent être puissantes. Concrètement, on n'aborde pas dans nos expérimentations le cas des systèmes embarqués.

Les propositions de cette étude tournent autour de trois axes :
- une conception d'une opération SDR sur GPU. L'étude dans [Mil10] proposait une solution pour des opérations basiques. Nous nous intéressons dans cette étude à des opérations plus complexes, et nous proposons une approche qui utilise l'opération complète comme *kernel*, au lieu de définir un algorithme optimisé pour le GPU ;
- la définition de l'ordonnancement des blocs GPU, qui diffère de celui des blocs CPU à cause du besoin de charger au maximum le GPU. Cet ordonnancement se base sur un seuil, et peut être utilisé pour toutes les approches ;
- deux stratégies d'intégration de ces blocs dans une application, c'est-à-dire de la communication entre les éléments, et de l'implantation de l'ordonnancement dans un environnement réel, en l'occurrence GNURadio.

Les expérimentations visent à évaluer la différence de performance entre une implantation CPU, et les différentes approches proposées dans ce travail ou dans d'autres.

4.6.2 Opérations implantées

Afin d'étudier les résultats des différentes solutions proposées, trois opérations différentes ont été sélectionnées et implantées. On présente ici les détails de l'implantation.

4.6.2.1 FFT

La transformée de Fourier rapide est une opération classique de traitement du signal permettant de passer du domaine fréquentiel au domaine temporel et inversement. La formule a été donnée précédemment. Afin d'implanter de manière efficace la FFT, on utilise l'étude [bea] qui s'intéresse aux implantations optimales de la FFT sur GPU. L'algorithme utilisé est un algorithme à base de papillons radix-2, présenté dans l'algorithme 5 pour une FFT de N points.

Algorithm 5 Algorithme FFT Radix-2

```
input ← vector                                  ▹ Contient l'entrée de l'itération courante
it_max ← log_2(N)
for it ∈ [0, it_max[ do
    for i ∈ [0, N/2[ do
        BUTTERFLY(i, i + N/2, i × it, (i + 1) × it)
    end for
end for
```

L'implantation de l'opération FFT sert d'exemple dans les sections précédentes, on ne revient donc pas dessus. La première approche utilise comme *kernel* la fonction `butterfly`, qui est un papillon de FFT. Au contraire, l'approche à gros grain utilise toute la fonction présentée dans l'algorithme comme *kernel*. L'intérêt de la FFT dans cette étude réside dans l'existence d'algorithmes optimisés pour le GPU.

4.6.2.2 *Demapping*

Le *demapping* est une fonction simple qui permet de traduire les échantillons du domaine complexe en information binaire. On s'intéresse ici à une modulation PSK, qui utilise une variation de phase pour coder l'information. Plus précisément, dans les expérimentations effectuées, la constellation est une constellation QPSK avec les phases $\frac{\pi}{4}$, $\frac{3\pi}{4}$, $-\frac{\pi}{4}$ et $-\frac{3\pi}{4}$. La constellation est assez simple à décoder, puisqu'il suffit de regarder dans quel cadran se trouve l'échantillon en cours de traitement, alors qu'une constellation plus complexe requiert plusieurs calculs de distance. L'algorithme 6 est utilisé, dans lequel I et Q représente les parties réelles et imaginaires de l'échantillon complexe (représentation classique en traitement du signal).

Algorithm 6 *Demapping* d'une constellation PSK

```
if I > 0 et Q > 0 then
    θ ← π/4
else if I < 0 et Q > 0 then
    θ ← 3π/4
else if I < 0 et Q < 0 then
    θ ← −3π/4
else if I > 0 et Q < 0 then
    θ ← −π/4
end if
TRADUIRE(θ)
```

Dans l'approche à grain fin, on définit la taille du *kernel* en fonction de l'élément de sortie. Dans le cas d'un QPSK, un échantillon code 2 bits. Le *kernel* traitera donc 4 échantillons, afin d'avoir un nombre de bits entier en sortie. Dans le deuxième cas, le *kernel* traite tout un bloc. Par exemple, si une trame est transformée à l'aide d'une FFT (modulation OFDM), le *kernel* travaillera sur tout le résultat de la FFT.

4.6.2.3 IIR

Finalement, nous nous intéressons à l'implantation d'un filtre IIR, opération pour laquelle l'extraction d'un algorithme SIMD est plus compliquée. On peut citer comme étude [Kut08] qui se base sur une extraction des parties récursives et non récursives. Cette approche s'applique bien dans le cas d'un processeur SIMD pas trop large, mais beaucoup plus difficilement dans le cas d'un processeur SIMD très large comme le GPU. On propose donc ici une nouvelle approche qui utilise la parallélisation à gros grain, basée sur des modifications similaires, mais adaptées pour le GPU.

Le filtre IIR est un filtre linéaire :

$$IIR(n) = \sum_{i=0}^{F} b_i x(n-i) - \sum_{j=1}^{B} a_j IIR(n-j)$$

Si on s'intéresse à un calcul de ce filtre sur des blocs, sans répercussion du résultat du bloc précédent sur le premier élément d'un bloc (il y a indépendance complète entre les blocs), on obtient une implantation parallélisable sur GPU en utilisant l'approche à gros grain. Le *kernel* est alors le calcul complet d'un IIR bloc. La formule de ce filtre est :

$$IIR_t(n) = \sum_{i=0}^{F} b_i x(tN+n-i) - \sum_{j=1}^{\min(n,B)} a_j IIR_t(n-j), \forall n \in \{0 \ldots N-1\}$$

En utilisant une correction des coefficients, on peut calculer le filtre complet. La correction permet de corriger l'ensemble des éléments d'un bloc t, en utilisant les éléments corrigés du bloc $t-1$, et des coefficients constants c_j :

$$\begin{cases} IIR(tN+n) & = IIR_t(n) - \sum_{j=1}^{B} c_j(n) IIR(tN-j) \\ c_j(0) & = a_j \\ c_j(n) & = a_{j+n} - \sum_{k=1}^{n} a_k c_j(n-k) \end{cases} \quad \forall n \in \{0 \ldots N-1\}, \forall t \in \mathbb{N}$$

La démonstration de la correction de cette formule se fait par récurrence. Elle est présentée dans l'annexe B. Dans cette formule, le calcul de chaque élément n du bloc est indépendant des autres éléments. On peut donc appliquer bloc par bloc un modèle SIMD.

On en déduit l'algorithme 7 pour le calcul d'un filtre IIR en utilisant le GPU. La procédure `iir_block` permet de calculer une IIR_t pour un bloc donné. Ce calcul est facilement parallélisable sur un GPU, puisqu'il n'y a pas de dépendances entre les différents éléments d'un bloc. Une fois que plusieurs blocs ont été calculés, on peut appliquer la correction. Le calcul de l'IIR corrigé ne dépend que des IIR_t, obtenus lors de la phase précédente, on peut donc utiliser un algorithme SIMD. C'est la fonction `correction` qui est utilisée dans ce cas.

On notera que l'utilisation de cette approche coûte cher à l'initialisation ($O(N \times B^2)$ multiplications/accumulations). Il y a également un surcoût durant l'exécution, de B multiplications/accumulations par éléments. Ce surcoût est similaire à l'approche de [Kut08]. L'algorithme nécessite également $N \times B$ emplacements mémoires afin d'enregistrer les coefficients c_j. En le comparant à l'algorithme de [Kut08], la méthode est similaire. Cependant, certaines différences existent qui permettent de rendre notre approche plus efficace et plus adaptée à la radio logicielle :

- nous utilisons une approche en bloc et non pas une approche en flux, ce qui permet de maximiser l'utilisation du GPU, et d'être plus efficace ;
- la taille de bloc utilisée est supérieure à N, qui est le cas problématique de l'approche dans [Kut08].

Algorithm 7 Implantation d'un filtre IIR

```
procedure IIR_BLOC(B, F, N)
    for n ∈ [0, N[ do
        IIRt[i] ← 0
        for i ∈ [n − F, n] do                          ▷ boucle sur l'entrée
            IIRt[n]←IIRt[n]+a[i]x[i]
        end for
        for j ∈ [1,MIN(n,B)] do                        ▷ boucle récursion
            IIRt[n]←IIRt[n]−b[j]IIRt[n − j]
        end for
    end for
end procedure
procedure CORRECTION(c, B, N)
    for n ∈ [0, N[ do
        for j ∈ [1, B] do
            IIR[n] ← IIRt[n] - c[j][n] × IIRt[n − j]
        end for
    end for
end procedure
```

Utiliser le corps de la boucle comme *kernel* n'est pas envisageable à cause de la dépendance des données. L'approche à grain fin n'est donc pas adaptée. On pourrait concevoir une solution qui mélangerait les deux approches, avec un ordonnancement qui se baserait sur l'approche à gros grain, mais suite à nos expérimentations, le GPU n'arrive pas à extraire l'indépendance, et exécute tous les noyaux un par un. On n'étudiera donc pas cette solution. De même, comme attendu, nos essais sur d'autres implémentations indépendantes de l'IIR n'ont pas été concluant. L'approche à grain fin ne peut donc pas implanter efficacement l'IIR.

La nouvelle approche à gros grain que nous avons introduite permet en revanche de définir un nouvel algorithme pour l'IIR, qui exploite efficacement le GPU. En effet, le calcul de l'IIR bloc par bloc est compatible avec l'architecture SIMD du GPU, puisque le calcul sur un bloc ne dépend pas du calcul sur le bloc précédent. L'opération de correction à la fin, si on a besoin de la dépendance d'un bloc à l'autre, est également parallélisable (bloc par bloc). Cette approche permet donc d'implanter des opérations qui ne seraient pas implantables autrement.

4.6.3 Protocole d'expérimentation

Afin de tester les différentes solutions proposées pour permettre l'utilisation du GPU dans la radio logicielle, on utilise des données générées aléatoirement dans un fichier, et on applique les calculs qui nous intéressent. Le résultat est ensuite écrit dans un fichier, puis comparé au résultat d'une exécution CPU avec un GNURadio non modifié (et donc supposé exact). Le protocole est représenté sur la figure 4.10. On calcule le temps requis pour effectuer cette opération (hors initialisation). Ce temps est ensuite utilisé pour calculer le débit échantillon de l'application, c'est à dire le nombre d'échantillons traités par seconde.

Le matériel utilisé pour l'exécution est un ordinateur de bureau basé sur

- un CPU Intel Core i5 760 (Quad Core, 2.80 GHz, 2Mo de cache) ;
- 8 Go de mémoire DDR3 ;
- une carte graphique ASUS connectée sur le bus PCI-Express 16x (8 Go/s) basée sur :
  - un GPU NVidia GTS 450 de 4 *multiprocessors* de 32 *cores* chacun, cadencés à 1566 MHz, soit un total de 128 *cores* ;

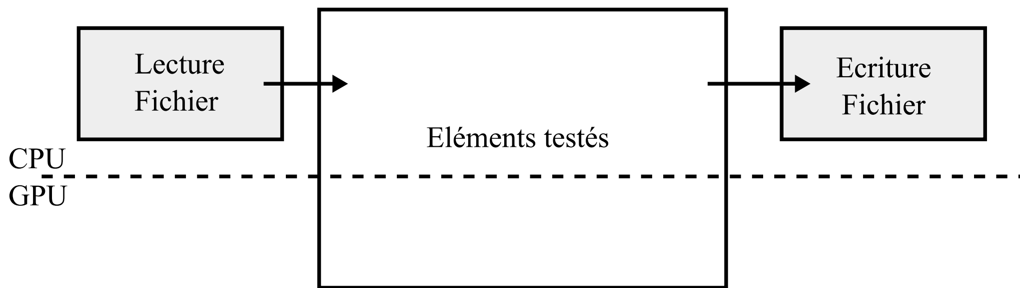


FIGURE 4.10 – Protocole d'expérimentation

– 1 Go de mémoire DDR5, avec une bande passante de 60 Go/s en interne.

Les opérations sont exécutées sur le GPU lui-même. NVidia fournit, dans ses pilotes propriétaires, l'ensemble de l'environnement OpenCL, ainsi que le compilateur des *kernels*. Les opérations sont donc implantées en utilisant la spécification normalisée d'OpenCL, puis en faisant l'édition de lien avec la librairie OpenCL fournit par NVidia, qui donne l'implantation des différentes fonctions de la spécification. Il y a donc beaucoup d'éléments inaccessibles dans l'implantation. Les conclusions que l'on tires des résultats suivants sont vraies pour le GPU utilisé. Pour des raisons de moyens, il n'a pas été possible d'envisager d'autres GPU, on peut cependant faire quelques suppositions qui seront présentées au fur et à mesure.

4.6.4 Résultats des opérations unitaires

L'étude des opérations unitaires permet d'avoir une première vision de l'efficacité des différentes méthodes considérées. On étudie ici les débits d'échantillons pour 10 000 opérations. Deux facteurs entrent en jeu pour étudier les performances : le seuil minimal pour lancer l'exécution, et la taille du vecteur. Ces deux paramètres permettent en effet de définir à quel point le GPU est utilisé.

4.6.4.1 Influence du seuil

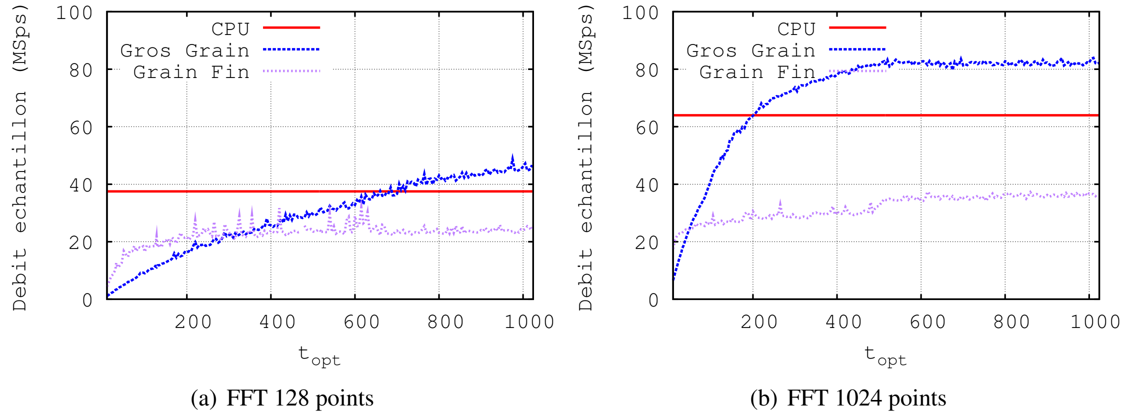
L'étude de l'influence du seuil permet de déterminer un seuil optimal à utiliser pour chacune des opérations et pour chacune des tailles de vecteur, afin de permettre une exécution optimale.

La figure 4.11 montre le débit échantillon lors d'une exécution de 10 000 FFT pour différentes valeurs du seuil de déclenchement. On étudie trois implantations : une implantation classique sur CPU, une implantation à grain fin et une implantation à gros grain. On peut remarquer deux choses sur les courbes :

– l'implantation à grain fin donne de meilleurs résultats que l'implantation à gros grain pour des petites valeurs de seuil ;
– l'implantation à gros grain est plus stable dans ses performances que l'implantation à grain fin, et permet d'atteindre un débit minimal plus grand.

L'implantation à grain fin n'est clairement pas adaptée à ce type d'applications. Le point de stabilisation que l'on remarque sur la courbe pour $N = 1024$ correspond à une valeur de seuil de 512. Le GPU utilisé pour les tests utilise 128 cœurs, ce seuil correspond donc à 4 fois le nombre de cœurs. C'est le nombre minimal requis pour obtenir un ordonnancement optimal de manière déterministe, étant donnée la taille du *kernel*. On peut supposer que plus le nombre de *cores* augmente, plus le seuil augmentera, ce qui aura donc pour effet d'augmenter la latence. A l'opposé, moins on a de *cores*, et plus le seuil diminue. Ceci signifie qu'un GPU avec peu de PE sera probablement utilisé de manière optimal avec un nombre de vecteurs faible. Cependant, les débits atteignables seront probablement plus faibles aussi.

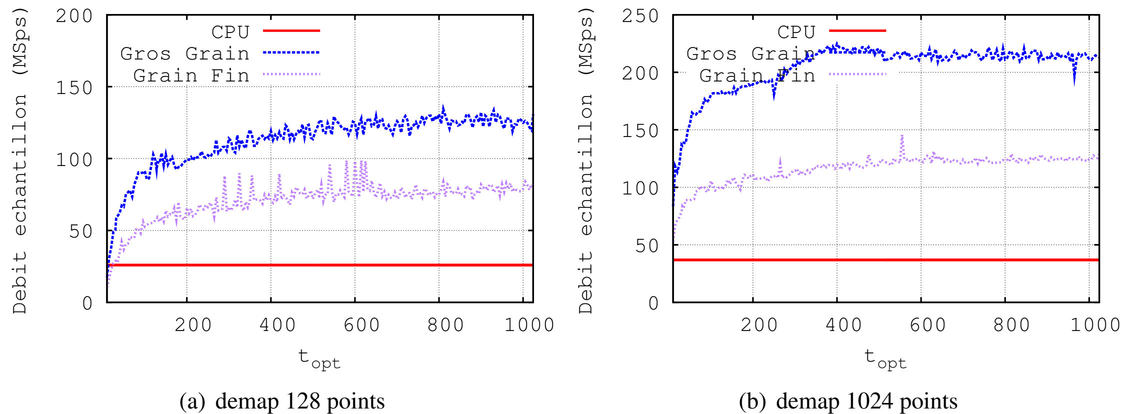
On constate également que l'implantation à grain fin est plus efficace pour $N = 1024$ que pour $N = 128$. Ceci était attendu, du fait de l'étude de performance réalisée en [bea].



(a) FFT 128 points (b) FFT 1024 points

FIGURE 4.11 – Débit échantillons en MS/s pour la FFT pour différents seuils, à taille de vecteur fixée

Le gain pour l'opération de *demapping* est beaucoup plus perceptible, comme on peut le voir sur la figure 4.12. L'approche à gros grain est également meilleure que l'approche à grain fin en termes de performance. Cependant, cette dernière permet ici d'augmenter le débit de traitement par rapport au CPU, ce qui n'était pas le cas pour la FFT. Ceci s'explique par la simplicité du calcul comparé par exemple à l'opération FFT.



(a) demap 128 points (b) demap 1024 points

FIGURE 4.12 – Débit échantillons en MS/s pour le *demapping* pour différents seuils, à taille de vecteur fixée

Pour l'opération IIR, il n'y a pas d'implantation à grain fin. On donne les résultats avec la correction pour la récursion bloc à bloc, et sans cette correction. L'utilisation de la correction impose un bloc de grande taille également. Dans la deuxième phase, le *kernel* s'applique en effet sur un vecteur de la taille du bloc, et pour maximiser l'utilisation, il faut donc avoir un bloc de taille conséquente. Ceci se remarque sur la figure 4.13. Le filtre IIR utilise ici $F = 1$ et $B = 1$ comme paramètre. Pour $N = 128$, la version en bloc de l'algorithme, sans la correction, est efficace, comparable à l'exécution CPU. Au contraire, la version avec la correction est peu efficace comparée au CPU. Par contre, pour $N = 1024$, les courbes se rapprochent. Le GPU prend l'avantage à partir de $N = 8192$, comme on le voit sur la figure 4.13(c). Il y a équivalence en termes de débit pour $N = 4096$.

L'opération IIR étant à la base une opération linéaire et non pas en bloc, on peut donc arbitrairement augmenter la taille du bloc N en diminuant le seuil pour obtenir de bonnes performances

sur GPU, sans augmenter la mémoire. Comme le seuil à utiliser diminue quand N augmente (on a quasiment $N \times T_{opt}$ constant), augmenter N ne change pas la taille requise. Ceci est vrai pour le filtre corrigé. La taille du bloc a son importance pour le filtre non corrigé.

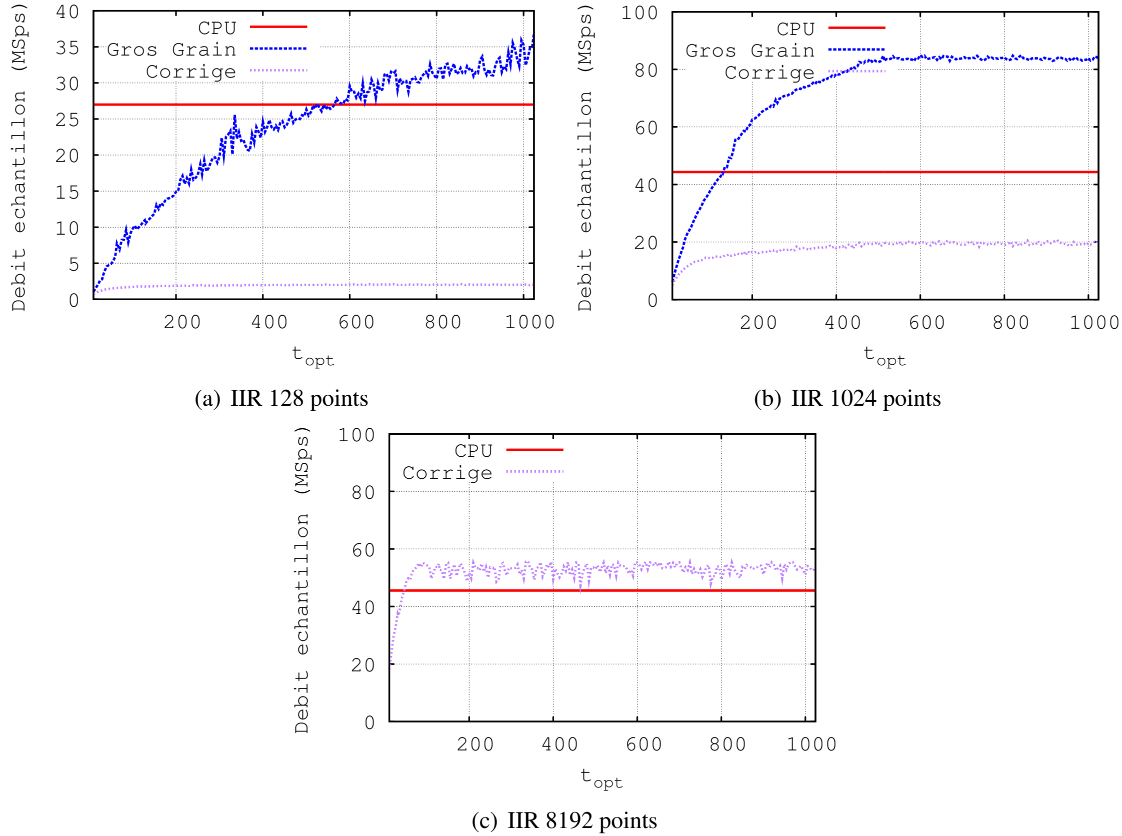


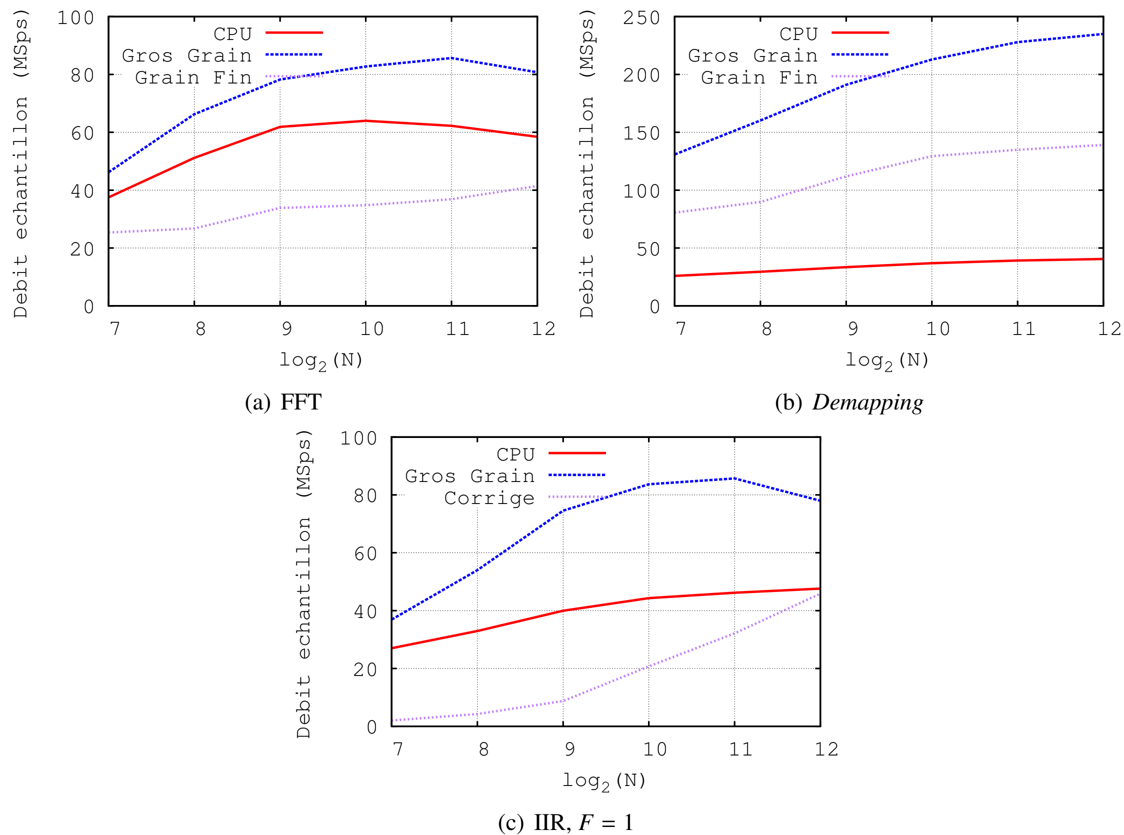
FIGURE 4.13 – Débit échantillons en MS/s pour l'IIR pour différents seuils, à taille de vecteur fixée

Plusieurs remarques sont à faire sur ces résultats. Tout d'abord, le filtre IIR utilisé est un filtre avec une seule récursion. Le calcul est donc très simple pour le CPU. L'augmentation du nombre de récursions complexifierait le calcul, et rendrait donc le GPU plus attractif.

On remarque également le comportement étrange de la courbe pour la version corrigée. Deux raisons expliquent cet aplatissement soudain de la courbe autour de la valeur du CPU. La première raison est la taille limitée de la FIFO d'entrée, qui empêche d'atteindre le seuil optimal. Les valeurs au delà du seuil indiqué ne sont donc pas significatives. On remarque cette limitation sur les résultats en fonction de la taille des vecteurs dans la section 4.6.4.2. La deuxième raison est une erreur de la version du driver utilisée au moment des expérimentations. Alors que dans les expérimentations précédentes, le GPU réussissait à extraire le parallélisme du *kernel* de la correction, la version actuelle n'y parvient pas. Les résultats des expérimentations précédentes ne sont par contre pas complets. Nous espérons obtenir à nouveau des résultats intéressants avec des versions ultérieures.

4.6.4.2 Influence de la taille des vecteurs

La taille des vecteurs influence principalement le seuil requis pour être efficace. On montre sur la figure 4.14 les résultats pour différentes tailles de vecteur, avec un seuil optimal pour chacun des vecteurs. On remarque que dans tous les cas sauf l'IIR avec la correction, l'approche à gros grain permet d'obtenir un gain par rapport au CPU pour les opérations considérées, alors que l'approche classique à grain fin donne de plus mauvais résultats que l'implantation CPU.



(a) FFT (b) *Demapping*

(c) IIR, $F = 1$

FIGURE 4.14 – Débit échantillons en MS/s en fonction de N, pour un seuil optimal calculé expérimentalement

On remarque également, ce qui est nouveau, que pour une taille de bloc suffisante, utiliser un GPU peut améliorer (ou au moins ne pas détériorer) l'exécution d'un filtre IIR par rapport au CPU.

4.6.5 Résultats pour des applications multitâches

Afin d'étudier l'efficacité du GPU dans le cadre d'une application avec plusieurs opérations, on utilise les trois opérations en séquences (IIR, FFT, *demapping*). Cette application ne correspond pas à une application réelle, mais permet de voir l'évolution des performances dans le cas du multitâche. Deux éléments ont été étudiés qui peuvent influer sur le multitâche : le type d'intégration (centralisée ou distribuée) et les communications entre les opérations. Il parait évident que l'utilisation du *buffer* OpenCL permet d'obtenir des performances plus intéressantes. On ne s'intéressera ici qu'à la différence entre l'exécution centralisée et distribuée, et sur l'effet de l'utilisation des *units* indépendantes.

Les résultats sont présentés dans la figure 4.15. L'approche implantée utilise l'intégralité du GPU pour chaque opération. Nous avions déjà discuté de la possibilité de réduire le seuil en utilisant les capacité de parallélisme de tâches des nouveaux GPU. Cette possibilité est en cours d'étude. Les résultats montrent un réel intérêt pour l'approche centralisée, et un vrai gain d'un facteur allant jusqu'à 4 pour un découpage en blocs de 512 échantillons. La "dégradation" des performances observées pour $N > 512$ s'explique par une taille de FIFO non adaptée, les seuils optimaux ne pouvant plus être atteints. L'évolution proposée précédemment devrait permettre de corriger ce problème. Une augmentation de la taille des FIFO est également possible.

Ces résultats sont extrêmement dépendants du GPU utilisé. Dans les conditions utilisées pour les résultat de la figure 4.15, c'est-à-dire avec l'intégralité du GPU pour chacune des opérations, le

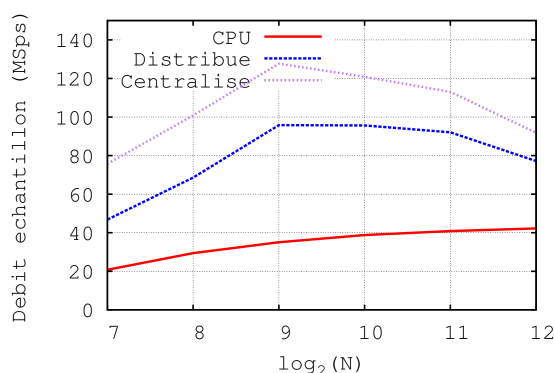


FIGURE 4.15 – Résultats pour une séquence d'opérations, pour un seuil optimal calculé expérimentalement

nombre de cœurs influe sur les résultats par opération, et donc sur les résultats globaux. Certains GPU récents offrent la possibilité d'utiliser les différentes *computing units* de manière indépendante. Il devient donc possible de séparer l'exécution d'une séquence d'opérations en créant un *pipeline* d'opérations (comme expliqué en 4.5.3. Cette séparation n'est pas encore implémentée. Cette solution permet également d'envisager des optimisations sur la localisation des données, en utilisant de la mémoire locale des CU.

Dans ce cas de figure, la taille et le nombre des CU deviennent des paramètres importants du GPU. Le GPU utilisé a 4 CU de 32 cœurs chacun.

4.7 Perspectives et conclusion

4.7.1 Discussion : portabilité vers l'embarqué

L'utilisation de la radio logicielle dans les systèmes embarqués fait l'objet de plus en plus d'études. L'utilisation du multimode sur les terminaux mobiles serait un gros avantage pour la téléphonie sans fil, les *smartphones* pourraient suivre les évolutions des normes, et mieux encore, utiliser n'importe quelle norme sans avoir besoin de matériel dédié.

Parallèlement, des GPU pour les applications embarquées commencent à voir le jour, dont certains comme le Mali T604 de ARM supportent l'OpenCL. Les plateformes ION de NVidia pour les ultra-portables sont un autre exemple.

Les résultats présentés dans cette étude sont prometteurs. Cependant, on utilise ici un processeur relativement puissant, qui suffit pour exécuter la plupart des applications radio. Dans le cadre des applications embarquées, le processeur est beaucoup moins puissant et ne permet pas d'envisager une exécution logicielle. Les DSP sont une solution possible, mais l'utilisation d'un GPU embarqué pourrait permettre d'exécuter plus d'applications.

Il n'y a pas encore au moment de la rédaction de ce mémoire de GPU compatible OpenCL (ou environnement équivalent) conçu pour l'embarqué. Le Mali T604 [arm], annoncé en 2010 [mal], n'est pas encore fondu. Il est donc difficile de savoir si l'étude réalisée durant cette thèse sera portable sur ce GPU. On peut cependant émettre certaines hypothèses.

Les contraintes de l'embarqué pourraient faire diminuer le nombre de *cores*. Les spécifications accessibles du Mali parlent d'une architecture basée sur un nombre variable de cœurs, entre 1 et 4. La notion de cœurs est floue. Si le cœur est un *core* NVidia, alors entre 1 et 4 cœurs indiquent un très petit GPU. Si on table sur quatre *cores*, au vu de l'effet du multitâche qui diminuait le nombre de *cores* accessibles par opération, on peut estimer que l'approche à gros grain permettrait toujours un gain par rapport au CPU, surtout que ce dernier est moins efficace. Si le cœur est un

unit OpenCL, ce qui parait plus plausible, alors l'étude réalisée dans ce chapitre reste également valable.

La place limitée, et la limite de consommation d'énergie, risquent également de pousser vers une diminution des fréquences, et surtout une absence de mémoire dédiée au GPU. On peut supposer que l'intégration du GPU avec le CPU sera très forte, avec une mémoire partagée et des caches (comme pour le Mali 400). Ceci pousse vers une limitation de la mémoire utilisée, qui imposerait alors de diminuer les seuils ayant pour conséquence une forte détérioration des performances.

Dans l'ensemble, il est assez difficile de savoir quel sera le résultat sur un GPU embarqué. Mais au vu des performances déplorables de la radio logicielle sur un CPU embarqué, il y a lieu de penser que l'utilisation du GPU pourrait améliorer ces performances. La question à laquelle il faudra répondre sera : ces performances seront-elles suffisantes pour implanter un vrai terminal ?

4.7.2 Conclusion

Nous nous sommes intéressés dans ce chapitre à la possibilité d'utiliser le GPU pour exécuter des applications de radio logicielle, et plus particulièrement des applications GNURadio. Afin d'étudier les éventuels avantages et inconvénients du GPU, trois points ont été abordés.

Dans un premier temps, nous nous sommes intéressés à la conception d'une opération exécutée par le GPU. L'architecture particulière d'un GPU et de ses applications impose une implantation différente par rapport à un CPU. La solution couramment utilisée, par exemple dans [Mil10], s'applique principalement sur des opérations basiques. Nous avons proposé deux solutions pour des opérations plus complexes. L'une, baptisée approche à grain fin, se base sur des optimisations des opérations pour le GPU, avec des *kernels* petits. Elle est une extension de la méthode classique. L'autre est une méthode originale basée sur l'utilisation de gros *kernels* qui représentent toute l'opération. Cette approche dite à gros grain permet d'envisager une parallélisation même pour des opérations avec une dépendance de données.

Dans un second temps, nous nous sommes intéressés à l'ordonnancement, ou plus précisément au choix du moment de déclenchement de l'opération sur le GPU. La maximisation de l'utilisation nécessite en effet que beaucoup d'opérations soient lancées en parallèle. L'utilisation d'un seuil de déclenchement, qui définit combien d'échantillons doivent être prêts à être traités avant l'exécution par le GPU, permet de contrôler l'efficacité. L'effet de ce seuil sur la latence et la mémoire a été discuté, ainsi que des solutions possibles pour diminuer cet effet. Cet ordonnancement a ensuite été appliqué à deux intégrations possibles des opérations GPU, l'intégration distribuée et centralisée. La deuxième option, qui définit un bloc dans lequel toutes les opérations GPU seront réalisées, est une contribution de cette thèse. Elle permet en particulier de gérer le multitâche efficacement, de diminuer l'empreinte mémoire, et d'utiliser au mieux les différentes solutions d'ordonnancement proposées.

Finalement, une étude des communications entre les blocs GPU a été faite. Dans le cas d'une intégration centralisée, ces communications sont simples. Un *buffer* conçu pour les opérations OpenCL a été défini dans le cas d'une intégration distribuée, qui évite les transferts inutiles de GPU à CPU. Ceux-ci sont en effet coûteux.

L'étude des résultats expérimentaux montre que l'approche à gros grain proposée permet d'obtenir une légère amélioration de performance dans le cas des opérations unitaires. Pour les applications multitâches, l'utilisation du *buffer* OpenCL proposée permet une amélioration plus marquée dans le cas d'une intégration distribuée, alors que l'intégration centralisée permet une utilisation très efficace du GPU, avec une empreinte mémoire et une latence minimisées. L'utilisation de la nouvelle approche permet également une amélioration de l'implantation du filtre IIR, contribution intéressante de ce travail. On notera que ces résultats sont vrais pour le GPU utilisé. Il serait intéressant de regarder l'évolution des résultats sur d'autres GPU, plus ou moins perfor-

mant, afin de se faire une idée plus précise de l'influence des paramètres du GPU (nombre de cœurs, fréquence, taille d'une unité...) sur les résultats obtenus.

L'intérêt du GPU sur la plateforme utilisée n'est toutefois pas démontré ici, et ce pour deux raisons principales. Tout d'abord, le gain en performance n'est pas spectaculaire, même si la carte utilisée pour les expérimentations est ici une carte graphique de faible puissance, alors que le CPU est relativement performant (la consommation électrique est la même, d'après les spécifications). Un gain d'un facteur 4 reste cependant intéressant, même si au vu des 128 cœurs du GPU, on aurait pu espérer mieux. La libération du CPU est un atout, mais la complexité d'une application à base de GPU limite son intérêt. Cependant, le GPU peut être envisagé comme une cible de secours, qu'il n'est pas nécessaire d'ajouter dans une plateforme pour la radio flexible, mais qu'il est dommage de ne pas utiliser si elle est disponible.

Le second point noir de l'utilisation du GPU est la latence. Dans cette étude, nous nous sommes intéressés principalement à la puissance de calcul atteignable. Cependant, pour atteindre un débit symbole plus intéressant qu'avec un CPU, les latences envisagées sont inacceptables pour la plupart des normes réseaux. Si dans les protocoles de la couche MAC, une réponse est requise (un acquittement, par exemple) dans un temps contraint, il est fort probable que ce temps sera dépassé en utilisant le GPU. Cependant, ce dernier conserve un intérêt dans le cas d'applications de type *Digital Video Broadcasting* (DVB), qui sont des applications de diffusion, sans retour possible de la part du récepteur. L'utilisation de GPU dans le contexte embarqué pourrait également changer la donne, en diminuant le nombre de cœurs et donc la latence.

Chapitre 5

Définition d'un environnement pour la radio flexible

COMME nous l'avons évoqué au chapitre 2, les multiples possibilités d'implantation de la radio flexible sont un frein au développement d'algorithmes utilisant cette flexibilité. Dans ce chapitre, nous présentons un environnement conçu pour abstraire ces multiples implantations afin d'offrir une interface unique aux utilisateurs de la radio flexible.

5.1 Introduction **62**
5.1.1 Contexte 62
5.1.2 Objectifs 62
5.2 Architecture générale **64**
5.2.1 Présentation générale de l'environnement 64
5.2.2 R-HAL et traducteur 65
5.2.3 Couche protocolaire 66
5.3 Gestion des applications **67**
5.3.1 Présentation : flot de développement d'une application avec FRK . . . 67
5.3.2 Représentation d'une application 68
5.3.3 Traduction et implantation de l'application 68
5.3.4 Gestion de l'exécution 71
5.4 Implantation des cibles **73**
5.4.1 TaME et extensions 73
5.4.2 Cible logicielle 74
5.5 Intégration des coprocesseurs **75**
5.5.1 Définition et postulat 75
5.5.2 Représentations 76
5.5.3 Fonctionnement de la cible coprocesseur 77
5.6 FRK en pratique **80**
5.6.1 État de FRK 80
5.6.2 Implantation 81
5.6.3 Exemple du codeur convolutif 81
5.6.4 Exemple complet : IEEE 802.11 87
5.6.5 Quelques chiffres 89
5.7 Conclusion **89**

5.1 Introduction

5.1.1 Contexte

La radio flexible telle qu'elle a été définie dans [DZD+05] regroupe toute implantation évolutive d'un terminal radio. On retient généralement deux propriétés : l'adaptabilité qui concerne des changements de paramètres numériques, et la reconfiguration qui concerne des changements dans la structure même de l'application. Ces propriétés peuvent s'obtenir de différentes manières, par exemple :

- en utilisant une implantation logicielle de l'application, qui s'exécutera donc sur un ou des processeurs, spécialisés ou non ;
- en utilisant des accélérateurs matériels paramétrables ;
- en utilisant des FPGA, qui permettent de modifier l'architecture de la plateforme d'exécution afin de l'adapter à l'une ou l'autre des applications.

Les différents éléments matériels sont reliés en utilisant une méthode d'interconnexion, classiquement un bus matériel, ou un *Network on Chip* (NoC) comme pour les plateformes Magali [NKM+09] et Leocore [LNT+09]. Dans ce chapitre, on s'intéresse à une plateforme générique présentée dans le chapitre 2, que l'on présente à nouveau dans la figure 5.1 pour simplifier la lecture.

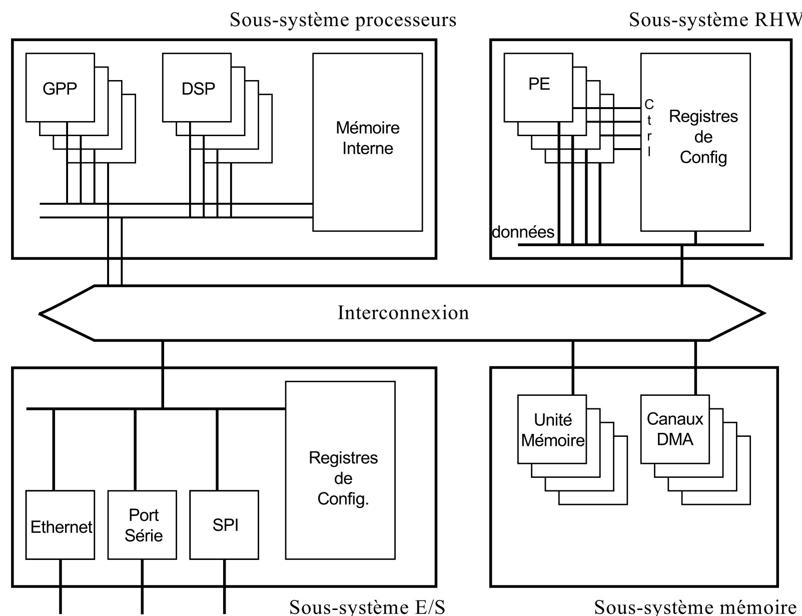


FIGURE 5.1 – Architecture de plateforme flexible générique

On donnera des cas concrets d'utilisation de cette plateforme dans la section 5.6.

5.1.2 Objectifs

Ce chapitre a pour objectif de présenter un environnement permettant de résoudre les problèmes liés à l'unification des plateformes. Le but principal est donc de permettre à un utilisateur de définir et d'utiliser une application sans se soucier de la plateforme qui l'exécutera ce qui regroupe deux aspects :

- le support de la plateforme, qui est nécessaire à son abstraction ;
- la spécification d'une application.

5.1.2.1 Support des unités d'exécution

Tout d'abord, comme nous l'avons évoqué dans l'état de l'art dans le chapitre 3, la plupart des environnements de radio flexible actuels sont en fait des environnements de radio logicielle, qui ne proposent pas de support pour les coprocesseurs dédiés. L'intégration de ces coprocesseurs, quand elle est disponible, se fait en utilisant une encapsulation logicielle. Cette encapsulation permet de présenter à l'environnement une interface logicielle classique, mais qui gérera en fait la configuration d'un élément matériel réalisant le calcul. Les communications avec le composant matériel sont également gérées en utilisant la surcouche logicielle. Cette méthode est fonctionnelle, mais pas complètement satisfaisante, principalement pour des raisons de performance.

En effet, le surcoût généré est important, et l'utilisation du coprocesseur n'est pas optimale. L'utilisation de l'interface périphérique (*device*) d'un noyau POSIX par exemple force l'utilisation d'appels systèmes, qui détériorent les performances et sont de plus non déterministes. En se passant de système d'exploitation, si on conserve une interface logicielle, on limite les performances du coprocesseur, puisque celui-ci est tributaire du logiciel.

5.1.2.2 Application, adaptabilité et reconfigurabilité

Un deuxième point important d'un environnement de radio flexible concerne la gestion des applications radio. Même si les applications sont gérées dans les environnements existants, certaines améliorations peuvent être apportées.

Une application radio est de manière générale donnée sous forme de graphe, avec chaque nœud représentant une opération à effectuer, et les arrêtes représentant les communications entre les opérations. Lors de l'écriture d'une application pour un environnement, on définit ce graphe avec les opérations implantées en utilisant l'API spécifique de l'environnement. Dans le cas de SCA, par exemple, les opérations sont des composants avec des interfaces CORBA, qui utilisent ou étendent des objets définis dans l'environnement. Dans le cas de GNURadio, les opérations sont des classes C++, qu'il faut instancier et connecter pour former le graphe.

Dans ce chapitre, trois points principaux de la gestion des applications sont étudiés. Tout d'abord, la définition d'une application sur une plateforme hétérogène pose le problème de sa représentation. Il faut en effet pouvoir représenter l'application indépendamment de la plateforme, et pouvoir la traduire pour qu'elle soit compréhensible et exécutable par la plateforme.

Lorsqu'une application est écrite et chargée en utilisant un environnement précis, il faut pouvoir modifier son fonctionnement afin de supporter la flexibilité. Dans les environnements actuels, ceci se fait en rechargeant une nouvelle application et en déchargeant l'ancienne. Cependant, dans le cas où la modification est une modification mineure, pour adapter par exemple le débit aux conditions du canal, reconfigurer complètement une application est excessif. L'un des objectifs de ce chapitre est de définir une méthode permettant une modification mineure de l'application sans nécessairement modifier toute l'application. Cette modification doit pouvoir s'appliquer de manière générique sans connaissance de la plateforme, pour respecter l'objectif d'unification.

L'autre point important de la gestion des applications est la possibilité de créer un terminal multimode. Ce type de terminal permet d'exécuter en concurrence plusieurs applications radio en utilisant la même plateforme. Le multimode soulève tout d'abord un problème de performance qui se partage entre la définition de la plateforme capable de supporter plusieurs applications, et la possibilité pour l'environnement de savoir s'il pourra exécuter une application donnée ou pas. Le premier point n'est pas abordé, le deuxième est évoqué dans cette étude. Mais permettre l'exécution concurrente des applications radio impose également de les gérer correctement, afin de les faire cohabiter et d'utiliser au mieux la plateforme disponible.

5.2 Architecture générale

Afin de répondre aux objectifs que nous nous sommes fixés, nous présentons ici un nouvel environnement de radio flexible baptisé FRK.

5.2.1 Présentation générale de l'environnement

FRK est un environnement de radio flexible basé sur une architecture classique en couches, chaque couche permettant de gagner un niveau d'abstraction. Au contraire des environnements actuels, il est conçu pour être un environnement de radio flexible au sens large, et non un environnement de radio logicielle. Il permet en effet de gérer des implantations matérielles, qu'elles soient basées sur des FPGA ou sur des ASIC. FRK est construit comme un ensemble de librairies, au même titre qu'un noyau. Le choix de faire appel à FRK pour les différentes tâches est donc laissé au développeur.

Son intégration avec le système d'exploitation est plus forte que pour les autres environnements. Contrairement à SCA, qui se construit en surcouche d'un noyau POSIX, ou à GNURadio, qui n'est qu'une application s'exécutant dans un système GNU/Linux, il ajoute des éléments qui pourraient s'intégrer dans le noyau, ce en quoi il s'apparente à Aloe.

FRK est construit autour d'une partie liée à l'exécution (que l'on appelle le *runtime* [1]), et d'une partie permettant de gérer tous les calculs s'effectuant en dehors de l'exécution (*offline*).

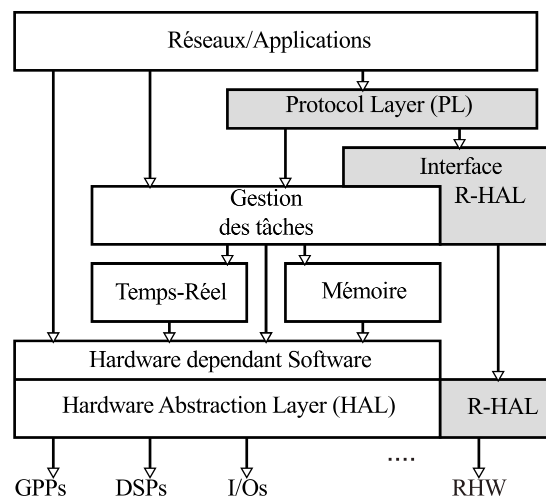


FIGURE 5.2 – Intégration de FRK dans une architecture logicielle classique

L'intégration du *runtime* dans le noyau est présenté dans la figure 5.2. Dans le *runtime*, on distingue deux couches, qui correspondent grossièrement aux couches du modèle OSI :

– la *Protocol Layer* (PL) est la couche de plus haut niveau. Elle permet d'implanter les protocoles d'accès au medium, généralement concentrés dans la couche MAC. Cette couche n'est que rapidement abordée dans ce document ;
– la couche permettant l'exécution des applications radio, appelée *Radio Hardware Abstraction Layer* (R-HAL), qui fait l'interface avec la couche protocolaire, et qui gère le chargement et le déchargement des applications. Elle correspond à la couche PHY du modèle OSI. Elle s'intègre dans le noyau au niveau de la *Harware Abstraction Layer* (HAL), afin de fournir le support pour les coprocesseurs, et au niveau de la gestion des tâches, afin de permettre l'exécution logicielle.

1. par abus de langage, FRK étant construit comme une librairie

La partie *offline* permet principalement de traduire une application pour la plateforme sur laquelle on va l'exécuter.

5.2.2 R-HAL et traducteur

La R-HAL permet de gérer l'exécution et la configuration de la plateforme afin de permettre le déploiement des applications radio. La figure 5.3 donne une représentation des différents éléments composant la R-HAL. On distingue donc trois rôles principaux pour cette couche.

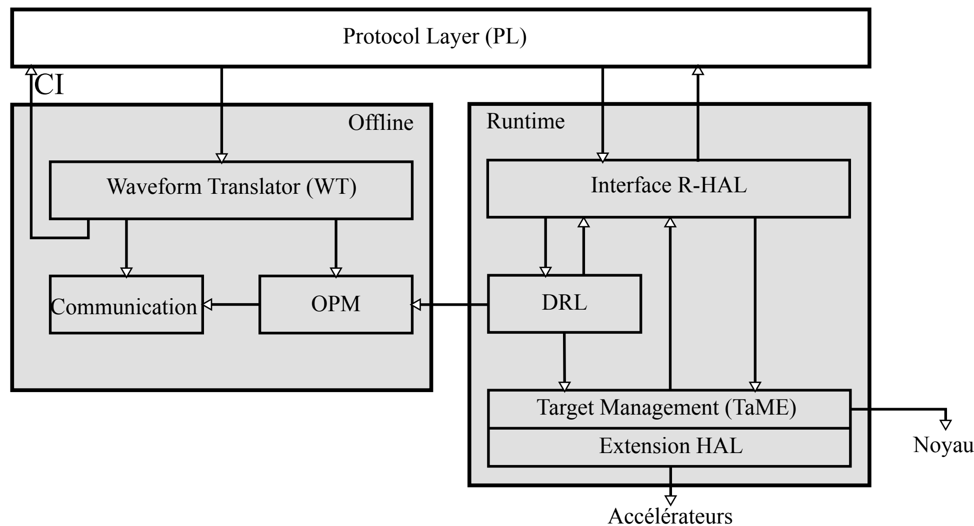


FIGURE 5.3 – Intégration de FRK dans une architecture logicielle classique

5.2.2.1 Représentation et contrôle de la plateforme

La R-HAL permet de représenter la plateforme. La particularité de FRK, qui est à la base de sa conception et de laquelle découle son architecture, est sa gestion originale des différentes unités d'exécution. Une plateforme exécutant FRK est représentée sous la forme de cibles (*targets*). Une cible est un ensemble d'unités d'exécution pour lesquelles l'implantation est similaire. Par exemple, toute plateforme exécutant FRK supporte la cible GPP. De même, on peut définir une cible *Open Computing Language* (OpenCL), qui englobe les GPU, les GPP et les DSP avec un support OpenCL. Finalement, on peut définir comme cible les coprocesseurs matériels tels qu'ils seront présentés dans la section 5.5.

Les cibles sont gérées dans la partie basse de la R-HAL. Cette gestion est réalisée par le *TArget Management Element* (TaME). On peut aussi avoir besoin de définir des extensions de la HAL qui sont propres à FRK. Elles permettent de représenter les cibles non-gérées par le système d'exploitation, et également de s'interfacer avec celui-ci. On revient plus loin sur les spécificités de ces différents éléments et de la gestion de certaines cibles. Le but est de permettre d'utiliser et de configurer les différentes unités d'exécution présentes sur la plateforme.

5.2.2.2 Traduction

Le *Waveform Translator* (WT) permet la traduction d'une application définie de manière générique pour qu'elle soit exécutable sur la plateforme. Bien que ne faisant pas réellement partie de la R-HAL, elle y est rattachée indirectement puisqu'elle doit s'exécuter de manière cohérente avec le *runtime* de FRK.

Le WT est la partie *offline* de FRK. Cette partie est basée sur deux librairies permettant le portage de l'environnement sur la plateforme. La première librairie est l'*Operation to Platform Mapping* (OPM), qui traduit une liste d'opérations prédéfinies en configurations pour la plateforme. Ce mécanisme est expliqué plus précisément dans la section 5.3.

La deuxième librairie concerne les échanges de données entre les opérations. La librairie implante des canaux de transferts pour les différentes cibles présentes sur la plateforme. Ainsi, par exemple, on trouvera un canal de communication logiciel pour les communications entre tâches logicielles. De même, un canal de type DMA pourra être implanté pour communiquer avec les accélérateurs matériels. Le canal de communication OpenCL défini dans le chapitre 4 est un autre exemple d'implantation.

La partie *offline* interagit en fait avec les couches basses de la R-HAL. Le port de FRK sur une nouvelle plateforme passe par la définition d'une R-HAL supportant cette plateforme, et par l'écriture des librairies (OPM et communications) pour chaque cible de la plateforme.

5.2.2.3 Gestion des applications

La gestion des applications est également réalisée dans la R-HAL, et implique deux types d'actions.

La première est le rôle d'ordonnancement et d'interface, et est pris en charge par la partie d'interface visible sur la figure 5.3. L'objectif est de permettre le chargement et le déchargement d'applications traduites pour la plateforme, et d'assurer leur bon fonctionnement. Cette partie est également responsable de faire remonter aux différents appelants les problèmes éventuels rencontrés lors de l'exécution de la plateforme. L'ordonnancement entre les différentes applications permet d'assurer que toutes les applications actives s'exécutent correctement.

Le second objectif est de permettre la modification dynamique de l'application. Plus précisément, on veut pouvoir répercuter des changements effectués sur l'application haut niveau. Ceci est effectué par la Dynamic Reconfiguration Layer (DRL), qui fonctionne en lien avec la partie *offline* de FRK.

5.2.3 Couche protocolaire

La couche protocolaire (PL) a pour objectif de faciliter l'implantation de protocoles de type MAC. Elle n'est actuellement pas encore complètement définie. La figure 5.4 donne l'architecture actuellement envisagée.

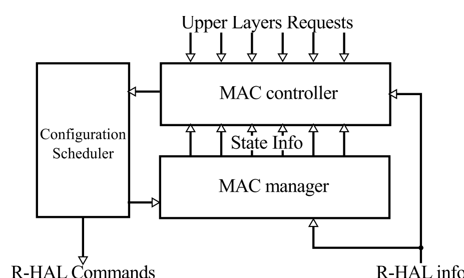


FIGURE 5.4 – Architecture de la PL

Le *MAC manager* (MM) est l'élément dans laquelle les couches MAC s'exécutent. Pour chaque norme qui va s'exécuter dans l'environnement, des couches MAC sont définies. Ces couches MAC sont implantées sous la forme de tâches logicielles. Chacune des tâches est associée à une application de la R-HAL, qui permet la communication. Il est également envisagé que la PL ait une vision des cibles (TaME). Les protocoles MAC peuvent en effet être implantés en matériel, certains protocoles étant normalisés. L'exemple du CSMA/CA, utilisé dans les normes

IEEE 802.11 et IEEE 802.15.4, en est une illustration. Cependant, intégrer ces éléments peut se révéler problématique, surtout pour les appels à la R-HAL. Ce problème n'est pas encore résolu et fait partie des perspectives de ce travail.

Les couches MAC sont gérées par le *Configuration Scheduler* (CS), c'est-à-dire l'ordonnanceur des configurations. C'est l'élément central de la PL. Cet ordonnanceur est l'ordonnanceur de plus haut niveau de FRK, qui active ou désactive les normes déployées dans l'environnement.

Cette (dés)activation se fait sur la base des éléments s'exécutant dans le contrôleur MAC (*MAC controller*, MC). Le MC est l'entité dans laquelle les différents algorithmes faisant usage de la radio flexible s'exécutent. Il récupère des informations de la R-HAL et des couches MAC en cours d'exécution, et prend ses décisions sur l'activation ou la désactivation des différentes normes, ou sur la nécessité de reconfigurer ou de s'adapter. Une implantation d'algorithmes propres à la radio cognitive pourrait se faire dans le MC.

5.3 Gestion des applications

5.3.1 Présentation : flot de développement d'une application avec FRK

Les applications dans FRK s'écrivent en utilisant l'API offerte par la librairie OPM. L'API générale de FRK est donnée dans l'annexe C. L'écriture de l'application se fait selon la vision classique de type graphe. On notera l'existence d'une vision différente, appelée algébrique par son auteur, dans laquelle on ne donne pas la séquence d'opérations avec les transferts de données à effectuer, mais la fonction de transformation à appliquer aux échantillons reçus [Dic]. Le flot de développement pour FRK est présenté dans la figure 5.5.

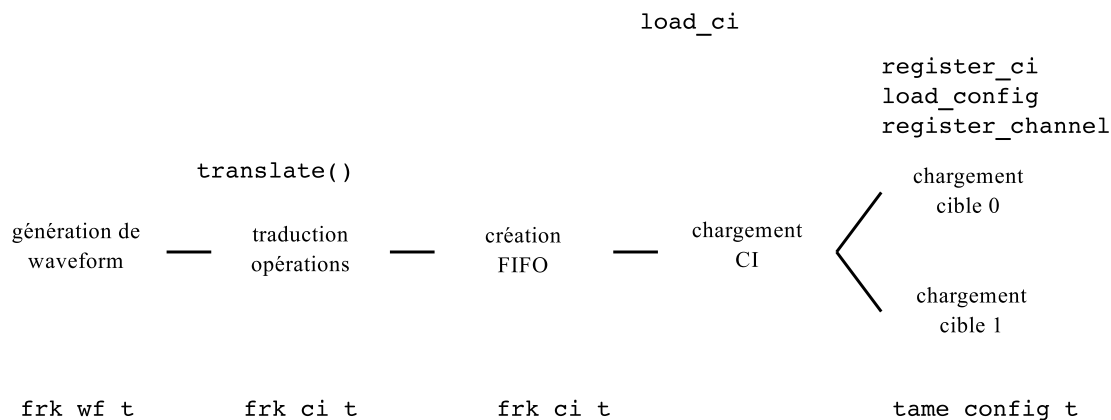


FIGURE 5.5 – Étapes de chargement d'une application FRK

L'application est d'abord écrite en utilisant une des méthodes disponibles (pour l'instant, en écrivant un code d'initialisation). Cette application de haut-niveau que l'on appellera *waveform* par analogie avec l'application SCA est ensuite traduite pour la machine sur laquelle elle va s'exécuter. Cette traduction se fait en utilisant la partie *offline* de FRK. Bien qu'elle ait lieu en dehors du *runtime*, la traduction a tout de même lieu juste avant le chargement, et se fait au moment de l'exécution de la *waveform*. Le processus de traduction est présenté dans la section 5.3.3. Une fois la *waveform* traduite, on obtient une *Configuration Instance* (CI), qui est en fait l'application que l'on peut charger dans la R-HAL. La *waveform* sera utilisée dans la PL ou dans les couches supérieures, afin de contrôler l'application. La CI sera conservée dans la PL, mais utilisée dans la R-HAL. Une fois la CI obtenue, l'étape suivante est le chargement de l'application sur la plateforme.

Quand la CI est chargée, elle doit être gérée par la R-HAL qui est responsable de l'ordonnancement des différentes CI et des opérations de la CI. Elle peut également être modifiée (adaptée), activée ou désactivée. Cette modification se fait au niveau de la *waveform*, et est ensuite répercutée sur la CI. Ces différents mécanismes de gestion des applications sont présentées dans les parties suivantes.

5.3.2 Représentation d'une application

On distingue dans FRK deux niveaux de représentation de l'application. Le premier niveau est la *waveform*, portable sur toutes les plateformes pour lesquelles FRK est disponible, et utilisée dans la PL. Le second niveau est l'application traduite (la CI), spécifique pour une plateforme.

5.3.2.1 *Waveform*

La représentation en graphe des applications permet d'envisager plusieurs modes de développement. Elle se prête plutôt bien à l'utilisation d'une interface graphique, mais peut également être mise en place simplement à l'aide de fichiers contenant du code ou une représentation du graphe à l'aide d'un langage de description. Dans l'état actuel de FRK, c'est l'approche de GNURadio qui est utilisée. La représentation se fait avec un fichier contenant du code. Les blocs correspondant aux opérations sont instanciés, puis connectés explicitement à l'aide d'une fonction. L'application est donc écrite en utilisant un programme, qui permet de générer la *waveform* proprement dite.

L'OPM offre aux utilisateurs un ensemble d'opérations prédéfinies. Ces opérations peuvent avoir différentes granularités. On peut ainsi avoir des opérations simplistes, du type addition ou multiplication. On peut également avoir des opérations plus complexes, comme par exemple des démodulations d'amplitude ou de fréquence. Finalement, on peut aussi avoir des opérations sur des vecteurs comme la transformée de Fourier, ou encore un décodeur Viterbi.

Une opération est définie avec ses paramètres. On distingue deux types de paramètres :

- les paramètres liés à FRK, qui sont obligatoires. Ces paramètres sont l'identifiant de l'opération, le nombre de canaux en entrées et le nombre de canaux en sorties, ainsi que les blocs avec lesquels il y a connexion. Le débit symbole requis pour l'opération fait également partie des paramètres, même s'il n'est pas encore utilisé ;
- les paramètres liés à l'opération, qui peuvent varier. Dans le cas d'une opération d'amplification, par exemple, on a besoin de connaître la constante d'amplification. Pour la FFT, on peut utiliser le nombre de points comme paramètre.

La *waveform* est une structure qui contient les différentes opérations de l'application.

5.3.2.2 *Configuration Instance*

La CI est l'application traduite pour la plateforme. Elle contient deux types d'information. Tout d'abord, des informations d'état, en particulier le chargement et l'activation. Elle contient aussi une référence vers la *waveform* dont elle est issue. Mais la CI contient également les informations sur les opérations instanciées. Pour chaque cible de la plateforme, l'ensemble des instances d'opérations à exécuter sur cette cible est donné. On marque également les entrées et sorties de l'application, qui sont utilisés pour des opérations spécifiques comme la désactivation. Les communications entre les opérations sont aussi disponibles.

5.3.3 Traduction et implantation de l'application

La traduction de l'implantation, comme précisé dans l'explication du flot de développement de FRK, se fait en deux phases : d'abord les opérations puis les communications.

5.3.3.1 Opérations

La première étape permet de récupérer les différentes opérations qui sont utilisées dans l'application. Les opérations sont ensuite traduites pour être exécutables par la plateforme considérée à l'aide de l'OPM.

Chacune des opérations peut avoir différentes implantations en fonction des capacités de la plateforme. De même, en fonction des paramètres, une même opération peut ne pas être implantée de la même manière. Prenons une nouvelle fois l'exemple de la FFT. Sur une plateforme comportant un GPU, un GPP, et un coprocesseur matériel FFT, on peut théoriquement avoir trois implantations d'une même fonction. Cependant, le coprocesseur matériel n'est pas nécessairement capable de calculer une FFT quelque soit le nombre de points. De même, pour l'implantation logicielle, un algorithme de type radix-4 sera plus efficace qu'un algorithme de type radix-2, mais ne s'appliquera que quand le nombre de points est un multiple de 4.

Lorsqu'on traduit une opération, les paramètres initiaux de cette opération sont fixés. L'objectif de la traduction est de pouvoir extraire, pour un jeu de paramètre donné, les implantations pour les différentes cibles existantes.

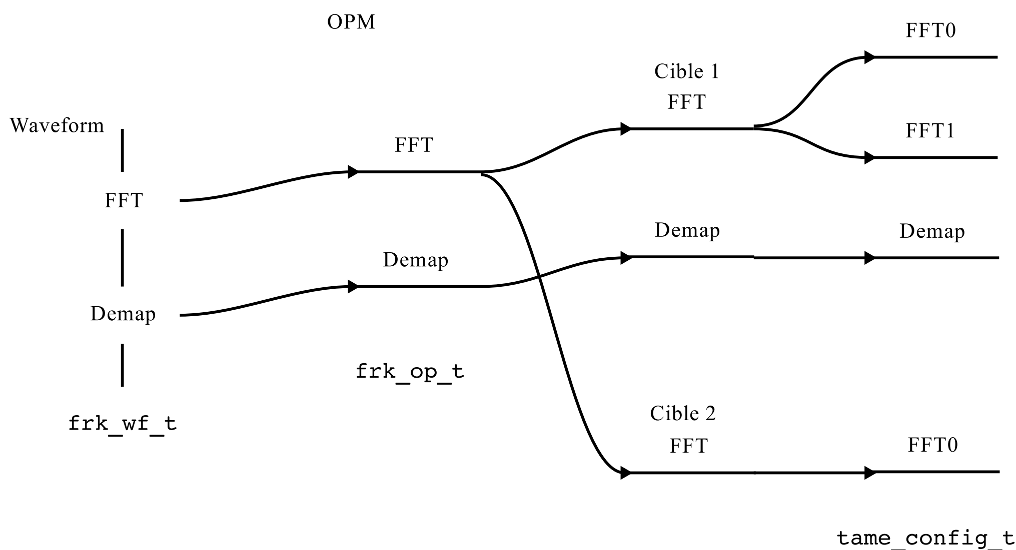


FIGURE 5.6 – Traduction d'une opération en utilisant l'OPM

Le processus de traduction d'une opération est représenté dans la figure 5.6. On donne également les types de données pour chaque étape, ces types étant décrits dans l'annexe C. Lors de l'initialisation de la plateforme, le nombre et le type de cibles sont connus, ainsi qu'un ordre de priorité pour ces cibles. Chaque opération a un identifiant unique (ID). Pour chaque cible, on définit les opérations qui seront exécutables ainsi que l'implantation qui correspond. Pour les cibles génériques indépendantes de la plateforme, comme par exemple les GPP ou la cible OpenCL, l'implantation est faite une seule fois. Pour les cibles dépendantes de la plateforme, c'est-à-dire les accélérateurs matériels, il faut faire l'implantation pour chaque plateforme. Cette implantation fait partie du portage de FRK sur la plateforme. Lors de la traduction, pour chaque opération, on récupère l'ID correspondant. Puis, on cherche toutes les implantations pour toutes les cibles de la plateforme. On sélectionne ensuite l'implantation pour la cible ayant la priorité la plus élevée. Les implantations se présentent sous la forme d'un tableau d'implantations et d'une fonction `select_config`. Cette fonction permet de sélectionner l'implantation correspondant au jeu de paramètres souhaité pour la cible.

5.3.3.2 Transferts de données

La deuxième étape de la traduction de l'application est la génération des canaux de transfert de données. L'implantation des communications dépend de l'ordonnancement que l'on veut adopter. Dans FRK, l'implantation des communications se fait au moyen de FIFO implantées pour chaque cible. L'utilisation de tampons de taille fixe a été envisagée au début de la conception de FRK, comme pour l'implantation centralisée du GPU. Cependant, cette approche n'a pas encore aboutie dans FRK, du fait de sa complexité au niveau de l'ordonnancement, et de sa faible évolutivité. L'utilisation des FIFO génère un surcoût lié au contrôle, principalement en logiciel, mais permet une plus grande flexibilité dans l'ordonnancement, et autorise certaines libertés sur l'exécution ou non d'une opération. Si on tarde à exécuter une certaine opération pour laisser la priorité à une autre, les données ne sont pas perdues. Les FIFO rendent également les opérations interchangeables facilement.

On définit deux types de FIFO, les FIFO dédiées et les FIFO génériques. Dans une FIFO générique, les données sont localisées dans la mémoire centrale du GPP principal. La FIFO générique est implantée au niveau de la R-HAL, et la mémoire nécessaire est donc localisée à ce niveau là. Pour chaque cible, on définit des opérations de synchronisation, qui permettent effectivement d'effectuer le transfert dans la cible. Les opérations de contrôle, comme la purge ou la vérification de l'état, sont communes au niveau de la R-HAL. Cette implantation a l'avantage de permettre une modification facilitée de la cible de l'opération. L'ordonnanceur des applications, que l'on décrit plus précisément dans la section 5.3.4, permet en effet de passer d'une cible à une autre en fonction de la charge de la cible, comme dans l'environnement Surfer [DLD11]. Elle permet aussi d'envisager d'utiliser une implantation double, pour les goulots d'étranglement de l'application. Cette implantation double a été expérimentée sur FRK, mais reste pour l'instant compliquée à mettre en place. Par contre, l'utilisation de FIFO génériques n'est pas toujours optimale. Si on reprend l'exemple de la cible OpenCL, l'étude du chapitre 4 montre que faire transiter les données par la mémoire centrale est à proscrire.

Les FIFO dédiées de FRK sont principalement utilisées pour faire transiter les données au sein d'une même cible. À la différence des FIFO génériques, une FIFO dédiée laisse libre la localisation des données. Elle permet de diminuer le nombre de transferts, qui sont souvent coûteux. Une FIFO dédiée peut ne pas avoir d'implantation logicielle et être gérée automatiquement par le matériel. De même, les données ne sont pas nécessairement accessibles par l'environnement.

Dans les deux cas, chacune des cibles implante les interfaces de communication dont elle a besoin. L'interface est fixée et présentée en annexe C. On définit également une relation d'ordre entre les différentes cibles pour définir quelle sera l'interface utilisée. Ainsi, si on prend le cas de la cible des accélérateurs matériels, en cas de connexion avec une cible logicielle, on utilise les FIFO des accélérateurs matériels. De manière générale, les FIFO logicielles sont toujours les moins prioritaires. Lors de la traduction, le WT va instancier toutes les FIFO requises une par une. Ces FIFO sont sélectionnées en faisant appel à la fonction `select_fifo` de la cible prioritaire qui, comme pour `select_config`, sélectionnera la meilleure FIFO pour l'opération en fonction des cibles impliquées. Lorsqu'une FIFO est instanciée, les fonctions de lecture et d'écriture sont données au niveau de la CI, pour permettre aux opérations de savoir comment accéder à ses FIFO. Si on reprend l'application "FFT/demap" proposée dans la figure 5.6, avec la cible 2 prioritaire, le traducteur utilisera l'implantation FFT0 de la cible 2 pour la FFT, et l'opérateur "Demap" de la cible 1. Au moment de générer les FIFO, si on considère que la cible 2 est encore prioritaire, l'appel à `select_fifo` sélectionnera une implantation compatible pour une connexion avec une autre cible, et renseignera dans la structure de l'implantation FFT0 et demap les fonctions de lecture et d'écriture associée à cette FIFO.

5.3.3.3 Chargement d'une application

Une fois la *waveform* traduite, on obtient donc une CI, qui peut être chargée sur la plateforme. Le chargement se fait cible par cible, opération par opération, en utilisant la R-HAL. Pour chaque opération, on regarde la cible, et on appelle la fonction de chargement associée à cette cible. Les transferts de données sont gérés au niveau des cibles.

5.3.4 Gestion de l'exécution

La gestion de l'exécution d'une CI implique deux étapes :
- l'ordonnancement, qui gère comment la CI va s'exécuter sur la plateforme ;
- la modification de la CI, qui permet l'adaptabilité.

5.3.4.1 Ordonnancement

L'ordonnancement des applications radio dans FRK se fait à plusieurs niveaux :
- l'activation/désactivation des applications en fonction des normes requises est gérée au niveau de la PL ;
- le partage de la plateforme entre toutes les applications actives est géré au niveau de l'interface R-HAL ;
- le partage d'éléments spécifiques de la plateforme est l'une des tâches de la partie basse de la R-HAL qui gère les cibles.

Cette hiérarchie de l'ordonnancement, qui correspond à la hiérarchie générale de FRK, se rapproche de la vision hiérarchique de la reconfiguration évoquée dans le chapitre 3 et dans [DMLP05]. On se place ici au deuxième niveau de la hiérarchie. Le niveau 1 (le plus haut) est géré dans la PL, et fait intervenir des mécanismes plus proche de la radio cognitive, que l'on n'aborde pas dans cette étude. Le niveau 3 est géré spécifiquement en fonction des différentes cibles et sera abordé au cas par cas dans les sections suivantes.

L'ordonnanceur des CI implanté dans la R-HAL gère différentes actions. Tout d'abord, il répond aux ordres d'activation et de désactivation émis par la PL. Ces actions s'effectuent sur des CI chargées dans le système. La désactivation est différente du déchargement : la CI sera toujours présente dans le système, mais ne traitera plus d'échantillons. Afin d'effectuer la désactivation, on utilise l'algorithme 8. La désactivation des entrées permet de stopper les échantillons entrant dans le système. Le fait de marquer les opérations comme inactives va permettre, au niveau de la gestion des cibles, de forcer leur mise à l'écart lors du prochain traitement de cette opération par l'ordonnanceur de la cible. Cette action est faite en appelant la fonction de désactivation des CI `deactivate_ci`. On marque ensuite la CI comme inactive (même si les opérations ne sont pas forcément inactivées), pour information, et on sort la CI de la file des CI actives. Tout ceci se fait évidemment sous la protection des verrous associés, pour garantir l'atomicité.

Algorithm 8 Désactivation d'une CI

```
procedure DÉSACTIVATION(CI)
    Désactiver les entrées
    Marquer toutes les opérations comme inactives
    Marquer la CI comme inactive
    Mettre la CI dans la file correspondante
end procedure
```

L'activation suit la marche inverse. Les opérations sont marquées comme actives et remises immédiatement dans la file des opérations actives. Les sources sont activées en dernier, quand toute la CI est prête, afin d'éviter d'accumuler trop d'échantillons.

L'ordonnancement entre les CI se fait selon un mode de priorité établi par la PL. On distingue deux modes de fonctionnement dans l'ordonnanceur : le mode démon et le mode prioritaire. Une CI en mode démon traite des données en permanence en fonction de leur disponibilité. Elle n'a pas de contraintes de temps pour traiter ces données, seulement une contrainte de stabilité, c'est-à-dire que toutes les données doivent être traitées. Une CI en mode prioritaire est traitée immédiatement au détriment de toutes les autres. Ce mode prioritaire est mis en place au niveau des cibles. Quand une CI passe en mode prioritaire, toutes les opérations associées deviennent prioritaires sur les cibles. Dès que ces opérations ont suffisamment de données à traiter, elles sont exécutées, quel que soit l'état des opérations des autres CI s'exécutant sur cette cible. Le mode prioritaire peut perturber le système, et doit donc être utilisé avec parcimonie. Il est destiné à être utilisé ponctuellement, par exemple pour une transmission, ou pour le traitement de canaux de contrôle dont on sait quand la réception est prévue. Il n'y a pas actuellement de mode temps-réel dans FRK. Il est encore en cours d'étude, et devrait prendre la forme d'un ordonnancement basé sur les *deadlines* au niveau des cibles.

La gestion des calculs proprement dits de la CI se fait au niveau des opérations. Toutes les opérations d'une CI ne sont pas nécessairement actives au même moment, afin de maximiser l'utilisation de la plateforme. Si deux CI sont actives au même moment, il est en effet sous-optimal d'ordonnancer CI par CI, celles-ci n'utilisant pas nécessairement les mêmes cibles. Ordonnancer opération par opération permet de gérer la concurrence uniquement où il y en a. Chaque TaME implante donc son propre ordonnanceur.

5.3.4.2 Adaptabilité : modification dynamique de la CI

Une CI chargée n'est pas figée. Deux raisons principales peuvent pousser à une modification de la CI : un ordre de la PL ou un manque de ressource. L'ordre de la PL signifie une modification de la *waveform*. On parle ici d'adaptabilité : la modification doit toucher à un paramètre d'une opération existante, mais ne peut pas modifier la structure de la *waveform*, c'est à dire qu'elle ne peut pas changer d'opération, ni changer son interface. Le WT implante deux fonctions, `translate` et `update`. `translate` est utilisée pour traduire une *waveform* complète et générer une CI. La fonction `update` permet de retraduire uniquement une opération, sans modifier les canaux de transferts ni les autres opérations. On génère donc une nouvelle configuration pour l'opération, qui est remplacée dans la CI initiale. L'ancienne configuration vide la FIFO d'entrée et est supprimée de la liste des opérations pour l'ancienne cible. La nouvelle est insérée dans la liste des opérations pour la nouvelle cible.

Le manque de ressource est plus complexe à gérer. Dans la version actuelle de FRK, on parle de manque de ressource quand un canal de transferts atteint son seuil critique. Cette définition n'est pas optimale, mais on cherche juste à mettre en place le mécanisme permettant de traiter ce manque de ressources. En cas de détection, la DRL est appelée pour fournir une implantation de remplacement pour l'opération concernée. Si elle en trouve une, le même mécanisme que pour une adaptation a lieu, sauf que la FIFO d'entrée n'est pas vidée. Si elle n'en trouve pas, la DRL tente de trouver une autre implantation pour les autres opérations s'exécutant sur la cible saturée et modifie la CI concernée. Dans une prochaine version de FRK, il est envisagé d'implanter un système de priorité dynamique pour les opérations. Dans ce mode dynamique, l'arrivée à saturation d'une cible diminue sa priorité lors de la traduction.

5.4 Unités hétérogènes : implantation des cibles

5.4.1 TaME et extensions

5.4.1.1 Structure

Implanter une cible dans FRK revient à implanter le support pour un type d'unité d'exécution. L'implantation de ce support passe nécessairement par deux phases :

– une description de la cible, qui se fait au niveau des extensions de la HAL ;
– une mise en place du contrôle de l'unité, qui comprend entre autres les fonctions de chargement et de déchargement ainsi que l'ordonnanceur spécifique de la cible, qui se fait au niveau du TaME.

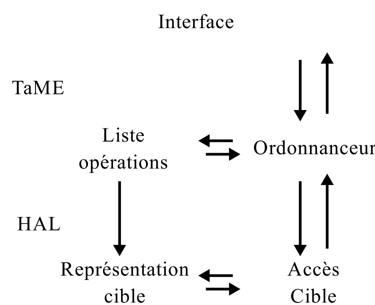


FIGURE 5.7 – Architecture général d'un contrôleur de cibles

L'architecture générale d'une cible est présentée dans la figure 5.7. Les couches basses de la R-HAL sont constituées d'un ou plusieurs TaME s'appuyant sur la HAL, qui peut être définie dans le noyau ou dans les extensions.

5.4.1.2 Interface : API principale

Afin d'être utilisable, un TaME doit implanter une API précise, qui permettra à l'interface de la R-HAL de fonctionner quelque soit la cible. L'API est assez limitée, les éléments de FRK étant très cloisonnés.

On distingue deux types de fonctions :

– les fonctions de modification d'opérations ;
– les fonctions de recherche d'informations.

Les fonctions de modification permettent le chargement, l'activation, la désactivation ou le déchargement des opérations. Les fonctions de recherche d'informations permettent de récupérer des informations sur l'état de la cible, sur les erreurs éventuelles, sur le volume de données traité, sur sa saturation éventuelle. La fonction de contrôle est couplée à une fonction offerte par l'interface de la R-HAL au TaME, qui permet de signaler qu'une cible est saturée ou que de l'information a été perdue. On présente en annexe C les structures principales et les fonctions de FRK.

On présente dans la suite de cette section l'implantation de la cible logicielle, c'est-à-dire les mécanismes mis en place pour pouvoir exécuter les opérations de manière logicielle. On se concentre sur la cible GPP, et une réflexion sur la cible OpenCL est donnée dans la fin de cette section. L'implantation pour les coprocesseurs matériels est abordée dans la section 5.5

5.4.2 Cible logicielle

5.4.2.1 Définition

La cible logicielle n'est pas implantée directement au-dessus du GPP, elle utilise le noyau comme HAL. Lorsque l'on parle de GPP, on ne parle donc pas du processeur en lui-même, mais du noyau. L'interface avec le noyau se fait en utilisant les tâches définies par le noyau.

Nous pouvons remarquer une première chose avant de détailler l'utilisation du noyau et l'implantation de la cible logicielle. La cible GPP, même si elle se représente de la même manière au niveau de l'OPM, dépend du noyau qui sera utilisé. Dans la version actuelle de FRK, deux noyaux sont supportés, Linux (ou n'importe quel noyau POSIX puisque seuls les appels POSIX sont utilisés) et RTEMS (pas dans sa version POSIX, bien sûr).

L'unité logicielle est donc représentée par le noyau du système d'exploitation. Implanter une opération pour l'unité logicielle se fait en donnant la fonction qui devra être exécutée. Cette fonction `process` doit avoir une interface simple, avec comme argument l'opération sous sa forme générique, afin de connaître les différents paramètres, ainsi qu'une zone mémoire en entrée et en sortie. Afin de permettre l'exécution de cette fonction, la cible utilise une méthode basée sur une tâche par opération. Cette méthode n'est pas la plus efficace en termes de performance, à cause du surcoût important lié à l'ordonnancement des tâches au niveau du noyau.

5.4.2.2 Transferts de données

Les transferts de données de la cible logicielle sont effectuées avec des FIFO logicielles bloquantes [2]. Les données sont physiquement localisées dans la mémoire centrale du processeur. On définit des fonctions `read` et `write` qui sont bloquantes en cas de manque de ressources. Ainsi, si une requête de lecture est faite alors qu'il n'y a pas suffisamment de données disponibles, le processus appelant se met en attente d'écritures sur la FIFO. De même, si une requête d'écriture est faite sans espace mémoire disponible, la tâche qui tente l'écriture se met en attente tant qu'il n'y a pas eu de lectures. Les FIFO utilisent la granularité minimale pour la taille de données. Ainsi, si une opération se fait sur un vecteur, l'élément de base de la FIFO ne sera pas le vecteur mais un élément du vecteur. C'est la tâche logicielle associée qui gère la présence d'un vecteur complet avant d'exécuter l'opération.

5.4.2.3 Ordonnancement

L'ordonnanceur de la cible logicielle est réparti entre la HAL (ou dans ce cas précis, le noyau), et le TaME. Au niveau du noyau, les tâches sont réparties entre les différents GPP existants. Elles sont également sélectionnées pour exécution selon la politique propre au noyau. L'ordonnanceur du TaME n'est finalement responsable que de l'activation et de la désactivation des différentes tâches. Dans le cas d'un ordonnancement à priorité, il peut également modifier les priorités des différentes tâches.

En l'état actuel de l'environnement, les opérations sont implantées comme présenté dans la figure 5.8. La fonction `process`, qui définit l'action effectuée par l'opération, est intégrée dans une tâche, qui va gérer la récupération des données et l'exécution du calcul. Le bloc grisé correspond à cette fonction. Les autres blocs sont génériques (et ne dépendent d'ailleurs pas du noyau employé). Le bloc d'interface en entrée et en sortie instancie pour chaque entrée (et sortie) de l'opération une structure représentant la FIFO associé. En particulier, cette structure `rhal_channel_t` permet de renseigner les fonctions de lecture et d'écriture associées aux FIFO entourant le bloc.

L'ordonnancement au niveau du TaME se fait en utilisant deux verrous, l'un associé à une CI, l'autre associé à la cible elle-même. Ces verrous permettent d'influencer l'ordonnanceur du

2. du même type que pour la *Kahn Process Network* (KPN)

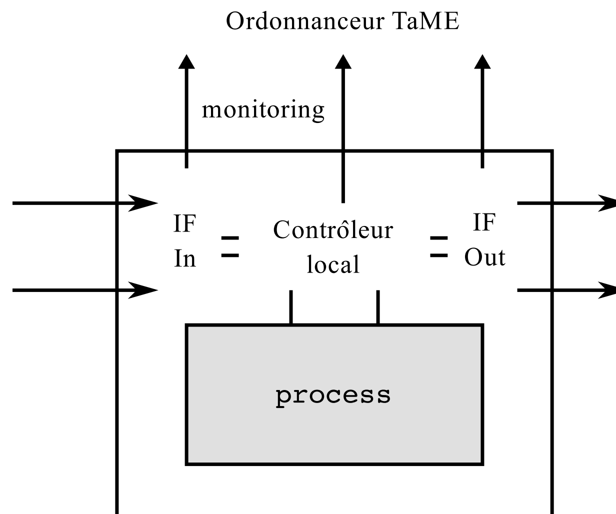


FIGURE 5.8 – Représentation d'une tâche pour la cible logicielle

noyau, en forçant une tâche à se mettre en attente. Chacune des tâches tente de prendre un verrou périodiquement, plus précisément, à chaque appel de la fonction `process`. Elle le libère immédiatement après. Afin de limiter l'impact de l'environnement, il est possible d'augmenter la période. Si le verrou est déjà pris, la tâche ne peut pas s'exécuter et est mise en attente. Les tâches vérifient toujours leurs deux verrous (celui de la CI et celui de la cible). Une tâche prioritaire ne vérifie que celui de la CI. La prise des verrous est faite par la PL ou par l'application utilisant la CI. Si le verrou de la CI est pris, alors la CI est désactivée, et les tâches associées sont mises en attente. Le verrou de la cible est utilisé pour les tâches prioritaires. Quand une CI est transformée en CI prioritaire (ou quand elle redevient démon), le verrou de la cible est pris ou relâché en conséquence, afin de forcer les tâches non prioritaires à se mettre en attente.

L'ordonnanceur est également protégé, c'est-à-dire que tout accès à l'ordonnanceur ne peut se faire que sous la protection d'un verrou propre à cet ordonnanceur. Cependant, ce verrou n'entre pas en jeu dans la gestion des opérations.

5.5 Intégration des coprocesseurs

5.5.1 Définition et postulat

On présente dans cette section les détails de la cible des coprocesseurs matériels. On définit les coprocesseurs ou accélérateurs matériels comme des unités de calculs paramétrables mais non programmables. On s'intéresse ici à des accélérateurs pouvant être paramétrés en utilisant des registres. Pour plus de simplicité, on considère également que ces registres sont accessibles en lecture et en écriture, en utilisant le plan d'adresses. Cependant, ce n'est pas une obligation, la méthode d'accès au périphérique étant dépendante de la méthode d'interconnexion utilisée. Ceci est vrai également pour les transferts de données. L'utilisation des interruptions est possible et doit donc être prise en compte. L'implantation du coprocesseur en lui-même est laissée libre, il peut par exemple être déployé sur un FPGA ou défini comme un ASIC.

5.5.2 Représentations

5.5.2.1 Coprocesseur

Dans FRK, les coprocesseurs sont représentés en utilisant les registres. On distingue deux types de registres. Certains registres du coprocesseur sont constants, et ne seront donc pas modifiés durant l'exécution par le coprocesseur. Ces registres sont généralement les registres de configuration. Les registres qui peuvent être modifiés par le coprocesseur sont des registres volatiles. Ils représentent généralement l'état du coprocesseur.

Pour représenter un coprocesseur dans FRK, on donne les adresses auxquelles sont accessibles les registres des deux types. Même si ces adresses ne sont pas dans le plan d'adresses général, il est possible de définir une adresse "locale" pour ce coprocesseur.

Pour illustrer la représentation des coprocesseurs, prenons l'exemple (hypothétique) d'un codeur convolutif à 3 additionneurs avec une mémoire de 3 éléments, présenté sur la figure 5.9. Dans ce codeur, on peut configurer le polynôme du code en donnant, pour chaque additionneur, les éléments mémoires qui seront utilisés ou non. Comme le code a trois éléments mémoire, les additionneurs ont 4 entrées possibles pour les bits mémorisés et le bit en entrée. On peut également poinçonner le code obtenu pour augmenter son rendement. Pour les registres volatiles, l'état des éléments mémoires est accessible dans un registre, et peut être chargé.

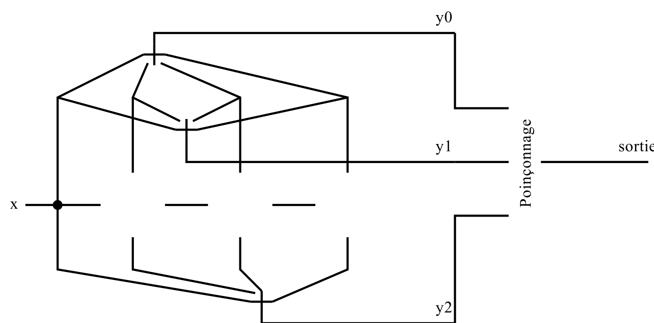


FIGURE 5.9 – Codeur convolutif matériel paramétrable

On se retrouve donc avec une représentation du coprocesseur en 5 registres, dont 1 volatile :
- trois registres pour activer les différentes opérandes de chaque additionneur (masque pour les entrées) ;
- un registre pour donner le schéma de poinçonnage en sortie du codeur ;
- un registre volatile pour l'état des bascules dans le codeur.

5.5.2.2 Opérations

Une opération exécutée par un coprocesseur est définie en donnant l'ensemble des valeurs pour les registres constants et volatiles du coprocesseur. Reprenons l'exemple du codeur convolutif dans lequel on veut implanter le code représenté en figure 5.10. Pour implanter ce code sur l'accélérateur matériel défini précédemment, il faut donc donner 5 valeurs de registre, une pour chaque registre existant. Les polynômes pour chaque additionneur (1000, 1011, 1110) sont d'abord données, puis la valeur de poinçonnage (ici, pas de poinçonnage, c'est donc une matrice de 1). Finalement, on initialise la valeur des mémoires à 000.

Dans la version actuelle de FRK, il faut avoir une correspondance exacte entre une opération et son implantation. Cela signifie que le traducteur n'est pas encore capable de gérer les coprocesseurs implantant les séquences d'opérations. Le codeur convolutif présenté ici implante en réalité deux opérations distinctes, l'encodage et le poinçonnage. Il faut donc définir une opération qui intègre les deux actions. Le seul cas où il n'y a pas nécessairement de correspondance exacte

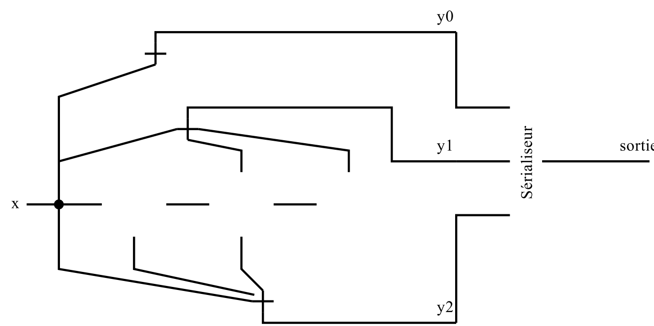


FIGURE 5.10 – Instance pour le codeur convolutif

est le cas où une opération est implantée en utilisant plusieurs implantations de la même cible. Par exemple, un couple codeur/poinçonneur défini comme une seule opération pourrait être implanté sur deux coprocesseurs, l'un pour l'encodage, l'un pour le poinçonnage. Cette opération ne peut pas encore être répartie entre plusieurs cibles.

5.5.3 Fonctionnement de la cible coprocesseur

5.5.3.1 Structure

Ces représentations de l'unité d'exécution et de l'opération à exécuter sont utilisées dans l'architecture présentée dans la figure 5.11. On la décrit ici pour un seul coprocesseur, le codeur défini précédemment.

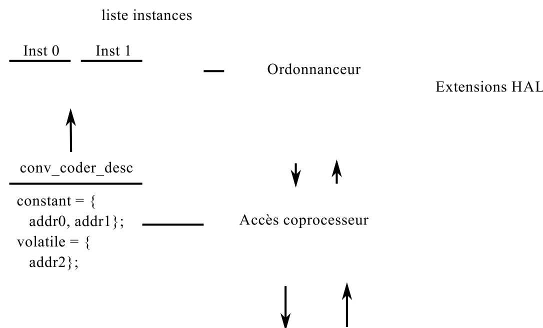


FIGURE 5.11 – Implantation de la cible pour le codeur

Le TaME pour les coprocesseurs regroupe l'ensemble des coprocesseurs d'une plateforme pour lesquels la méthode de contrôle est la même. Ceci permet de limiter le surcoût généré par le TaME, et d'éviter l'explosion du nombre de TaME. En effet, si plusieurs coprocesseurs sont disponibles et utilisés, il faudrait rajouter une cible par coprocesseur, ce qui pourrait entraîner un nombre de cibles important. On définit donc une seule cible, la cible coprocesseur, qui gérera l'ensemble des coprocesseurs. En interne du TaME, cependant, tout est géré coprocesseur par coprocesseur, sauf au niveau de l'interface avec les couches supérieures.

5.5.3.2 *Config Switch*

La configuration du coprocesseur est gérée par un mécanisme dit de *configuration switch* qui s'apparente à la commutation de contexte implantée pour les GPP. On dispose donc d'une configuration active sur le coprocesseur. Celle-ci est actuellement chargée. Lorsque l'ordonnanceur prend la décision de changer de configuration active, il sélectionne une nouvelle configuration.

Le principe de la commutation de configuration est simple. On arrête le coprocesseur, soit en bloquant sa FIFO d'entrée, soit en le désactivant si l'action est possible. On sauvegarde ensuite dans la configuration actuelle (c'est-à-dire dans sa représentation) les registres volatiles du coprocesseur. La configuration est ensuite remise dans la file des configurations. La nouvelle configuration, quant à elle, est activée. L'ensemble des registres est chargé dans le coprocesseur, qui est alors configuré pour la nouvelle opération. On active ensuite les transferts de données vers le coprocesseur.

5.5.3.3 Ordonnancement

L'ordonnanceur du TaME des coprocesseurs est un ordonnanceur distribué, qui s'applique pour chaque coprocesseur géré dans la cible. Chaque coprocesseur a une liste d'instances à exécuter, et une instance en cours d'exécution. La fonction d'ordonnancement peut être appelée dans trois cas :

- soit sur interruption du coprocesseur dans le cas où celles-ci sont gérées ;
- soit, comme décrit dans la suite, dans le cas où l'une des FIFO d'entrées passe un seuil (soit le dépasse, soit repasse en dessous) ;
- soit quand l'une des FIFO de sortie de l'instance courante est pleine, auquel cas cette instance ne peut plus s'exécuter puisqu'elle n'a plus de place pour enregistrer le résultat.

L'objectif de cette fonction est de sélectionner la configuration du coprocesseur.

La cible coprocesseur utilise un ordonnanceur basé sur l'état des canaux de communication en entrée, du même type que celui proposé dans [DDL10]. Le principe est assez simple. On définit pour chaque implantation un seuil minimal et un seuil maximal. Le seuil minimal est le seuil en deçà duquel on n'exécute pas l'opération. Le seuil maximal est le seuil au delà duquel l'opération doit être exécutée si possible. L'état d'une configuration est vérifié après chaque écriture et lecture dans une des FIFO d'entrées. La fonction d'écriture marque les opérations comme candidates (seuil maximal dépassé) ou exécutables (seuil minimal dépassé). La fonction de lecture ne marque pas les opérations, pour ne pas avoir à protéger le marquage (accès exclusif). L'ordonnanceur met lui-même à jour la fonction courante. Finalement, si l'écriture entraîne un dépassement du seuil maximal, ou la lecture du seuil minimal, l'ordonnanceur est appelé.

Le choix de l'opération à exécuter se fait en fonction des seuils, selon l'automate présenté dans la figure 5.12. L'ordonnancement se fait en deux parties : la sélection d'une opération `selected`, et la décision de reconfiguration. L'ordonnanceur commence par vérifier si des opérations ont atteint le seuil maximal (opération candidate). Si c'est le cas, il sélectionne la première opération candidate et s'arrête. Si ce n'est pas le cas, il cherche une opération ayant dépassé le seuil minimal (opération exécutable). Après la sélection, il vérifie l'état de l'opération courante et décide s'il faut reconfigurer ou non. Si l'opération courante n'est pas exécutable, et qu'une opération a été sélectionnée, il y a reconfiguration. De même, si l'opération sélectionnée est candidate, alors que l'opération courante n'est qu'exécutable, on reconfigure. Afin de limiter les changements de configuration, le choix par défaut est de ne rien faire.

L'algorithme 5.12 est donné pour une seule file. Pour implanter le mode prioritaire et le mode démon, l'ordonnanceur dispose de deux files, qui correspondent chacune à un des modes. Si une opération fait partie d'une CI prioritaire, elle est rangée dans la file appropriée. La file prioritaire est toujours traitée en priorité. Si elle n'est pas vide, aucune configuration de la file démon ne peut s'exécuter. La file des opérations en attente n'est bien évidemment pas traitée.

5.5.3.4 Transferts de données

Les transferts de données dans la cible des accélérateurs matériels sont implantés selon les deux méthodes définies précédemment, la méthode générique et la méthode dédiée. La méthode la plus simple est la méthode dédiée. Comme décrit précédemment, la localisation des données

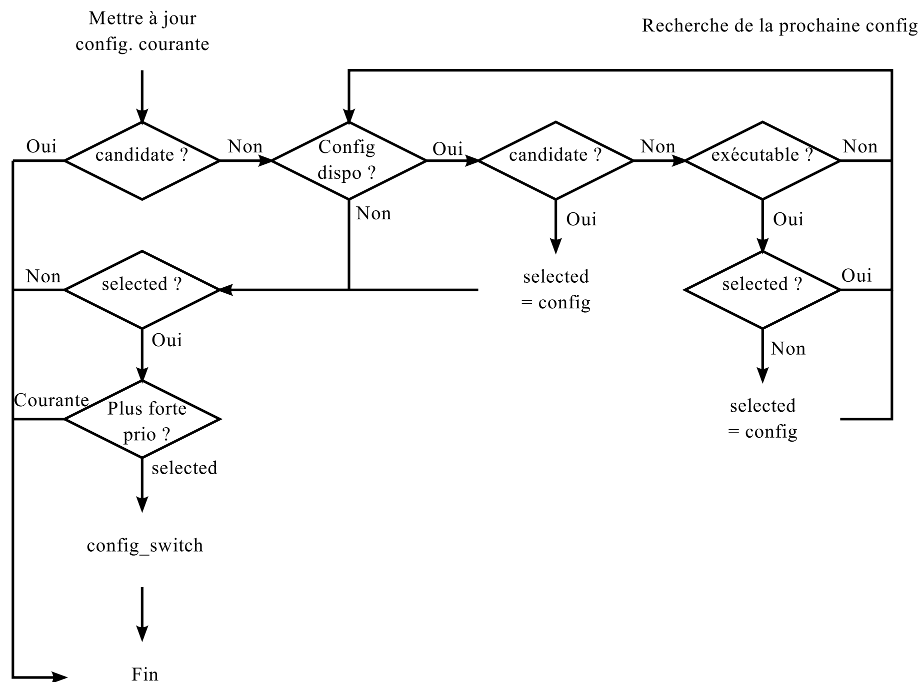


FIGURE 5.12 – Automate de sélection d'une configuration pour le TaME coprocesseur

n'est pas définie pour cette méthode. Elle s'implante quelque soit la méthode de transfert utilisée pour le coprocesseur, directement au niveau de la cible. Les appels aux fonctions `write` et `read` permettent d'écrire ou de lire directement les données. L'appel peut être bloquant si la synchronisation est longue, ou si l'accélérateur ne peut plus accepter ou fournir des données, ou encore si la configuration correspondant à l'opération est désactivée et qu'il n'y a pas de tampon avec la FIFO.

Au contraire, la FIFO générique s'implante dans la R-HAL. La cible fournit donc uniquement une fonction de synchronisation en lecture ou en écriture. Cette approche a deux avantages. Elle permet d'implanter une FIFO quand l'accélérateur n'en implante pas en matériel. Elle permet également de ne pas bloquer les opérations au bout des canaux de transferts, quand l'accélérateur est bloqué ou que la méthode d'interconnexion est lente. Cependant, elle génère des copies de données supplémentaires qui peuvent être longues.

5.5.3.5 Exemple : le codeur

Pour illustrer le fonctionnement du TaME, reprenons l'exemple précédent du codeur convolutif.

Imaginons deux applications basées chacune sur un code de rendement $\frac{1}{3}$, les applications A et B. Lors du chargement des applications A et B, les opérations de codage sont toutes les deux rangées dans le TaME. Comme l'application A se charge avant l'application B, c'est le code A qui est actif sur le coprocesseur. On considère pour l'exemple que les deux applications sont en mode démon (ce qui est peu probable pour une tâche d'émission). Chacune des applications à ses propres FIFO logiques, l'application active travaille sur la FIFO matérielle si elle est disponible, alors que l'application en attente a une FIFO logicielle.

Lorsque la FIFO d'entrée du code B dépasse le seuil fixé, l'ordonnanceur impose une commutation de configuration. Il sélectionne le code B comme nouvelle configuration, désactive les transferts de données vers le coprocesseur, et sauvegarde la configuration actuelle. Une fois ces

opérations effectuées, il charge la nouvelle configuration, modifie la source des données pour le transfert, puis relance les transferts de données.

5.6 FRK en pratique

Nous donnons dans cette section certains détails d'implantation de FRK, ainsi que des exemples d'utilisation. La jeunesse de FRK n'a pas permis de réaliser de comparaisons concluantes sur le coût de l'environnement. Cependant, les exemples présentés ici ont été implantés et sont fonctionnels. Nous avons utilisés GNURadio comme environnement de comparaison pour vérifier la correction du résultat.

5.6.1 État de FRK

FRK a été implanté en utilisant le langage C. Nous avons longuement hésité entre ce langage et un langage objet comme le C++ qui aurait permis de réaliser plus simplement certaines actions. Cependant, le C++ offre beaucoup plus de fonctionnalités que l'héritage utilisé dans FRK. Ces fonctionnalités, que nous n'utiliserons pas dans l'implantation, coûtent cher en termes de performance et d'espace mémoire requis, comme nous avons pu nous en rendre compte sur une plateforme embarquée lors de nos premiers essais. Il nous a donc semblé préférable d'utiliser le langage C, moins gourmand.

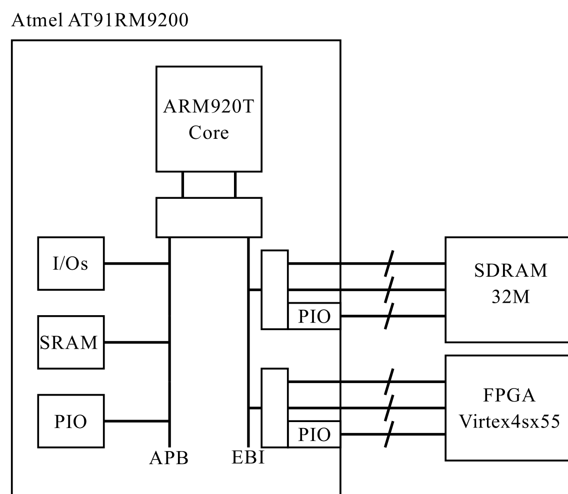


FIGURE 5.13 – Plateforme MagNET

FRK a été porté sur deux plateformes distinctes. Tout d'abord, une plateforme classique x86 a permis de valider la cible logicielle. La deuxième plateforme est basée sur un Atmel AT91RM9200 couplé à un FPGA. Elle a permis d'expérimenter sur la cible des accélérateurs matériels. On donne un schéma de cette plateforme, utilisée pour le projet européen MagNET, dans la figure 5.13. Un noyau Linux 2.6.23 avait déjà été porté sur cette plateforme. Nous avons réalisé le port du noyau *Real-Time Executive for Multiprocessor Systems* (RTEMS) 4.10.

Au moment de la rédaction de cette thèse, l'environnement est dans l'état suivant :

- la cible logicielle est implantée pour les deux noyaux ;
- la cible des coprocesseurs matériels est implantée pour certains types d'accélérateurs. Un port sur la plateforme Magali a été envisagée mais n'a pas pu être réalisé.
- l'interface R-HAL est complètement fonctionnelle du point de vue du contrôle des applications ;
- un mécanisme de *monitoring* a été défini et est partiellement implanté (*cf.* annexe C) ;

– l'OPM est également écrit et validé ;
– l'interface OpenCL est partiellement supportée. Cependant, certains éléments posent problème. La traduction de l'application en présence d'OpenCL en est un. La gestion de la surcharge est également difficile, à cause de l'unification requise des opérations. Ces problèmes sont en cours de résolution.

Les algorithmes utilisés actuellement pour tous les éléments sont encore basiques, l'accent ayant été mis sur le fonctionnement des éléments entre eux.

5.6.2 Implantation

L'implantation réelle du TaME des accélérateurs n'est pas aisée. On donne ici quelques informations sur les techniques employées pour les deux noyaux que nous avons expérimentés.

5.6.2.1 Linux

Le premier noyau expérimenté est un sytème Linux, en l'occurence un système Linux embarqué. Deux approches ont été suivies dans le cas du Linux, l'une s'exécutant dans le noyau, l'autre non.

La plus grosse difficulté vient en fait de la gestion des interruptions. Le nombre d'interruptions possibles est énorme, et elles peuvent virtuellement être déclenchées pour n'importe quelle raison, selon l'accélérateur étudié. Dans un premier temps, nous nous sommes donc intéressés à des accélérateurs sans interruption. L'utilisation de tels accélérateurs peut se faire dans l'espace utilisateur. La seule contrainte est de *mapper* les adresses sur lesquelles nous allons travailler pour la configuration dans le plan d'adressage virtuel imposé par Linux. Ceci se fait aisément, par l'intermédiaire de la fonction POSIX `mmap`, à condition de pouvoir travailler sur la mémoire comme sur un fichier. Une fois ceci fait, configurer le composant revient à écrire à une adresse connue, et il est possible d'exécuter le TaME en mode utilisateur.

Cependant, si les interruptions sont nécessaires, il faudra au minimum une partie en mode noyau. C'est l'implantation actuelle dans FRK. Les extensions HAL sont implantées comme un module du noyau, et ont donc accès à toutes les fonctionnalités de celui-ci. Au dessus de ce module, les fonctions du TaME proprement dites, comme l'ordonnanceur ou la commutation sont mises en place en mode utilisateur, par des appels système.

L'implantation de la cible logicielle utilise les *threads* POSIX. Les verrous sont réalisées en utilisant des *mutex* et des conditions.

5.6.2.2 RTEMS

L'implantation dans RTEMS suit la même logique. Chacun des éléments est ajouté normalement dans le *Board Support Package* (BSP), ainsi que les traitants d'interruption. Le TaME est ensuite implanté normalement comme pour l'espace utilisateur de Linux, comme une librairie qui implante les différentes fonctions requises.

La cible logicielle peut être implantée soit en utilisant l'interface POSIX du noyau, soit en utilisant les `rtems_task` spécifiques. Les verrous sont implantés en utilisant des *spinlock*, solution qui doit pouvoir être améliorée.

5.6.3 Exemple du codeur convolutif

5.6.3.1 Présentation du programme

Le codeur convolutif a déjà été présenté dans ce chapitre. On se base sur une application similaire à l'application de test pour les expérimentations du GPU au chapitre 4. Les sources et puits de l'application sont des fichiers. Il n'y a pas de notions de débit à tenir dans ce cas. Le

programme est simple à écrire. On crée les trois éléments constituant l'opération, à savoir la source et le puits de l'application qui sont tous les deux des fichiers, et l'opération de codage.

Deux applications avec deux codeurs sont créées, on donne ici le code pour l'application du codeur A.

```
int coderA_waveform(frk_wf_t *waveform)
{
    frk_filesource_t source;
    frk_filesink_t sink;
    frk_convcoder_t coder;

    init_filesource(&source);
    init_filesink(&sink);
    init_convcoder(&coder);

    // Ouverture des fichiers concernes
    FILE *source_file = fopen(sourcefilename, ...);
    FILE *sink_file = fopen(sinkfilename, ...);

    // Mise a jour source
    source.file = source_file;

    // Mise a jour puits
    sink.file = sink_file;

    // Reglage codeur
    coder.num_adder = 3;
    coder.num_mem = 3;
    // Definition des entrees actives des additionneurs
    coder.adder_mask[0] = 0x8; // 1000
    coder.adder_mask[1] = 0xb; // 1011
    coder.adder_mask[2] = 0xe; // 1110
    // Definition du schema de poinconnage
    coder.puncturing = 0xFFFFFFFF; // Pas de poinconnage

    connect(source,0,coder,0); // source ---> codeur
    connect(coder,0,sink,0);   // codeur ---> puits

    init_waveform(waveform,3); // waveform avec 3 elements
    add_op(source,waveform);
    add_op(coder,waveform);
    add_op(sink,waveform);
}
```

On va ensuite appeler la fonction `translate`, qui permet la traduction d'une *waveform* en une CI. On génère également une autre *waveform* avec des polynômes différents pour le codeur B. Cette CI dépendra de la plateforme en cours.

5.6.3.2 Exécution logicielle

Sur la plateforme x86, la traduction de cette *waveform* génère une CI avec trois opérations et 2 canaux de communications FIFO. Ces opérations, pour l'exécution logicielle, sont représentées sous la forme de trois fonctions, une pour chaque opération. On représente sur la figure 5.14 la CI correspondant au codeur A. Seul le TaME Linux est disponible sur la plateforme x86. La structure servant à représenter la CI, ainsi que les types annexes sont donnés dans l'annexe C. Il y a une liste

d'opérations par cible, ainsi qu'une liste des FIFO utilisées. Dans FRK, les listes sont en réalité implantées comme des tableaux de pointeur sur les éléments concernés. Pour ces FIFO, on donne la représentation générique au niveau de la R-HAL. Cette représentation utilise l'implantation au niveau du TaME.

Les flèches sur la figure servent à représenter les relations entre les différents éléments utilisés pour définir une CI. Ainsi, le champ `outputs` de l'opération `source_config` est en fait un pointeur sur la FIFO `channel_0`. Les opérations sont représentées au niveau du TaME par une tâche associée dans le noyau, et une fonction `process` (non représentée ici) qui est fournie lors de la traduction par le TaME. Pour simplifier la figure déjà très chargée, les entrées/sorties des configurations pointent sur les FIFO du TaME. Dans l'implantation réelle, ces entrées/sorties pointent sur les FIFO du R-HAL. Une simplification de cette interface est en cours, qui supprimera un niveau en fusionnant les deux. De même, pour alléger la figure, les pointeurs sur les producteurs `prod` et les consommateurs `cons` des FIFO ne sont pas représentés, ainsi que les pointeurs sur la CI.

La concurrence pour s'exécuter sur la GPP est gérée au niveau du noyau, les `pthread_t` étant intégrés dans la file des éléments exécutables de celui-ci.

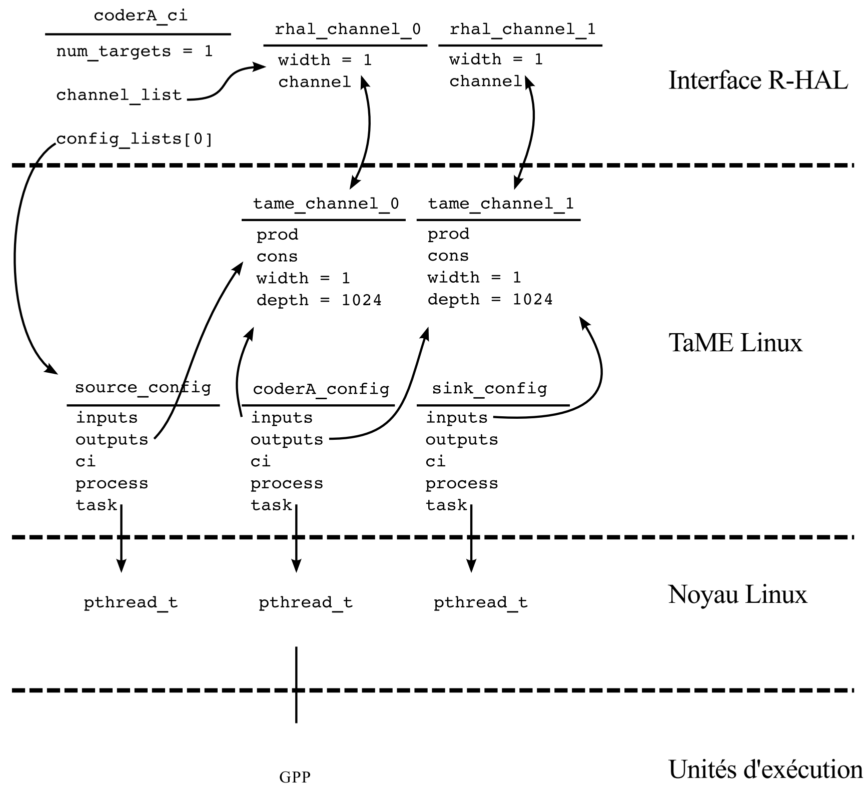


FIGURE 5.14 – Application traduite pour une cible logicielle

5.6.3.3 Exécution matérielle : une seule application

On commence par synthétiser sur le FPGA un codeur convolutif figé ne permettant que l'exécution du codeur A. Lors de la traduction, l'appel à la fonction de sélection pour cette cible détecte que le codeur A peut être implanté en utilisant le codeur matériel. La traduction de l'application pour le codeur B détecte qu'il n'y a pas d'implantation matérielle pour ce codeur. Une implantation logicielle est donc mise en place.

On se retrouve donc avec une CI purement logicielle pour le codeur B, et une CI mixte pour le codeur A, les sources et puits des applications étant nécessairement logiciels. Le codeur convolutif a été intégré dans le TaME des accélérateurs. Il n'est pas configurable. On utilise dans cet exemple une FIFO dédiée dans laquelle le processeur est chargé de réaliser la copie à une adresse précise. Cette adresse est le point d'entrée d'une FIFO matérielle de taille 1024 octets. La sortie fonctionne de la même manière. Une implantation possible des FIFO dans ce cas est présentée dans l'annexe D.

On montre dans la figure 5.15 la CI correspondant au codeur A, s'exécutant de manière mixte. Une fois de plus, seule la cible logicielle doit gérer la concurrence.

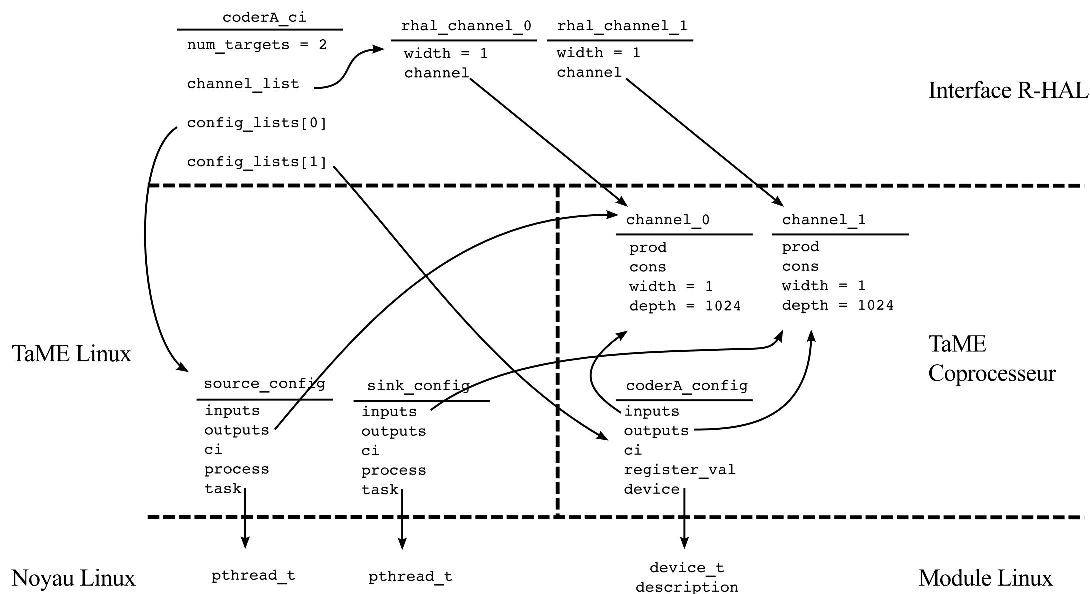


FIGURE 5.15 – FRK avec deux codeurs, cibles mixtes

Le processeur doit donc gérer cinq tâches, la dernière étant réalisée par l'accélérateur. Notons que ce type d'accélérateur non configurable n'est pas encore géré correctement dans FRK. En effet, si une deuxième application se base sur le même codeur A, elle ne pourra pas utiliser le codeur matériel. Il faudrait pour cela rendre accessible la mémoire du codeur, pour pouvoir remettre le codeur dans un état puis dans l'autre. Afin de gérer correctement ce cas, une dynamisation de l'OPM est en cours, qui permettra à terme de rendre l'OPM conscient des disponibilités de la plateforme au moment de la traduction. Ceci sera couplé à une requête basée sur le débit requis, afin de sélectionner l'implantation la mieux adaptée entre logiciel et matériel selon les besoins de chaque application. Un retour devra également être possible, si l'application utilisant le codeur est déchargée, afin de permettre une modification dynamique de la cible dans ce cas.

5.6.3.4 Exécution matérielle : deux applications

Finalement, on synthétise un codeur configurable permettant d'exécuter les deux codes A et B. Ce codeur est désactivable, c'est à dire qu'un registre de configuration [3] est disponible avec un *flag* pour activer ou désactiver l'encodage. La CI correspondante est présentée sur la figure 5.16. La représentation des deux applications complexifient encore la figure. Pour l'alléger, les flèches ont été quasiment toutes retirées. Elles restent similaires aux figures 5.14 et 5.15. Les FIFO d'indices 2 et 3 sont rattachées au codeur B.

3. utilisé par le TaME et non par les applications, celles-ci utilisant les registres de paramétrisation

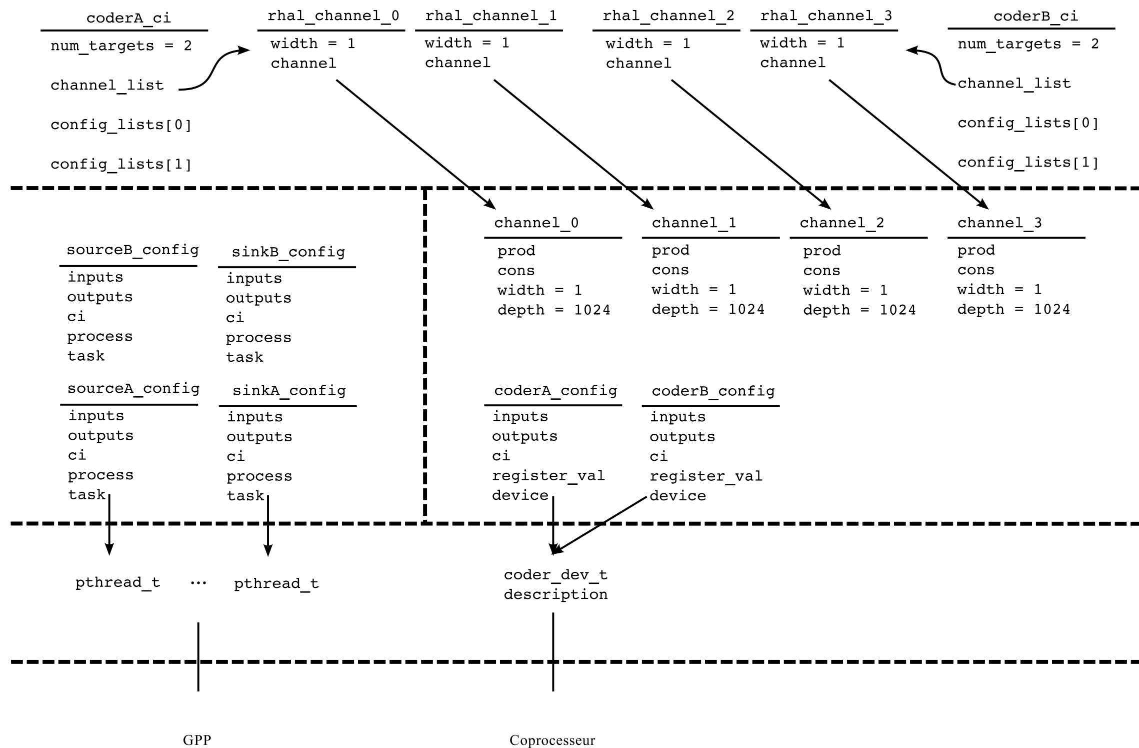


FIGURE 5.16 – FRK avec deux codeurs en matériel

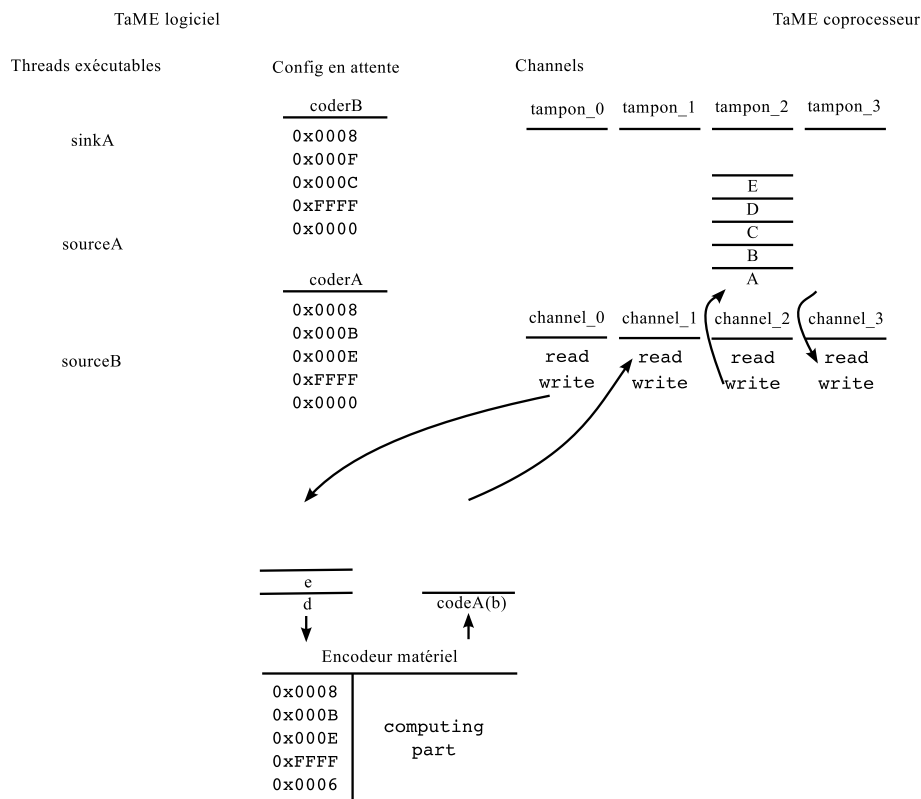
On remarque que contrairement aux deux cas précédents, il y a également de la concurrence pour un accès au coprocesseur matériel. Pour gérer cette concurrence, on utilise donc la commutation de configuration présenté dans la section 5.5. On présente dans la figure 5.17(a) l'état du système durant l'exécution juste avant l'appel à l'ordonnanceur, avec le codeur A actif et chargé sur l'encodeur matériel. Les flèches représentent le *buffer* utilisé par les FIFO. Les canaux rattachés à la configuration active utilisent directement les FIFO matérielles en entrée du codeur. Les canaux rattachés à une configuration en attente utilisent les tampons logiciels.

Le fichier contient l'alphabet, et est lu en boucle. Le codeur A encode l'alphabet en minuscule, alors que le codeur B encode l'alphabet en majuscule. Dans cet état, la source du codeur B est active, le tampon de la FIFO2 se remplit donc. Le codeur A est actif, le tampon des FIFO 0 et 1 n'est donc pas utilisé. Par contre, les FIFO matérielles en entrée du codeur sont utilisées, et l'opération `sinkA` a déjà lu un élément pour le copier dans son fichier de sortie. L'opération `sinkB` est en attente de données et n'est donc pas dans la liste des *threads* exécutables du noyau. L'encodeur ayant fonctionné, l'état du registre volatile (le dernier de la liste) n'est plus l'état initial, d'où le décalage entre la configuration dans le TaME, et la valeur dans l'encodeur matériel.

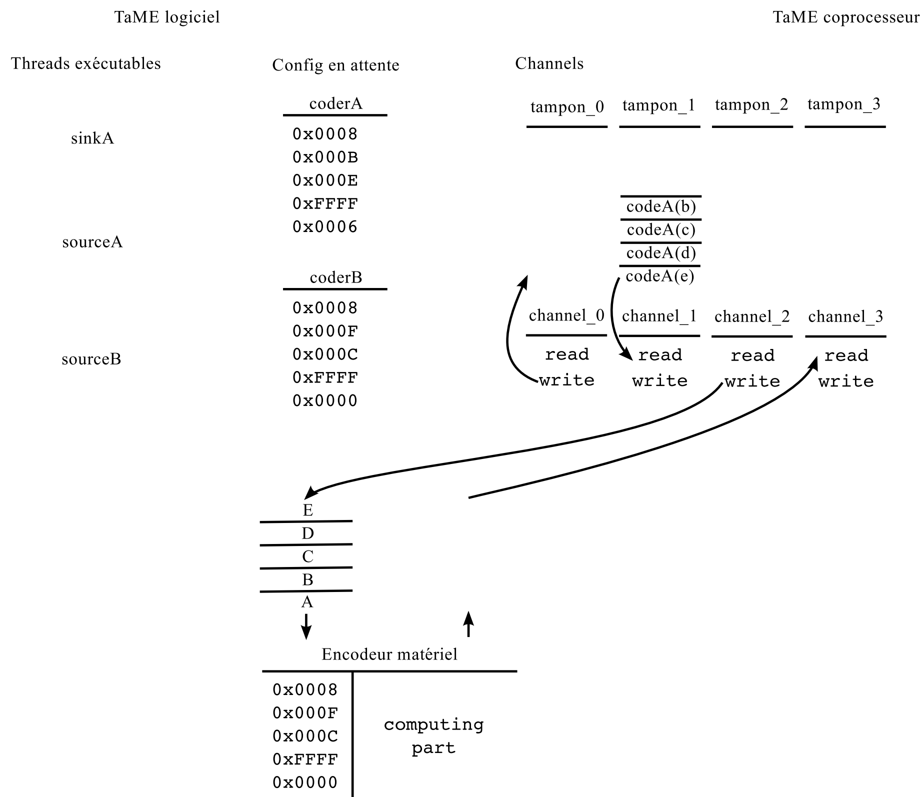
Lorsque la FIFO d'entrée du codeur B, alimentée par la tâche source `sourceB`, atteint son seuil de déclenchement, l'ordonnanceur est appelé. Comme l'encodeur est plus rapide que le taux de remplissage de la FIFO par la tâche `sourceA`, la FIFO d'entrée du codeur A n'est pas remplie jusqu'au seuil [4]. Lors de l'appel à l'ordonnanceur, la configuration `coderB` est sélectionnée, et une commutation de configuration a lieu. Le système se retrouve alors dans l'état représenté sur la figure 5.17(b), juste avant la réactivation de l'encodeur.

Ce cas est un exemple. Dans la réalité, l'ordonnanceur est plus souvent appelé parce que la FIFO atteint son seuil minimal (vide). Les FIFO sont ici un mélange de FIFO matérielle et logi-

4. dans le système réel, la FIFO est en fait quasiment toujours vide, le codeur traitant ses données très rapidement



(a) Avant l'ordonnancement



(b) Après l'ordonnancement

FIGURE 5.17 – Représentation de l'état du système

cielle comme dans l'annexe D. Elles utilisent un *buffer* pour écrire les données quand la configuration n'est pas chargée, et transfèrent les données dès que la FIFO se charge pour réaliser la synchronisation. Les données sont ensuite écrites directement dans le matériel.

5.6.4 Exemple complet : IEEE 802.11

5.6.4.1 Présentation du programme et de la plateforme

Afin d'essayer FRK avec une application plus complète, nous avons implanté la norme IEEE 802.11 (la première version, sortie en 1997). On donne dans la figure 5.18 une représentation de la *waveform* de l'émission et de la réception de la norme. On implante la transmission et la réception sur la même plateforme. La *waveform* de transmission envoie des données à la *waveform* de réception. La chaîne de transmission est fonctionnelle, les données décodées à la réception sont celles avant la transmission. Cependant, il n'y a pas de notions de temps réel, puisqu'il n'est pas pris en compte dans FRK dans l'état actuel. Il n'y a pas non plus eu de validation avec un périphérique 802.11 réel.

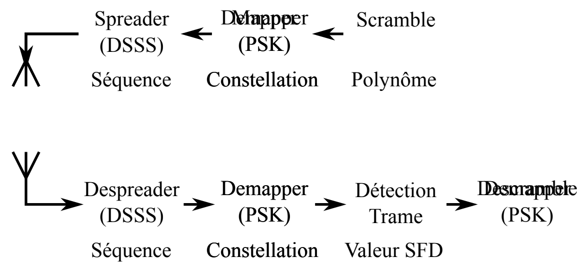


FIGURE 5.18 – Application IEEE 802.11 simplifiée utilisée

On utilise pour cette exécution la plateforme représentée en figure 5.19. Cette plateforme fournit un corrélateur utilisable pour réaliser la corrélation avec le code de Barker utilisé dans la norme. Ce corrélateur travaille sur des phases. Elle fournit également un modulateur DPSK utilisable pour le BPSK et le QPSK, ainsi qu'un démodulateur matériel pour le QPSK mais pas pour le BPSK. Un mélangeur et un démélangeur ont également été implantés en matériel. On cherche donc à utiliser les *waveforms* pour envoyer de l'information, puis pour recevoir cette même information. Ces accélérateurs sont accessibles sur le bus dédié au FPGA dans l'architecture.

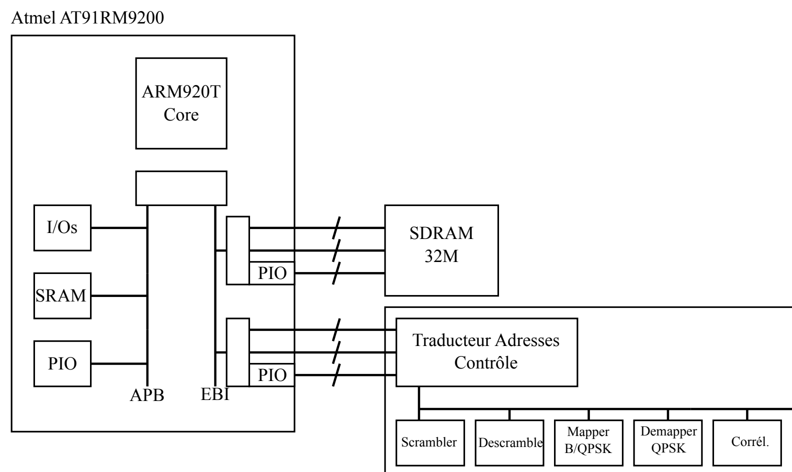


FIGURE 5.19 – Plateforme matérielle pour l'application IEEE 802.11

5.6.4.2 Fonctionnement BPSK

On commence par exécuter la norme avec la modulation BPSK. La CI utilise alors le processeur pour la démodulation. L'interface du démodulateur PSK fourni dans FRK est la suivante :

```
struct frk_demap_psk_s {
    frk_op_t param;
    unsigned int symbol_size;
    unsigned char gray_coded; // 1 : TRUE, 0 : FALSE
    unsigned char differential;//1 : TRUE, 0 : FALSE
    float first_symb;
}
```

L'attribut `symbol_size` permet de donner le nombre de bits contenus dans un symbole, et par conséquent le nombre de symboles possibles. Les symboles en sortie du démodulateur sont des phases représentées en flottant. L'attribut `gray_coded` est un booléen, qui permet de définir si la modulation utilise un code Gray pour donner l'ordre des symboles. De même, l'attribut `differential` permet de définir si la modulation est différentielle ou non. Finalement, `first_symb` permet de donner la phase pour le premier symbole de la liste, afin de reconstruire la constellation.

La configuration de l'opération pour le cas BPSK avec est donc :

```
    demap.symbol_size = 1;
    demap.gray_coded = 1; // inutile
    demap.differential = 1;
    dempa.first_symb = 0.0f;
```

Cette configuration, étudiée par la fonction `select_config` de l'opération pour la cible matérielle, renvoie une configuration nulle, aucun démodulateur matériel ne permettant une démodulation avec une taille de symbole de 1. L'implantation du démodulateur est donc logicielle.

5.6.4.3 Demande d'adaptation

La fonction suivante est implantée pour contrôler l'application.

```
// waveform est l'application de reception IEEE 802.11
// i est l'indice de l'operation de demodulation dans la waveform
int set_qpsk_modulation(frk_wf_t *waveform, int i)
{
    // op_list est la liste des operations de la waveform
    waveform->op_list[i].symbol_size = 2;
    waveform->op_list[i].gray_coded = 1;
    waveform->op_list[i].first_symb = 0.0f;
    // repercussion de la modification dans la ci
    update(waveform, (frk_op_t *)&(waveform->op_list[i]));
}
```

Cette fonction donne les paramètres de la démodulation pour qu'ils correspondent à une modulation QPSK, utilisant un code Gray, et dont le premier symbole (00) se code avec une phase nulle. L'appel à `update` permet de mettre à jour l'opération de modulation. La R-HAL appelle la DRL qui utilise l'OPM, et vérifie l'implantation sur les différentes cibles. Au contraire du BPSK, la cible des accélérateurs matériels permet d'exécuter cette démodulation. Comme on utilise des FIFO dédiées, la FIFO d'entrée est d'abord vidée, puis la tâche est supprimée de la liste des tâches

dans la cible logicielle. La nouvelle opération est ajoutée dans la cible matérielle, et les informations pour l'écriture et la lecture dans les opérations précédentes et suivantes sont mises à jour. L'adaptation est ainsi réalisée pour la chaîne de réception.

Dans le cas de la transmission, la cible est la même avant et après la modification. Rien n'est supprimé, on désactive l'opération, on attend que la FIFO soit vidée, puis on met à jour la configuration du coprocesseur pour cette opération. On réactive ensuite l'opération.

On remarquera que, mis à part le temps d'attente lié au besoin de vider la FIFO, le temps d'adaptation est court. La vidange des FIFO n'empêche d'ailleurs pas les autres opérations de se faire, sauf pour les opérations qui précédent directement l'opération en cours de modification.

5.6.5 Quelques chiffres

Il est actuellement difficile de donner des résultats chiffrés sur l'environnement FRK. En effet, celui-ci est très récent, et n'existe actuellement qu'à l'état de preuve de concept. On peut tout de même donner les quelques résultats disponibles à titre d'indications. Ces résultats sont susceptibles de varier grandement au fur et à mesure de l'évolution de l'environnement.

Les premiers chiffres concernant l'environnement sont la taille du projet. L'environnement est actuellement géré en utilisant le gestionnaire de version Git. Le projet fait une taille d'environ 4500 lignes de codes, dont 4000 sont dédiées à l'environnement lui-même, et 500 sont présentes pour les configurations (les fonctions de la cible logicielle et de la cible OpenCL partielle, et les configurations de la cible matérielle).

Une fois l'application compilée, on peut estimer l'empreinte mémoire de cette application. Il est par contre difficile de dissocier l'espace occupé par l'environnement de l'espace occupé par les configurations elles-mêmes. Pour l'application 802.11, présentée dans la suite, le binaire pour la plateforme x86 a une empreinte statique d'environ 156 kio, avec autour de 5 Mo d'allocations dynamiques (les FIFO logicielles sont configurées pour être grandes). Pour la plateforme ARM, le binaire fait environ 172 kio pour la cible Linux (l'augmentation est liée à la présence d'une cible matérielle), et 320 kio pour la cible RTEMS (le noyau se lie en statique à l'application, d'où le binaire plus grand).

Les chiffres concernant le surcoût en temps d'exécution de l'environnement ne sont pas réellement représentatifs des possibilités de l'environnement. Pour la cible Linux, la proportion du temps d'exécution consacrée à l'environnement est d'environ 10% pour la cible logicielle uniquement, 12% quand on mélange la cible logicielle et la cible matérielle. Pour la cible RTEMS, le manque d'outils de profilage n'a pas permis d'extraire ces valeurs.

5.7 Conclusion

Dans ce chapitre, nous avons abordé l'environnement de radio flexible FRK. Cet environnement a été conçu pour répondre à deux objectifs majeurs :

- permettre une intégration de n'importe quelle unité d'exécution ;
- unifier l'écriture d'une application quelque soit les unités d'exécution utilisées.

FRK est conçu comme une librairie à utiliser pour constuire et gérer des applications radio. Il se base sur des cibles, qui représentent les différentes unités. Chacune des cibles doit implanter une API commune, mais reste libre de gérer ses opérations. Ces différentes cibles sont reliées à une librairie de traduction d'un jeu d'opérations basiques. Cette librairie, l'OPM, permet de définir quelle sera la cible de prédilection pour une opération traduite donnée. Elle est utilisée pour permettre l'unification de la définiton des applications.

Cet environnement a été implanté et porté pour deux architectures différentes, et pour deux noyaux différents. Différents cas pratiques ont été présentés, afin de montrer les mécanismes régis-

sant l'environnement. Même s'il reste expérimental, et largement améliorable, FRK remplit les objectifs fixés.

Chapitre 6
Conclusion

CINQ questions ont guidé notre réflexion dans le but de permettre l'intégration d'unités d'exécution hétérogènes pour des applications de radio flexible. Elles ont donné lieu à des travaux innovants, dont nous dressons le bilan dans cette conclusion. Ces travaux poussés à l'état de prototypes valident la pertinence de la radio flexible et laissent entrevoir d'élégantes solutions pour la flexibilité des terminaux mobiles de demain.

Réponse à la problématique

L'utilisation du GPGPU pour la radio logicielle a été étudiée dans le chapitre 4. Les travaux réalisés au cours de cette thèse montrent que le GPGPU peut présenter un intérêt pour des applications radio à fort débit, et que les futurs processeurs graphiques dédiés au domaine de l'embarqué devront faire l'objet d'une attention toute particulière.

Comment utiliser de manière efficace le GPGPU pour la radio logicielle ?

Pour rendre attrayante l'utilisation du GPGPU, le défi à relever consiste à compenser les pertes de temps occasionnées par le transfert des données et le contrôle de l'architecture SIMD par l'emploi de toute la puissance offerte par les unités de calcul. Nous avons donc concentré notre étude sur les techniques de parallélisation des traitements de données, à la fois pour des opérations s'appliquant à des éléments unitaires et pour des opérations s'appliquant à des vecteurs ou à des données interdépendantes. La conception d'une opération a été envisagée selon deux méthodes. Une première méthode, dite à grain fin et résultant d'une adaptation de travaux précédents, consiste à optimiser les implantations pour le GPU avec un découpage en petits *kernels*. Nous l'avons mise en œuvre et les résultats obtenus sont décevants et en deçà des capacités d'un GPP. Une seconde méthode, dite à gros grain, a été introduite dans cette thèse. Elle consiste à utiliser comme *kernels* l'opération optimisée complète. Bien qu'induisant une plus grande latence, les résultats obtenus avec cette approche sont encourageants, avec un gain observable pour des opérations unitaires. Afin de permettre le contrôle des opérations, nous avons défini un paramètre, nommé seuil d'exécution, qui définit le nombre d'opérations devant être exécutées en parallèle. Nous avons pressenti l'existence d'un seuil optimal et nous avons montré sa valeur expérimentalement.

Comment intégrer les opérations sur GPU dans une application radio ?

Une application radio consiste en une suite d'opérations sur les données. Plusieurs techniques de contrôle ont donc été testées pour enchaîner les opérations exécutées par le GPU. Nous nous sommes attardés sur deux d'entre elles que nous appelons respectivement intégration distribuée et intégration centralisée. Dans l'intégration distribuée, des FIFO assurent le transfert des données d'une opération à une autre. Afin de rendre cette approche plus performante, une FIFO spécifique à l'environnement à été développée. Bien que le débit de calcul présente un gain par rapport à une implantation CPU, cette intégration reste peu efficace car inadaptée à l'architecture de contrôle centralisée du GPU. La mise en œeuvre de l'intégration centralisée s'est avérée plus prometteuse.

Elle utilise des tampons de taille fixe entre les opérations. Lors de l'initialisation, le seuil optimal est utilisé pour définir la séquence d'appel des opérations permettant d'exécuter l'application voulue. Les débits atteints avec cette technique d'intégration se sont révélés 4 fois supérieurs à ceux obtenus avec un CPU, prouvant ainsi l'intérêt du GPU pour exécuter une application radio.

Quelle solution adopter pour utiliser les accélérateurs matériels dans une application de radio flexible ?

Le GPU n'est pas la seule unité d'exécution qui peut être utiliser pour la radio flexible. L'utilisation de coprocesseurs matériels peut également conduire à diminuer les temps d'exécution des applications radio. Le chapitre 5 propose une solution pour utiliser ces coprocesseurs. Plutôt que de les encapsuler dans des opérations logicielles, nous avons préféré les considérer comme des processeurs extrêmement spécialisés. La gestion de ces coprocesseurs passe donc par la gestion d'une liste de tâches dédiées à ces accélérateurs. La liste contient l'ensemble des opérations devant être exécutées, et les tâches représentent la configuration d'un accélérateur pour une opération donnée. Afin de permettre la concurrence des tâches sur ces coprocesseurs, nous avons introduit la notion de commutation de configuration, dont le but est d'enregistrer l'état d'un coprocesseur et de charger une nouvelle configuration.

Quelle architecture définir pour permettre l'exécution d'une application sur des architectures hétérogènes ?

Ces différentes unités d'exécution possibles peuvent coexister sur une platforme. Nous avons donc proposé dans le chapitre 5 une approche pour permettre la coopération de ces unités. FRK est un environnement formé d'un ensemble de librairies qui permettent la gestion d'unités d'exécution hétérogènes dans le cadre de la radio flexible. L'environnement repose sur la définition de cibles d'exécution. Chaque cible est définie pour chaque type d'unité d'exécution utilisée sur la plateforme et permet d'implanter les différentes techniques de contrôle et de transfert de données. Trois cibles ont été considérées au cours de la thèse :

- la cible logicielle générale (cible des GPP) ;
- la cible OpenCL qui a été rapidement présentée, mais n'est pas encore fonctionnelle, même si l'étude du chapitre 4 permet d'envisager une intégration rapide ;
- la cible des accélérateurs matériels.

Comment décrire et traduire une application radio ?

L'utilisation de FRK permet d'apporter une solution élégante au problème de généricité de l'application soulevé par l'utilisation de plateformes hétérogènes. Pour cela, FRK s'appuie sur deux caractéristiques. Une librairie d'opérations génériques communément employées par les applications radio est proposée au développeur. Associée avec la séparation en cibles d'exécution, FRK permet d'utiliser la plateforme à haut niveau, sans connaître le détail de chacune des architectures supportées. Un développeur voulant décrire une application radio utilise les opérations génériques offertes par la librairie et crée une application générique appelée *waveform*. Cette waveform est automatiquement traduite en une application exécutable sur la plateforme à partir des implantations disponibles sur les cibles présentes. L'application générique et l'application exécutable sont intimement liées, et toute modification sur l'une peut être répercutée sur l'autre pour offrir une adaptabilité sans reconfiguration.

Perspectives

A la lecture de ces travaux, plusieurs chemins se dessinent, tant du point de vue des développements futurs que des axes de recherche à venir. Tout d'abord, les travaux présentés sur le GPU seront poursuivis. Nous souhaitons mettre en œuvre l'approche multitâche pour diminuer le nombre de blocs requis pour atteindre les gains optimaux, ceci afin de réduire l'occupation mémoire. Ensuite, il nous parait essentiel de terminer l'intégration de la cible OpenCL dans l'environnement FRK, puis de tester son efficacité. Notre premier objectif était de prouver la fonctionnalité de l'environnement. Dans un deuxième temps, nous pourrons travailler sur le surcoût lié à la présence de cet environnement sur la plateforme. On définit dans notre cas le surcoût comme la part du temps d'exécution due à l'environnement. Actuellement, FRK présente un surcoût inférieur à celui de GNURadio ou SCA, mais plus important qu'Aloe, autour de 10%. Ce surcoût est calculé sur les différentes applications implantées en utilisant un outil de profiling, pour la cible logicielle. Outre, l'optimisation de FRK en termes de coût, la prise en compte du temps réel est un objectif à court terme. Une définition du temps est déjà en cours dans l'environnement, qui sera le point de départ de cette prise en compte. En effet, la notion de moment d'une reconfiguration n'est pas triviale, mais nécessaire pour permettre l'utilisation de l'environnement dans une application réelle. L'intégration de nouvelles cibles, comme le FPGA (et plus particulièrement, la reconfiguration dynamique partielle de celui-ci) est également envisagée. ou l'intégration de nouvelles cibles comme les FPGA pourront être envisagées.

L'évolution de l'environnement FRK et les choix qui seront à poser dans le futur ouvrent sur de nouvelles questions. Notamment, la correspondance opération/implantation s'avère être une tâche complexe qu'il peut être intéressant d'analyser en détail. Prenons l'exemple du codeur convolutif. D'un point de vue applicatif, il nous a paru naturel d'associer au codeur convolutif un poinçonneur. Le poinçonnage est une opération « simple » et définir un accélérateur dédié à cette opération, qu'il faudra par la suite intégrer dans la chaîne de communication, semble sous-optimal à cause du surcoût important. Cependant, l'opération « poinçonnage » doit malgré tout figurer dans la librairie et être proposée aux développeurs. Il apparait alors comme impératif de mettre en place le support des accélérateurs matériels implantant une séquence d'opérations. Des premiers pas ont été faits dans cette direction au niveau des TaME. En particulier, la possibilité d'employer des FIFO dédiées à une cible et inaccessibles des autres cibles découle de cette problématique. Par contre, le support au niveau du traducteur et pour l'adaptabilité est plus complexe à mettre en œœuvre et est encore en projet. La question va plus loin que le simple support des accélérateurs. Plus les opérations sont petites, plus leur définition est flexible, mais plus le surcoût devient important. Quelle que soit la cible finale, il est peut-être plus intéressant de définir de petites opérations génériques qui se traduisent en grosses opérations à exécuter.

La radio flexible souffre aujourd'hui de son manque d'unification et de clarté. Il est difficile de discerner, dans le foisonnement de possibilités et d'objectifs une tendance bien définie. Malgré tout, certains points paraissent acquis. Tout d'abord, la radio logicielle sur GPP ne sera pas utilisée pour l'exploitation réelle de la radio avant très longtemps, et ce n'est pas nécessairement un but à poursuivre. Ceci ne veut pas dire qu'elle est impossible : en multipliant les processeurs, en complexifiant un peu l'architecture, il sera toujours possible d'obtenir suffisament de puissance de calcul pour exécuter n'importe quelle norme, même les plus difficiles. Cependant, sans même considérer l'encombrement et la consommation électrique qui ne permettront ce genre de plateforme que pour certains cas bien définis, le coût logiciel deviendra tellement important, qu'il serait plus simple et moins cher d'utiliser un ASIC, et de le changer pour le faire évoluer. Qui plus est, il faudra toujours une partie matérielle pour la radio logicielle, et cette partie matérielle sera toujours plus compliquée qu'une simple antenne. La difficulté liée à la définition d'une antenne ayant un gain satisfaisant sur une gamme de fréquence de plusieurs gigahertz fait qu'une solution utilisant plusieurs antennes est préférable. L'échantillonnage est également limitant, comme nous l'avons

dit dans le chapitre 2. Même une plateforme de radio logicielle est donc hétérogène. Une plateforme comme la plateforme Magali (ou le processeur Leocore) est donc un excellent compromis, permettant une exécution logicielle pour certaines opérations, mais offrant un support matériel pour les opérations communes. La définition d'éléments matériels permettant une reconfiguration va également dans le bon sens.

Avec FRK, nous disposons maintenant d'une interface unifiée qui peut s'utiliser quelle que soit la plateforme. Nous espérons évidemment que cet environnement sera utilisé à l'avenir, sa mise à disposition étant prévue dans un avenir proche. L'unification nous semble une notion incontournable dans les travaux futurs et ne concerne pas seulement la couche PHY de la radio. La PL de FRK est un premier pas vers l'unification de la couche MAC, qui ouvre sur une plus grande coopération entre les différentes normes existantes et qui ne pourra qu'améliorer leur efficacité.

Annexe A

Résultats complets pour le GPU

CETTE partie est dédiée à la présentation des résultats pour l'intégration du GPU dans l'environnement GNURadio. Ces résultats sont données de manière brute, sans explication étendue, pour le lecteur intéressé. L'étude approfondie d'une partie de ces résultats est disponible dans le chapitre 4.

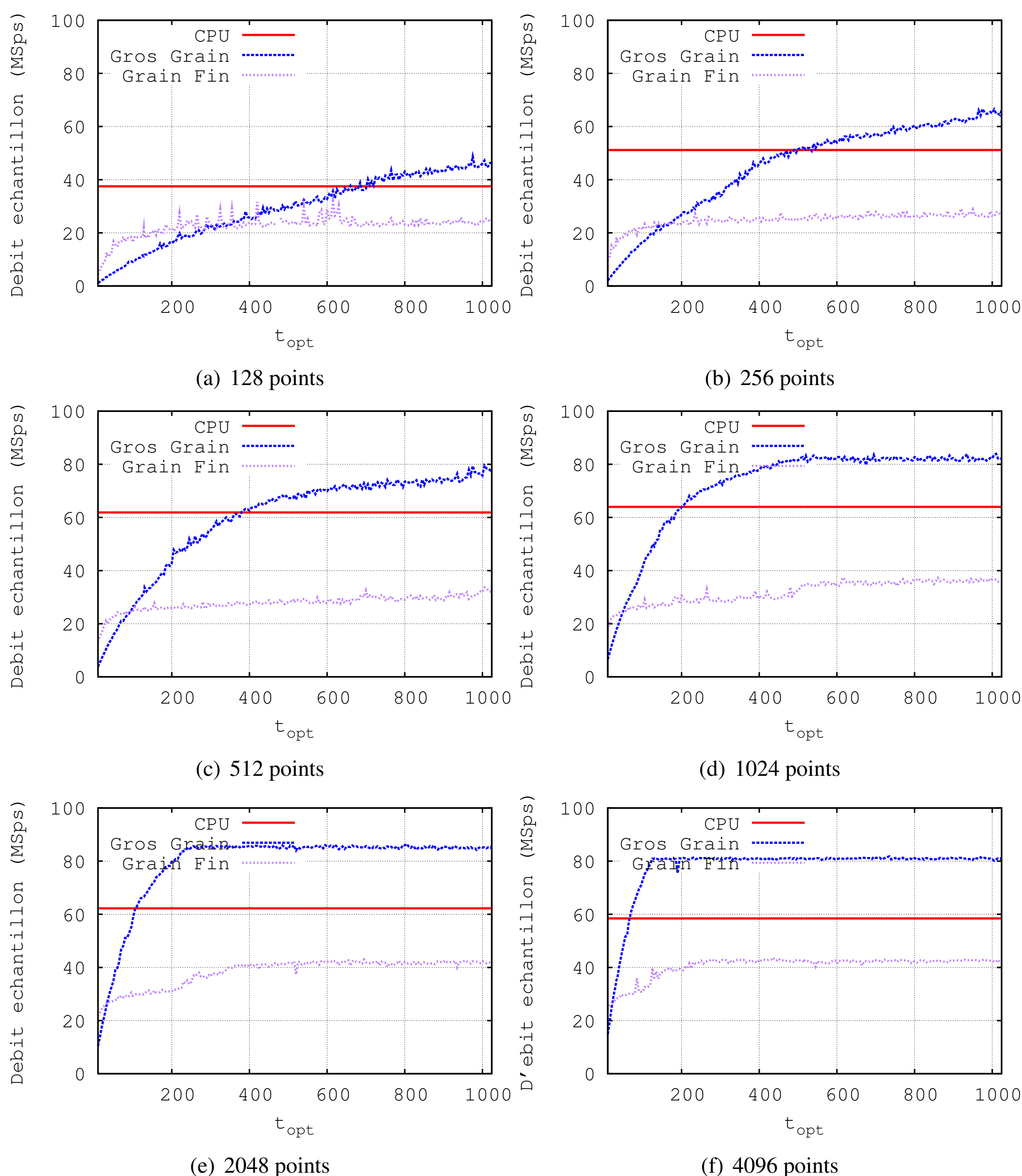
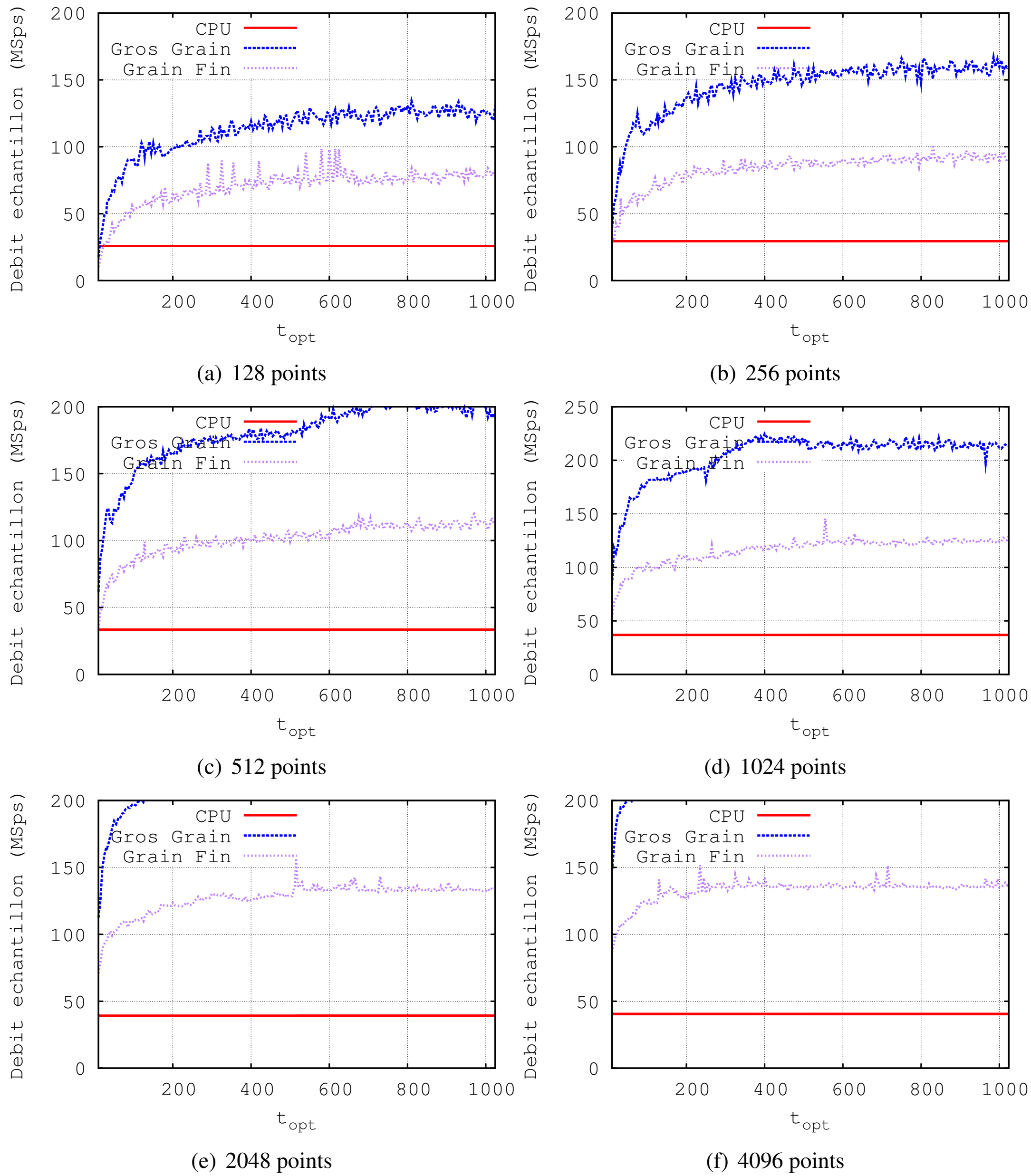


FIGURE A.1 – Résultats complets pour la FFT



(a) 128 points

(b) 256 points

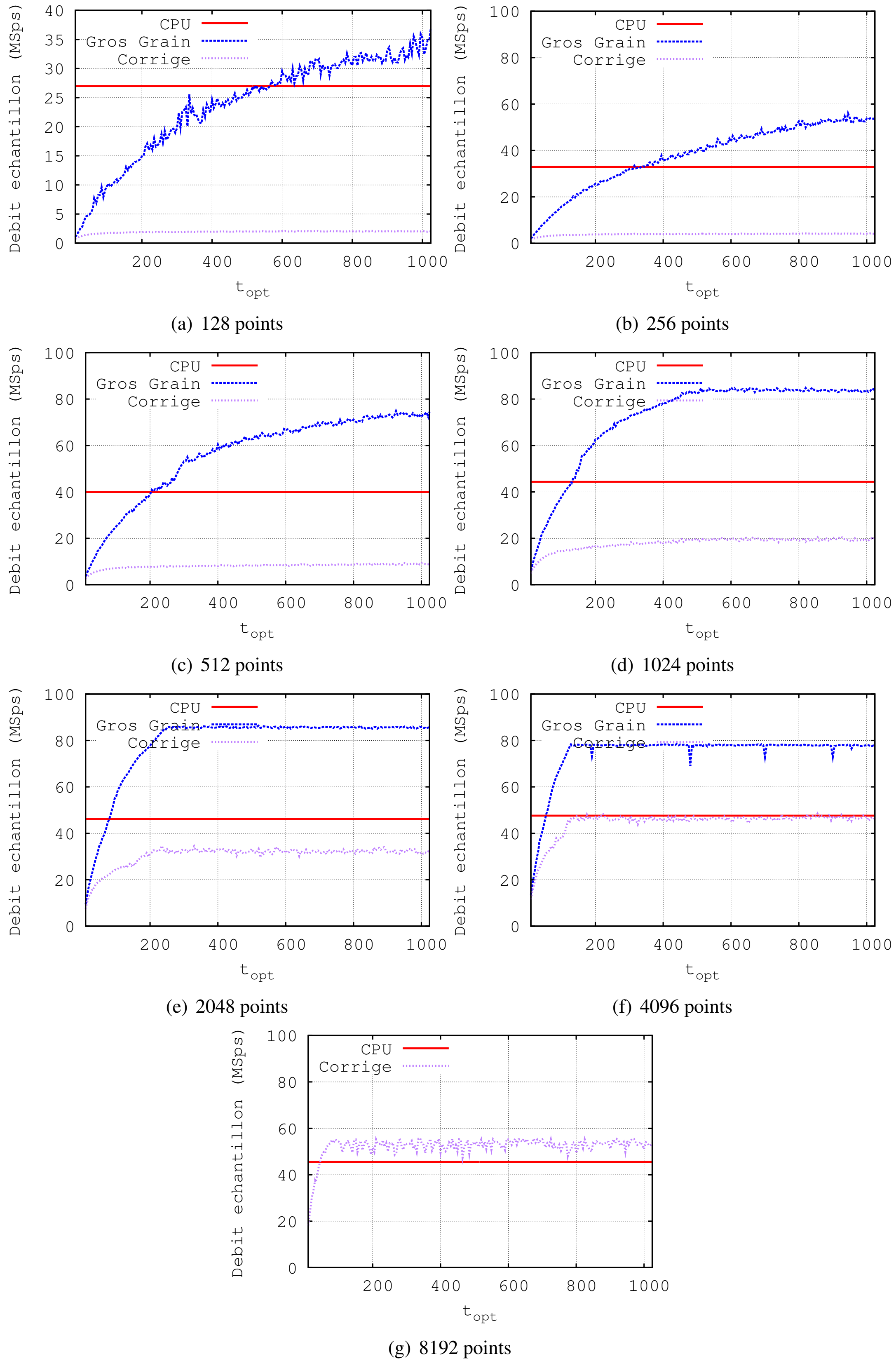
(c) 512 points

(d) 1024 points

(e) 2048 points

(f) 4096 points

FIGURE A.2 – Résultats complets pour le *demapping*



(a) 128 points

(b) 256 points

(c) 512 points

(d) 1024 points

(e) 2048 points

(f) 4096 points

(g) 8192 points

FIGURE A.3 – Résultats complets pour la IIR

Annexe B

Parallélisation d'un filtre IIR

L'objectif de ce chapitre est de démontrer la correction de l'algorithme IIR compatible avec l'architecture SIMD du GPU. Cet algorithme est utilisé pour l'implémentation de la radio logicielle sur le GPU en 4. On présente d'abord dans ce chapitre le principe du calcul et de la démonstration, puis on effectuera la démonstration proprement dite.

B.1 Principe

On cherche à calculer le filtre linéaire IIR sur un flux de taille infinie. Une taille infinie n'existe pas en pratique, mais on considère que ce flux est un vecteur de taille M très grande. La formule générale pour le calcul du filtre est :

$$IIR(n) = \sum_{i=0}^{min(n,F)} b_i x(n-i) - \sum_{j=1}^{\min(n,B)} a_j IIR(n-j), n \in \mathbb{N} \tag{B.1}$$

L'objectif est d'obtenir un calcul du filtre qui soit parallélisable sur une architecture de type SIMD. Pour cela, on utilise un calcul du filtre IIR en bloc, sans dépendance entre les blocs :

$$IIR_t(n) = \sum_{i=0}^{F} b_i x(tN+n-i) - \sum_{j=1}^{\min(n,B)} a_j IIR_t(n-j), \forall n \in \{0 \dots N-1\} \tag{B.2}$$

Ce calcul s'effectue sur un vecteur de taille N. On pose comme condition que $N > B$. On cherche à montrer :

$$\begin{cases} IIR(tN+n) & = IIR_t(n) - \sum_{j=1}^{B} c_j(n) IIR(tN-j) \\ c_j(0) & = a_j \\ c_j(n) & = a_{j+n} - \sum_{k=1}^{n} a_k c_j(n-k) \end{cases} \quad \forall n \in \{0 \dots N-1\}, \forall t \in \mathbb{N} \tag{B.3}$$

Pour simplifier les notations dans la suite, on définit a_j sur $\{1 \dots N\}$, en définissant :

$$a_j = 0, j > B \tag{B.4}$$

B.2 Démonstration

On cherche donc à démontrer que la formule (B.3) est vraie. On distingue pour cela deux cas pour la valeur de t.

Dans le premier cas, on a $t = 0$. La démonstration est alors triviale.

$$IIR(n) = IIR_0(n), n = 0 \dots N-1 \tag{B.5}$$

Dans le deuxième cas, on a $t > 0$. On utilise une récurrence sur n.

Le cas initial $n = 0$ peut se montrer relativement simplement. On a $t > 0$ et $N > B$ par hypothèse, donc $\min(tN, B) = B$. Si on applique la définition de IIR_t et IIR, on obtient :

$$\begin{aligned} IIR_t(0) &= \sum_{i=0}^{F} b_i x(tN - i) \\ IIR(tN) &= \sum_{i=0}^{F} b_i x(tN - i) - \sum_{j=1}^{B} a_j IIR(tN - j) \\ &= IIR_t(0) - \sum_{j=1}^{B} a_j IIR(tN - j) \end{aligned} \tag{B.6}$$

On suppose maintenant que la relation est vraie pour tous les rangs strictement inférieur à n. On va montrer qu'alors elle est vraie pour le rang n.

Par définition de IIR_t, on peut écrire :

$$IIR_t(n) = \sum_{i=0}^{F} b_i x(tN + n - i) - \sum_{j=1}^{\min(n,B)} a_j IIR_t(n - j)$$

En utilisant la formule de récurrence pour tous les points d'indice $n - 1, \ldots, n - \min(n, B)$, on obtient :

$$\begin{aligned} IIR_t(n) &= \sum_{i=0}^{F} b_i x(tN + n - i) \\ &- \sum_{j=1}^{\min(n,B)} a_j \left(IIR(tN + n - j) + \sum_{k=1}^{B} c_k(n - j) IIR(tN - k) \right) \end{aligned} \tag{B.7}$$

On en déduit :

$$IIR_t(n) + \sum_{j=1}^{\min(n,B)} a_j \sum_{k=1}^{B} c_k(n-j) IIR(tN-k) = \sum_{i=0}^{F} b_i x(tN+n-i) - \sum_{j=1}^{\min(n,B)} a_j IIR(tN+n-j) \tag{B.8}$$

Par définition de IIR au point $tN + n$ on a :

$$IIR(tN + n) = \sum_{i=0}^{F} b_i x(tN + n - i) - \sum_{j=1}^{\min(tN+n,B)} a_j IIR(tN + n - j)$$

Comme $t \geq 1$ et $N \geq B$, on a $\min(tN + n, B) = B$, d'où :

$$IIR(tN + n) = \sum_{i=0}^{F} b_i x(tN + n - i) - \sum_{j=1}^{B} a_j IIR(tN + n - j) \tag{B.9}$$

$$\begin{aligned} &= \sum_{i=0}^{F} b_i x(tN + n - i) - \sum_{j=1}^{min(n,B)} a_j IIR(tN + n - j) \\ &- \sum_{j=\min(n,B)+1}^{B} a_j IIR(tN + n - j) \end{aligned} \tag{B.10}$$

On distingue ici deux cas, $n < B$ et $n \geq B$.

Dans le premier cas, on a $n < B$, c'est à dire $\min(n, B) = n$. On peut donc simplifier l'équation (B.10) :

$$\begin{aligned} IIR(tN+n) &= \sum_{i=0}^{F} b_i x(tN+n-i) - \sum_{j=1}^{n} a_j IIR(tN+n-j) \\ &\quad - \sum_{j=n+1}^{B} a_j IIR(tN+n-j) \end{aligned} \tag{B.11}$$

En modifiant l'indice de la deuxième somme, on obtient :

$$\begin{aligned} IIR(tN+n) &= \sum_{i=0}^{F} b_i x(tN+n-i) - \sum_{j=1}^{n} a_j IIR(tN+n-j) \\ &\quad - \sum_{j=1}^{B-n} a_{j+n} IIR(tN-j) \end{aligned} \tag{B.12}$$

En injectant (B.8) avec $\min(n, B) = n$, on s'abstrait de la dépendance aux IIR pour le t courant :

$$\begin{aligned} IIR(tN+n) &= IIR_t(n) + \sum_{j=1}^{n} a_j \sum_{k=1}^{B} c_k(n-j) IIR(tN-k) \\ &\quad - \sum_{j=1}^{B-n} a_{j+n} IIR(tN-j) \end{aligned} \tag{B.13}$$

On développe et on inverse l'ordre de la double somme pour obtenir :

$$\begin{aligned} IIR(tN+n) &= IIR_t(n) + \sum_{k=1}^{B} \sum_{j=1}^{n} a_j c_k(n-j) IIR(tN-k) \\ &\quad - \sum_{j=1}^{B-n} a_{j+n} IIR(tN-j) \end{aligned} \tag{B.14}$$

On échange les indices j et k de la double somme :

$$\begin{aligned} IIR(tN+n) &= IIR_t(n) + \sum_{j=1}^{B} \sum_{k=1}^{n} a_k c_j(n-k) IIR(tN-j) \\ &\quad - \sum_{j=1}^{B-n} a_{j+n} IIR(tN-j) \end{aligned} \tag{B.15}$$

D'après (B.4), on a $a_j = 0$ pour $j > B$. On a donc $a_{j+n} = 0$ pour $j > B - n$, d'où :

$$IIR(tN+n) = IIR_t(n) - \sum_{j=1}^{B} \left(a_{j+n} - \sum_{k=1}^{n} a_k c_j(n-k) \right) IIR(tN-j) \tag{B.16}$$

Si on s'intéresse maintenant au deuxième cas, on a $n \geq B$, et donc $\min(n, B) = B$. On peut donc simplifier l'équation (B.10) :

$$IIR(tN+n) = \sum_{i=0}^{F} b_i x(tN+n-i) - \sum_{j=1}^{B} a_j IIR(tN+n-j)$$

On injecte (B.8) avec $\min(n, B) = B$, ce qui donne :

$$IIR(tN+n) = IIR_t(n) + \sum_{j=1}^{B} a_j \sum_{k=1}^{B} c_k(n-j) IIR(tN-k) \quad \text{(B.17)}$$

On développe, on inverse l'ordre de la double somme et on échange les indices j et k (*cf.* (B.14) et (B.15)), ce qui nous donne :

$$IIR(tN+n) = IIR_t(n) + \sum_{j=1}^{B} \sum_{k=1}^{B} a_k c_j(n-k) IIR(tN-j) \quad \text{(B.18)}$$

On a $n > B$, donc $j + n > B \forall j \in \mathbb{N}$. En utilisant (B.4), on obtient donc que $a_{j+n} = 0$ dans ce cas. On a donc

$$a_{j+n} - a_k c_j(n-k) IIR(tN-j) = -a_k c_j(n-k) IIR(tN-j) \quad \text{(B.19)}$$

De même, en utilisant toujours (B.4), et comme $n > B$, on a :

$$\sum_{k=1}^{B} a_k m = \sum_{k=1}^{n} a_k m \quad \text{(B.20)}$$

En utilisant (B.19) et (B.20) avec (B.18), on obtient :

$$IIR(tN+n) = IIR_t(n) - \sum_{j=1}^{B} \left(a_{j+n} - \sum_{k=1}^{n} a_k c_j(n-k) \right) IIR(tN-j) \quad \text{(B.21)}$$

On obtient donc, dans les deux cas, une formule de la forme :

$$IIR(tN+n) = IIR_t(n) - \sum_{j=1}^{B} c_j(n) IIR(tN-j) \quad \text{(B.22)}$$

avec

$$c_j(n) = a_{j+n} - \sum_{k=1}^{n} a_k c_j(n-k) \quad \text{(B.23)}$$

La formule est donc vraie au rang n.

Comme la formule est vraie au rang 0, et qu'elle est vraie au rang n si elle est vraie pour tout rang strictement inférieur à n, elle est vraie quelque soit $n \in \mathbb{N}$.

B.3 Conclusion

Cette méthode permet d'obtenir un gain pour le calcul d'un filtre IIR en utilisant le GPU.

Annexe C

Présentation générale de l'interface de FRK

Ce chapitre est consacré à l'API de FRK. On donne les structures et les fonctions principales des différentes fonctions constituant l'environnement, avec leur effet et leur objectif. Ce chapitre est organisé librairie par librairie, avec une section supplémentaire qui détaille les méthodes de *monitoring*. On ne donne pas ici les implémentations, qui peuvent dépendre de la plateforme, uniquement les prototypes.

C.1 OPM et communications

C.1.1 Structures

C.1.1.1 Opération

```
typedef struct frk_op_s frk_op_t;
struct frk_op_s {
    unsigned int id;
    int num_inputs;
    int *minimal_inputs;
    frk_op_s *inputs;
    int num_outputs;
    int *minimal_outputs;
    frk_op_s *outputs;
    int required_throughput;
    rhal_config_t *implem;
};
```

Une opération est un type générique utilisé pour représenter n'importe quelle opération de FRK. Elle est constituée de huit éléments. L'identifiant `id` est unique pour chaque type d'opération, défini à l'aide d'un `enum`. De manière générale, toutes les structures de FRK ont un identifiant. Les six paramètres suivants permettent de représenter le graphe en donnant, pour les entrées puis les sorties, le nombre, la taille minimale des données (qui permet d'obtenir le ratio entrées/sorties minimal), et les opérations avec lesquels il y a connexion. L'avant-dernier paramètre n'est pas encore réellement utilisé, il permet de définir le débit minimal requis pour cette opération afin de sélectionner une implantation adaptée. Le dernier paramètre permet de renseigner l'implantation choisie pour une opération. Il est utilisé, entre autres, pour permettre aux applications de haut-niveau de ne fonctionner qu'avec les opérations génériques, sans pour autant perdre le lien avec l'implantation au niveau de la R-HAL. Ce type des opérations génériques est ensuite étendu afin de permettre une spécialisation, comme pour l'opération `name`.

```
typedef struct frk_name_s frk_name_t;
struct frk_name_s {
```

```
    frk_op_t param;
    // Parametres specifiques de l'operation name
};
```

C.1.1.2 Waveform

La *waveform* est l'application générique de FRK. On donne ici sa structure, même si elle n'est pas nécessairement utilisée par le développeur, qui peut utiliser les fonctions proposées. La structure comporte un numéro d'identifiant comme toutes les structures de FRK. Elle comporte également le nombre d'éléments de la *waveform*, les éléments étant les opérations. Ces éléments sont ensuite donnés dans `op_list`. Ce paramètre représente un tableau d'opérations, mais ces opérations peuvent également être adressées comme des listes, puisque l'application a une structure de graphe et que le type `frk_op_t` contient les liaisons entre les différentes opérations.

```
typedef struct frk_wf_s frk_wf_t;
struct frk_wf_s {
    unsigned int wid;
    unsigned int num_elements;
    frk_op_t *op_list;
    frk_ci_t *implem;
};
```

C.1.1.3 CI

La traduction de la *waveform* permet d'obtenir la représentation de l'application au niveau R-HAL. Cette représentation s'appelle la CI. La structure de la CI est similaire à la structure de la *waveform* à quelques détails près :

- on ne travaille pas sur des opérations mais sur des `rhal_config_t`, qui correspondent aux opérations de la R-HAL ;
- les opérations sont différenciées en fonction de la cible qui va les exécuter ;
- les entrées et sorties sont marquées, elles sont tout de même enregistrées dans la liste globale mais on a besoin de pouvoir y accéder rapidement ;
- les canaux de transferts de données sont explicites, sous la forme d'une liste de canaux ;
- la structure de graphe est perdue (en fait, elle peut se retrouver en utilisant le pointeur sur l'opération générique associée).

```
typedef struct frk_ci_s frk_ci_t;
struct frk_ci_s {
    unsigned int id;
    unsigned char loaded; /* boolean */
    unsigned char active; /* boolean */
    rhal_config_t *source_lists;
    rhal_config_t *sink_lists;
    rhal_config_t *config_lists[PLATFORM_NUM_TARGETS];
    rhal_channel_t *comm_list;
    frk_wf_t *waveform;
};
```

C.1.2 Fonctions

C.1.2.1 Opérations

Les premières fonctions de l'OPM sont les fonctions de gestion des opérations. Elles permettent, entre autres, d'initialiser (et de nettoyer) ces dernières. Elles ont les prototypes suivants :

```
int init_name(frk_name_t *operation);
int clean_name(frk_name_t *operation);
```

L'initialisation permet de générer certains paramètres comme l'ID, que l'on veut masquer par simplicité à l'utilisateur. Les paramètres spécifiques de l'opération sont à mettre en place par l'utilisateur.

C.1.2.2 Création de la waveform

Afin de créer la *waveform* à partir des opérations proposées, on fournit tout d'abord une fonction de connection des opérations, qui permet de construire les arrêtes du graphe. Cette fonction est la même que la fonction `connect` de GNURadio. On donne le prototype dans FRK.

```
int connect (frk_op_t *source, int input_id, frk_opt_t *sink, int
    output_id);
```

On offre également des fonctions de génération de *waveform*, qui permettent d'initialiser la structure (génération d'un identifiant unique, allocation des structures, ...), et d'ajouter les opérations à la *waveform*.

```
int init_waveform(frk_wf_t *waveform, int num_ops);
int add_op(frk_op_t *operation, frk_wf_t *waveform);
```

C.1.2.3 Traduction

Finalement, deux fonctions de traduction sont fournies qui permettent pour l'une de générer complètement une CI à partir d'une *waveform*, pour l'autre de modifier une opération.

```
int translate(frk_wf_t *waveform, rhal_config_t *ci);
int update(frk_wf_t *waveform, frk_op_t *operation);
```

La fonction `update` pour les modifications ponctuelles permet en fait de prendre en compte des modifications déjà effectuées. L'opération doit être présente dans la *waveform*. Une fonction de remplacement d'une opération par une autre existe.

```
int replace(frk_wf_t *waveform, frk_op_t *old, frk_op_t *new);
```

C.2 Interface R-HAL

La R-HAL permet d'abstraire les unités d'exécution qui vont être utilisées. Elle définit en particulier les fonctions qui seront utilisables par les couches supérieures, l'interface avec l'OPM, ainsi que certaines fonctions pour les TaME.

C.2.1 Structures

C.2.1.1 Transferts de données

Contrairement à l'OPM qui définit les opérations génériques, la R-HAL ajoute de manière explicite les transferts de données. Ces transferts ne sont pas implantés au niveau de l'interface, mais au niveau du TaME. Cependant, afin de permettre l'interaction entre les différentes cibles, une structure générique de transferts de données, contenant principalement l'ensemble des fonctions permettant d'agir sur le transfert ainsi qu'un identifiant. On donne également un lien avec la cible qui sera responsable des transferts. Ce lien n'est pas forcément nécessaire, mais permet de simplifier la gestion des applications au niveau de l'interface. La structure `rhal_channel_t` est un héritage des premières versions de FRK, et n'est plus réellement nécessaire. L'objectif de cette structure était à l'origine de permettre une abstraction du canal de communication, quand elle n'était pas nécessairement gérée par la cible, pour les FIFO génériques. Cependant, dans les nouvelles interfaces, `tame_channel_t` est déja une abstraction du canal, qui est ensuite étendue en fonction de la cible. Elle est vouée à disparaître, afin d'être remplacée par une version étendue de `tame_channel_t`. Un ajout d'une cible générique est en effet au programme, pour mutualiser certaines opérations.

```
typdef struct rhal_channel_s rhal_channel_t;
struct rhal_channel_s {
    unsigned int id;
    unsigned int target_id_source;
    unsigned int target_id_sink;
    unsigned int width;
    tame_channel_t *channel;
    int (*read) (tame_channel_t, void *, int);
    int (*write) (tame_channel_t, void *, int);
    int (*deactivate) (tame_channel_t);
    int (*activate) (tame_channel_t);
}
```

Les quatre fonctions ne sont pas les seules fonctions pour gérer une FIFO. Cependant, ce sont les seules dont l'opération qui va les utiliser peut avoir besoin. On définit le type `tame_channel_t` dans la section C.3. Les fonctions `read` et `write` sont les fonctions qui permettent de lire et écrire dans la FIFO. Cette lecture (resp. écriture) se fait dans (resp. depuis) le *buffer* définit comme second argument des fonctions. La taille du transfert est définie par le troisième argument. La largeur `width` de ce transfert est fixée pour un transfert. Elle est de toute manière rappelée dans `tame_channel_t`.

On notera que cette structure va se retrouver dans deux opérations différentes. Il y a donc nécessité de protéger correctement l'appel aux différentes fonctions. Ceci est fait au niveau du TaME.

C.2.1.2 Opérations

La R-HAL définit également l'implantation sur cible.

```
typedef struct rhal_config_s rhal_config_t;
struct rhal_config_s {
    unsigned int cid;
    unsigned int target_id;
    unsigned char state; /* Charge ou non ? */
    unsigned char active; /* Active ou non ? */
```

```
    frk_op_t *operation;
    frk_ci_t *ci;
    /* I/O */
    rhal_channel_t *inputs;
    rhal_channel_t *outputs;
}
```

Au même titre que l'opération générique définie dans l'OPM, cette implantation est générique pour toutes les cibles. Elle est étendue par chaque cible afin de prendre en compte d'éventuelles arguments supplémentaires. On donne dans les opérations R-HAL un rappel de l'opération générique `operation`, ainsi que l'identifiant de la cible de cette opération. Ceci permet de faire appel aux fonctions du TaME associé à l'opération. On ne rappelle pas dans cette structure le nombre d'entrées et de sorties pour l'opération, puisqu'il est défini dans l'opération générique.

C.2.2 Fonctions

```
int load_ci (frk_ci_t *);
int unload_ci (frk_ci_t *);
int activate_ci (frk_ci_t *);
int deactivate_ci (frk_ci_t *);
int get_info (unsigned int info_code, unsigned int *info_size, void **
   info);
int set_prioritarty(frk_ci_t *);
int set_daemon(frk_ci_t *);
```

Les fonctions implantées dans la R-HAL sont des fonctions liées à la gestion de la CI. Chacune de ces fonctions exécute des éléments propres à la R-HAL, avant d'appeler les fonctions correspondantes pour les cibles de la plateforme. Tout d'abord, les fonctions de chargement `load_ci` et déchargement `unload_ci` d'applications. Ces fonctions permettent de charger les différentes opérations en appelant les fonctions des cibles associées, et d'initialiser les files associées. Les fonctions d'activation `activate_ci` et de désactivation `deactivate_ci` permettent, comme leur nom l'indique, d'activer et de désactiver une CI. La fonction `update` a déjà été définie, elle s'implante en fait au niveau de la R-HAL Une fonction de récupération d'information `get_info` permet de récupérer certaines informations prédéfinies, en fonction du code de l'information requise `info_code`. La taille de la réponse est donnée dans `info_size`, et la réponse elle même est donnée dans `info`. Finalement, deux fonctions permettent de modifier le type de la CI, à savoir la rendre prioritaire (`set_prioritary`) ou démon (`set_daemon`).

C.3 TaME

On donne dans cette section les structures à utiliser et les fonctions à implanter pour intégrer une nouvelle cible dans FRK. Les extensions de la HAL sont propres à chaque cible, et il n'y a pas d'interface définie pour cette partie. Seule le TaME sera utilisé par les couches supérieures, et elle devra être implanté pour chaque cible. On notera cependant que même si pour chaque plateforme, on définit une nouvelle cible, beaucoup d'éléments peuvent être réutilisés. Ceci est particulièrement vrai pour la cible des accélérateurs matériels.

C.3.1 Structures

C.3.1.1 Structure d'un TaME

La structure principale de TaME est sa propre représentation. En effet, une instance de FRK pouvant contenir plusieurs TaME, pour garder une certaine généricité, il est nécessaire d'encapsuler la description d'une cible dans une structure.

```
typdef struct frk_tame_s frk_tame_t;
struct frk_tame_s {
    unsigned int id;
    /* Infos etat */
    unsigned char state; /* Utilisable ou non */
    /* Fonctions CI*/
    int (*register_ci) (frk_ci_t *);
    int (*unregister_ci) (frk_ci_t *);
    int (*activate_ci) (frk_ci_t *);
    int (*deactivate_ci) (frk_ci_t *);
    int (*set_prioritary) (frk_ci_t *);
    int (*set_daemon) (frk_ci_t *);
    /* Fonctions operations*/
    int (*init_config) (rhal_config_t *);
    int (*load_config) (rhal_config_t *);
    int (*unload_config) (rhal_config_t *);
    int (*activate_config) (rhal_config_t *);
    int (*deactivate_config) (rhal_config_t *);
    int (*set_priority) (rhal_config_t *, int);
    int (*set_mode) (rhal_config_t *, int);
    int (*get_state) (rhal_config_t *, unsigned char *state);
    /* Gestion des transferts de donnees */
    int (*register_channel) (rhal_channel_t *, rhal_config_t *);
    int (*unregister_channel) (rhal_channel_t *);
};
```

C.3.1.2 Opérations

Dans les structures accessibles de la TaME, on trouve principalement la structure d'une opération pour cette cible. Dans les faits, cette structure peut également être étendue. L'exemple de la cible des accélérateurs matériels, dans laquelle plusieurs accélérateurs peuvent être regroupés malgré le fait qu'ils aient une interface différente, est caractéristique, on donne donc les instances pour ce TaME et pour le codeur convolutif présenté dans le chapitre 5. Ce codeur ne nécessite pas réellement d'extension. On donne ici la représentation d'un accélérateur dans le TaME, et la représentation d'une opération pour ce TaME.

```
typedef struct coder_magnet_config_s coder_magnet_config_t;
typedef struct coder_magnet_device_s coder_magnet_device_t;

struct magnet_config_s {
    rhal_config_t gen;
    unsigned int device_id; /* Cible de la configuration */
    unsigned char state; /* Actif ou non */
    unsigned char loaded; /* charge ou non */
    unsigned char candidate;
    unsigned char executable;
```

```
    unsigned int num_constant;
    unsigned int num_volatile;
    unsigned int *constant_val;
    unsigned int *volatile_val;
};

struct magnet_device_s {
    unsigned int device_id;
    unsigned char state; /* Actif ou non */
    magnet_config_t *running;
    unsigned int *base_address;
    unsigned int num_constant;
    unsigned int num_volatile;
    unsigned int **constant_addr;
    unsigned int **volatile_addr;
};
```

La représentation de l'accélérateur est donnée à titre indicatif, puisqu'elle n'est pas nécessaire pour l'utilisateur. Cette représentation est propre à la plateforme.

C.3.1.3 Transferts de données

La structure pour les transferts de données contient également l'intégralité des fonctions requises. Ces fonctions sont décrites plus précisément dans la partie adéquate.

Comme nous l'avons évoqué en présentant `rhal_channel_t`, la structure des *First In First Out* (FIFO) est amenée à changer, à cause de son manque de cohérence. Il y a donc beaucoup de points communs. Chaque FIFO a un *id* unique, et définit sa largeur (taille d'un élément) et sa profondeur (nombre d'éléments). Quatre fonctions sont renseignées lors de la connexion de la FIFO aux éléments qui vont l'utiliser. Les fonctions `deactivate_prod` et `activate_prod` sont utilisées pour désactiver et activer l'opération qui écrit dans la FIFO. Comme les méthodes d'activation et de désactivation des éléments sont propres à chaque cible, il est nécessaire de définir ces fonctions dans la structure accessible par les fonctions `read` et `write`. Le fonctions `activate_cons` et `deactivate_cond` sont utilisées pour l'opération qui lit la FIFO.

La structure `tame_channel_s` est commune à tous les TaME. Elle peut cependant être étendue pour les besoins spécifiques de certaines cibles, comme par exemple les coprocesseurs qui ont besoin des seuils.

```
struct tame_channel_s {
    int id;
    int width;
    int depth;
    rhal_channel_t *gen;
    int (*deactivate_prod)();
    int (*activate_prod)();
    int (*deactivate_cons)();
    int (*activate_cons)();
    /* Fonctions de la FIFO */
    int read(tame_channel_t *fifo, void *buffer, int size);
    int write(tame_channel_t *fifo, void *buffer, int size);
    int purge(tame_channel_t *fifo);
    int create(tame_channel_t **fifo); //fifo non allouee
    int activate(tame_channel_t *fifo);
    int deactivate(tame_channel_t *fifo);
    int register_fifo(tame_channel_t *fifo);
```

```
};

struct magnet_channel_s {
    tame_channel_t channel;
    int min_threshold;
    int med_threshold;
    int max_threshold;
}
```

C.3.2 Fonctions

Les fonctions du TaME sont toutes représentées dans la structure descriptive `frk_tame_t`. On décrit plus précisément leurs effets ici.

C.3.2.1 CI

Afin de permettre la gestion de la CI dans son ensemble, les TaME doivent implanter quatre fonctions. `register_ci` sert à enregistrer une CI au niveau du TaME. Ceci permet d'initialiser par exemple le verrou associé à la CI pour la cible logicielle. L'enregistrement de la CI ne charge pas les opérations. `unregister_ci` permet de libérer l'espace pris par la CI dans la cible. Les fonctions `activate_ci` et `deactivate_ci` servent à activer ou désactiver la CI (en prenant le verrou pour la cible logicielle). Les fonctions `set_prioritary` et `set_daemon` sont redondantes avec la fonction `set_priority` des opérations, et servent à marquer d'un coup une CI comme prioritaire.

C.3.2.2 Opérations

Les fonctions liées aux opérations sont utilisées par la R-HAL pour répercuter les actions sur les cibles. La fonction `init_config` permet, comme son nom l'indique, d'initialiser une configuration pour la cible. Cette fonction n'est utilisée que pour des tests, l'initialisation de la configuration étant géré dans l'OPM.

Les fonctions `load_config` et `unload_config` permettent d'ajouter une configuration (ou de la retirer) dans la liste des opérations à traiter par le TaME. Les fonctions `activate_config` et `deactivate_config` ne s'appliquent que sur une opération chargée. Elles permettent de la rendre exécutable ou non exécutable sur la cible, suite par exemple à la désactivation d'une CI.

Les trois dernières fonctions liées aux opérations servent à modifier certains paramètres des opérations. `set_priority` n'est pas encore implantée, mais sera utiliser pour gérer une priorité par CI dans FRK. `set_mode` permet de faire passer une opération en mode prioritaire ou en mode démon. `get_state` est la seule fonction de *monitoring* passif implantée pour l'instant, d'autres suivront en fonction des besoins lors du développement de la PL. FRK propose également un mécanisme de *monitoring* actif, détaillé dans la section C.4.

C.3.2.3 Transferts de données

Finalement, les dernières fonctions présentées ici sont les fonctions liées au transfert de données. Pour chaque type de FIFO implantées, les fonctions présentées dans la suite doivent être implantées, à l'exception de **`register`**. On notera que l'implantation peut ne rien faire si elle ne présente pas d'intérêt pour le cas présenté. Les fonctions de synchronisation dont nous avons discuté au chapitre 5, utilisées pour les FIFO génériques, sont en fait une instanciation des fonctions ci-après. Chaque cible implante donc au minimum les fonctions pour les FIFO génériques, et ajoutent ensuite autant d'implantation qu'elle le souhaite.

```
int read(tame_channel_t *fifo, void *buffer, int size);
int write(tame_channel_t *fifo, void *buffer, int size);
int purge(tame_channel_t *fifo);
int create(tame_channel_t **fifo); //fifo non allouee
int activate(tame_channel_t *fifo);
int deactivate(tame_channel_t *fifo);

int register_fifo(tame_channel_t *fifo);
```

Les premières fonctions sont explicites. `purge` permet de vider une FIFO pour des raisons de "maintenance", quand il y a saturation. `create` est utilisée lors de l'initialisation des canaux, au chargement. `activate` et `deactivate` permet de débloquer ou de bloquer une FIFO. Ceci est utile quand on veut empêcher une opération d'être approvisionné en donner. C'est une méthode pour désactiver un coprocesseur matériel, par exemple, si celui-ci n'a pas de support pour la désactivation ;

`register_fifo` est plus complexe. Elle est optionnelle, et est utilisée si besoin lors de l'appel aux fonctions générales d'enregistrement des FIFO définies dans la TaME.

```
int register_channel(rhal_channel_t *fifo, rhal_config_t *source,
   rhal_config_t *sink);
int unregister_channel(rhal_channel_t *fifo);
```

Quand l'enregistrement requiert une action au niveau de la FIFO, la fonction `register_fifo` permet d'effectuer cette action. Les fonctions `register_channel` et `unregister_channel` permettent d'enregistrer une FIFO au niveau d'une cible, en précisant quelle opération sera productrice (`source`), et quelle opération sera consommatrice (`sink`). Elles sont utilisées lors du chargement d'une CI.

C.4 *Monitoring*

On finit ce chapitre présentant l'API par une présentation rapide du principe du *monitoring* dans FRK. Implanter le *monitoring* dans FRK n'est pas évident, du fait de la construction sous forme de librairie de l'environnement.

Cet aspect de FRK n'est pas encore extrêmement développé, mais il est basé sur le principe de *handlers* fournis par les fonctions appelantes. FRK offre à l'extérieur une structure permettant d'enregistrer un certain nombre de fonctions. Ces fonctions sont enregistrées pour être appelées à des moments précis de l'exécution. En l'état actuel de l'environnement, trois points sont disponibles. D'autres seront mis à disposition au fur et à mesure des besoins. On ne peut pour l'instant qu'enregistrer une fonction par condition.

```
struct frk_mon_s {
    int (*saturation)();
    int (*read_alert)();
    rhal_channel_t *read_pointer;
    int (*switch)();
};
```

Elles sont définies sans paramètres pour l'instant. La première est appelée dès qu'il y a saturation dans FRK. On définit par saturation le non-respect du débit attendu. Dans l'état actuel de FRK, les notions de temps n'étant pas encore mises en place, la saturation correspond à la perte de données dans un élément non contrôlable. Si une opération lit ses données dans une FIFO remplie

par une entrée du système que l'on ne peut pas arrêter, et que cette opération ne suit pas la cadence, il y a saturation.

La seconde est appelée dès qu'une lecture est faite dans la FIFO `read_pointer`. La dernière permet de signaler qu'une commutation a eu lieu dans une cible matérielle.

C.5 Résumé et conclusion

On résume la définition des structures et des fonctions sous la forme du diagramme de classe proposé en figure C.1. Les fonctions ne sont pas représentées sur ce diagramme.

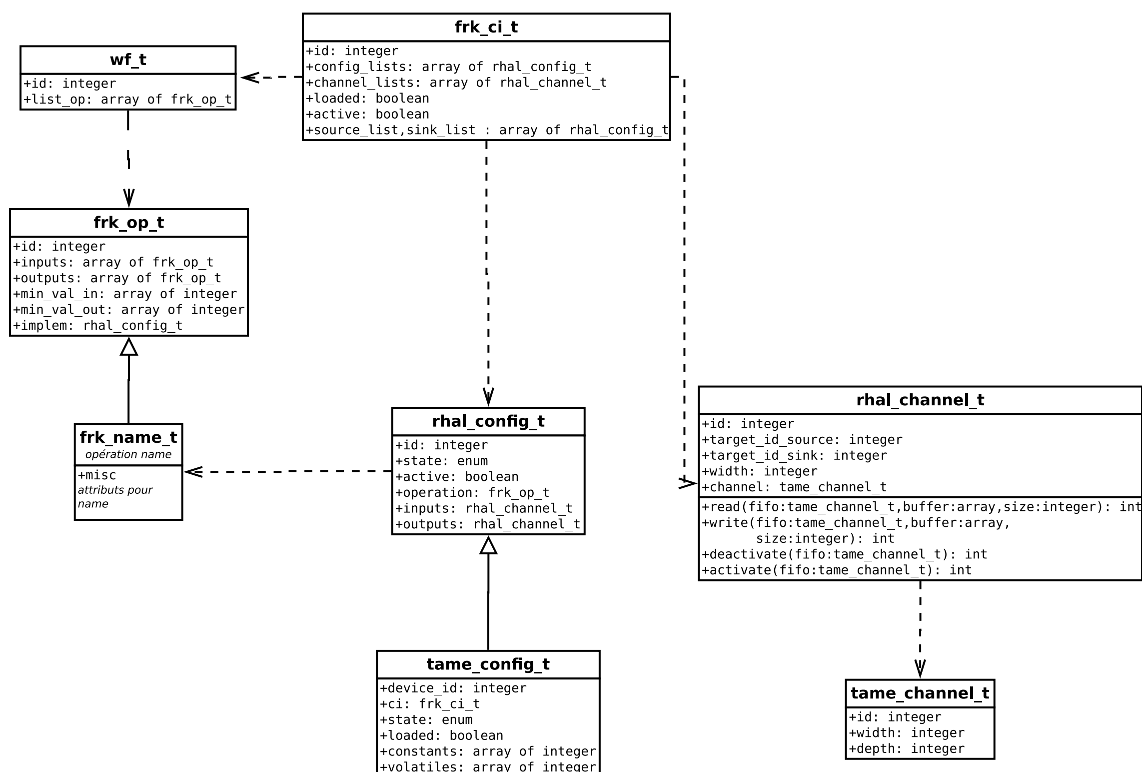


FIGURE C.1 – Diagramme de classe des structures de FRK

L'API de FRK dans sa version actuelle a été présenté dans ce chapitre. Cette API est fonctionnelle, mais est amenée à évoluer au fur et à mesure des évolutions de l'environnement. L'idée directrice de répartition en cibles et en couches restera cependant. D'autres fonctions sont disponibles dans FRK, pour du test et du développement, mais ne sont pas présentées ici.

Annexe D

Intégration de matériel dédié dans FRK

D.1 Utilisation de MAGNET

On présente ici une utilisation détournée de FRK pour le contrôle de matériel dédié. L'OPM permet en effet de rajouter des opérations spécifiques à une plateforme, pas nécessairement portables sur d'autres plateformes. Ces opérations sont typiquement des chaînes de communication non normalisées, mais complètes. On présente ici l'intégration d'une chaîne de communication utilisée pour un PAN à haut débit, et le contrôle d'un protocole de retransmission associé. Le PAN utilisé est issu du projet européen MAGNET On trouvera dans [Pra10] des informations sur l'application radio associée.

D.1.1 Plateforme MAGNET

La chaîne de communication de MAGNET est implantée en matériel. D'un point de vue pratique, elle est actuellement déployée sur le FPGA de la plateforme basée sur le Atmel AT91RM9200 présentée au chapitre 5, dans la figure 5.13. L'application MAGNET est basée sur une modulation *Orthogonal Frequency Division Multiplexing* (OFDM). Elle permet de faire varier le débit, en jouant en particulier sur la modulation utilisée (*Quadrature Phase Shift Keying* (QPSK), 16-*Quadrature Amplitude Modulation* (QAM), 64-QAM). Le 64-QAM est prévu, mais difficilement fonctionnel. Un codage canal est également utilisé. C'est un code convolutif à trois additionneurs. Le code en lui-même est fixe, cependant, il est poinçonnable afin d'autoriser une modification du rendement (et donc du débit). Côté réception, on utilise des *soft-bits* et un décodeur Viterbi.

D.1.2 Représentation

L'implantation matérielle de MAGNET est un périphérique, du point de vue du processeur. Ce périphérique est branché sur le bus des extensions du système AT91. Deux intégrations du composant dans le noyau Linux du composant sont disponibles, sans FRK Une première version de démonstration réside dans l'espace utilisateur. Elle *mappe* les adresses du composant directement dans l'espace adressable. Ces adresses s'utilisent ensuite comme des pointeurs de base. Ce mode de fonctionnement se fait nécessairement en utilisant la scrutation (*polling*) pour vérifier l'état du composant. Une deuxième version utilise l'interface *device* du noyau, et permet l'utilisation des interruptions. Cette intégration se fait donc dans l'espace noyau.

Le périphérique MAGNET est un composant dédié, qui ne peut réaliser que le protocole MAGNET. Il est intégré dans FRK en tant que périphérique dédié. Plus particulièrement, il est intégré en tant que deux accélérateurs dédiés, l'un pour l'émission, l'autre pour la réception. Ces deux accélérateurs sont respectivement vus comme une entrée et une sortie.

Au niveau de l'OPM, on rajoute donc une opération dédiée implantable uniquement sur la cible MAGNET. Les opérations dédiées ont par convention dans FRK un identifiant "négatif". L'identifiant étant strictement positif, l'identifiant négatif est en fait un identifiant dont le premier bit est un 1. On donne la création de l'opération d'émission, avec seulement certains paramètres. Plus de paramètres sont disponibles dans l'opération réelle, mais n'apportent pas grand chose à la

compréhension du mécanisme. On donne dans cette section quelques éléments représentatifs de l'implantation.

```
struct frk_magnettx_s {
    frk_op_t param;
    unsigned short interframe;
    unsigned short packet_size;
    unsigned char rate; /* base sur un enum */
}

init_magnettx(frk_op_t *op) {
    op->id = (unsigned_int)MAGNET_TX_ID;
    op->num_inputs = 1;
    op->num_outputs = 0;
    op->min_val_in = 2;
    op->min_val_out = 0;
    op->required_throughput = 0;
}
```

L'intégration du composant dans FRK utilise les interruptions, et se fait donc partiellement dans l'espace noyau. On crée donc un module spécifique dans le noyau pour le contrôle du composant MAGNET. Ce module correspond aux extensions HAL. On définit plusieurs choses dans ce module.

Tout d'abord, on donne la structure du composant, sous la forme *constantes/volatiles* définie au chapitre 5. L'implantation de l'opération se base sur cette structure. On donne également l'implantation des FIFO pour cette cible. Dans le cas de MAGNET, une FIFO matérielle est disponible [1].

```
struct magnettx_channel_s {
    tame_channel_t fifo;
    unsigned short *magnet_fifo_tx;
    unsigned short *more_buffer;
    int read_idx; /* indice pour more_buffer */
    int write_idx;
    int used; /* Remplissage de more_buffer*/
}

create_magnettx_channel(tame_channel_t *fifo)
{
    magnettx_channel_t *ext_fifo = (magnettx_channel_t *)fifo;
    fifo->id = get_channel_id();
    fifo->width = 2;
    fifo->depth = MAGNET_FIFO_SIZE + ADDITIONNAL_MAGNET_SIZE;
    ext_fifo->magnet_fifo_tx = (unsigned short *)MAGNET_TX_FIFO_ADDR;
    ext_fifo->more_buffer = kmalloc((ADDITIONNAL_MAGNET_SIZE) * sizeof(
        unsigned short), GFP_KERNEL);
}
```

Il n'y a pas de lecture dans la FIFO d'émission, le composant qui va lire est en effet le composant matériel, et la lecture n'est pas contrôlable. On présente donc l'implantation de la fonction `write`. Cette fonction se base sur de la scrutation pour détecter si la FIFO matérielle est pleine ou non. Afin d'augmenter la taille de stockage disponible, on utilise le *buffer* logiciel `more_buffer`. A chaque appel à `write`, on synchronise les deux éléments. On synchronise également quand le

1. il y a en fait deux FIFO, une pour l'émission, et une pour la réception, mais on ne s'intéresse ici qu'à l'émission

flag `almost_empty` de la FIFO matérielle se lève. Ce *flag* génère une interruption qui lance la synchronisation. Lors de l'appel à *write*, la première action est donc de désactiver cette interruption, pour éviter la concurrence. Il n'y a pas de périphériques DMA disponibles avec le composant MAGNET, la copie se fait donc explicitement.

```
int write(tame_channel_t *fifo, void *buffer, int size)
{
    magnettx_channel_t *ext_fifo = (magnettx_channel_t *)fifo;
    unsigned int buffer_idx = 0;
    unsigned short *ext_buffer = (unsigned short *)buffer;

    disable_txfifo_interrupts();
    /* Synchronisation more_buffer/composant */
    if (!magnet_fifo_tx_full()) {
        while(!magnet_fifo_tx_full() && (ext_fifo->used != 0)) {
            *(ext_fifo->maget_fifo_tx) = ext_fifo->more_buffer[ext_fifo
                ->read_idx];
            ext_fifo->read_idx+=1;
            if (ext_fifo->read_idx >= ADDITIONNAL_MAGNET_SIZE) {
                ext_fifo->read_idx = 0;
            }
            ext_fifo->used-=1;
        }
        /* Inutile d'activer ici, on ne peut pas etre desactive */
    }
    /* Synchronisation termine, debut de l'ecriture de buffer */

    while (!magnet_fifo_tx_full() && (buffer_idx < size)) {
        *(ext_fifo->magnet_fifo_tx) = ext_buffer[buffer_idx];
        buffer_idx++;
    }
    while (buffer_idx < size) {
        if (ext_fifo->used < (ADDITIONNAL_MAGNET_SIZE - 1)) {
            ext_fifo->more_buffer[ext_fifo->write_idx] = ext_buffer[
                buffer_idx];
            buffer_idx ++;
            ext_fifo->write_idx++;
            if (ext_fifo->write_idx == ADDITIONNAL_MAGNET_SIZE) {
                ext_fifo->write_idx = 0;
            }
            used++;
        } else {
            enable_txfifo_interrupts();
            magnet_schedule(MAGNETTX_ID);
            fifo->deactivate_prod();
            disable_txfifo_interrupts();
        }
    }
    if ((used + MAGNET_FIFO_SIZE) > tame_fifo->med_threshold) {
        magnet_schedule(MAGNETTX_ID);
    }
    enable_txfifo_interrupts();
}
```

Si il n'y a plus de place pour écrire, la FIFO désactive l'appelant en utilisant la fonction

renseignée à l'initialisation. La fonction de synchronisation appelée en cas d'interruption active automatiquement l'appelant, puisqu'elle libère de l'espace. De même, quand le seuil moyen est atteint, on appelle l'ordonnanceur, qui va vérifier si il est nécessaire de modifier la configuration du composant. `MAGNETTX_ID` est l'identifiant du composant d'émission dans la cible.

D.2 Conclusion

Nous avons montré dans ce chapitre quelques éléments de l'intégration d'un accélérateur spécialisé dans FRK. Cet accélérateur permet d'exécuter toute la chaîne de communication de MAGNET.

Publications

Nous donnons ici la liste des différentes communications ayant été réalisées durant cette thèse.

Conférences internationales

Damien Hedde, Pierre-Henri Horrein, Frédéric Pétrot, Robin Rolland et Franck Rousseau, *"A MPSoC Prototyping Platform for Flexible Radio Applications"*, Digital System Design Conference (DSD), Patras, Grèce, Août 2009

Pierre-Henri Horrein, Christine Hennebert et Frédéric Pétrot, *"A Flexible Bufferless H-ARQ Processor Based on Dataflow Scheduling"*, 1st International Conference on Advances on Cognitive Radio (COCORA), Budapest, Hongrie, Avril 2011

Pierre-Henri Horrein, Christine Hennebert et Frédéric Pétrot, *"Adapting a SDR environment to GPU architectures"*, Wireless Innovation Forum Conference (WinnComm), Bruxelles, Belgique, Juin 2011

Pierre-Henri Horrein, Christine Hennebert et Frédéric Pétrot, *"An Environment for (re)configuration and Execution Management of Flexible Radio Platforms"*, Digital System Design Conference (DSD), Oulu, Finlande, Septembre 2011

Articles de journal

Pierre-Henri Horrein, Christine Hennebert et Frédéric Pétrot, *"Integration of GPU computing in an Software Radio environment"*, Springer Journal of Signal Processing Systems, *accepté*

Pierre-Henri Horrein, Christine Hennebert et Frédéric Pétrot, *"An Environment for (re)configuration and Execution Management of Heterogeneous Flexible Radio Platforms"*, Elsevier MICPRO, *conditionnellement accepté*

Brevet

1 brevet en cours de dépôt par Florian Pebay-Peyroula et Pierre-Henri Horrein.

Poster

Pierre-Henri Horrein, Christine Hennebert et Frédéric Pétrot, *"Designing software radio applications for GPU parallel architecture"*, DEPCP (DATE Friday Workshop), Grenoble, France, Mars 2011

Bibliographie

[Abd10] Riadh Ben Abdallah. *Machine Virtuelle pour la Radio Logicielle.* PhD thesis, INSA Lyon, 2010.

[Alh10] Laurent Alhaus. *Architecture Reconfigurable pour un Équipement Radio Multistandard.* PhD thesis, Université de Rennes 1, 2010.

[alo] ALOE Middleware - FlexNets. http://flexnets.upc.edu/trac, visité le 07/07/2011.

[ANJ+04] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, Ed Komp, and P. Ashenden. Programming Models for Hybrid FPGA-CPU Computational Components : A Missing Link. *Micro, IEEE*, 24(4) :42 – 53, jul. 2004.

[APR+11] L. Alaus, J. Palicot, C. Roland, Y. Louet, and D. Noguet. Promising technique of parameterization for reconfigurable radio, the common operators technique : Fundamentals and examples. *Journal of Signal Processing Systems*, 62 :173–185, 2011. 10.1007/s11265-009-0353-4.

[arc11] [Discuss-gnuradio] GNURadio and CUDA reprised. http://lists.gnu.org/archive/html/discuss-gnuradio/2011-01/msg00220.html visité le 05/07/2011, January 2011.

[ARFD09] Riadh Ben Abdallah, Tanguy Risset, Antoine Fraboulet, and Yves Durand. The radio virtual machine : A solution for sdr portability and platform reconfigurability. In *IEEE International Symposium on Parallel Distributed Processing, 2009*, may 2009.

[ARFM10] Riadh Ben Abdallah, Tanguy Risset, Antoine Fraboulet, and Jérôme Martin. Virtual machine for software defined radio : Evaluating the software vm approach. In *IEEE International Conference on Computer and Information Technology 2010, CIT 2010*, 2010.

[arm] Mali T604 - ARM. http://www.arm.com/products/multimedia/mali-graphics-hardware/mali-t604.php Visité le 17/08/2011.

[ARM05] ARM. ARM1136JF-S and ARM1136J-S Technical Reference Manual, 2005.

[ASM+07] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. R.Cavallaro, and A. Sabharwal. WARP, a Unified Wireless Network Testbed for Education and Research. In *Proceedings of IEEE MSE*, 2007.

[bea] OpenCL Fast Fourier Transform. http://www.bealto.com/gpu-fft.html Visité le 08/08/2011.

[bea10] BeagleBoard-xM Rev C System Reference Manual, april 2010.

[BZZ08] Steve Bernier and Juan Pablo Zamora Zapata. The deployment of software components into heterogeneous SCA platforms. In *SDR 08 Technical Conference and Product Exposition*, 2008.

[CAL] CALIT2. Calradio. http://calradio.calit2.net/index.htm, visited on 30/06/2011.

[CDPR10] Andrew R Cormier, Carl B Dietrich, Jeremy Price, and Jeffrey H Reed. Dynamic reconfiguration of software defined radios using standard architectures. *Physical Communication*, 3(2) :73–80, 2010.

[cog] Cogeu. http://www.ict-cogeu.eu/ Visité le 11/07/2011.

[Cor08] Andrew R. Cormier. Reconfigurable SCA System Development Using Encapsulated Waveform Applications and Components. Master's thesis, Virginia Polytechnic Institute and State University, 2008.

[DBRC05] Maxime Dumas, Louis Bélanger, Sébastien Roy, and Jean-Yves Chouinard. Development of a SCA 3.1 compliant W-CDMA waveform on DSP/FPGA specialized hardware. In *SDR 05 Technical Conference and Product exposition*, 2005.

[DDL10] Mickael L. Dickens, Brian P. Dunn, and J. Nicholas Laneman. Thresholding for Optimal Data Processing in a Software Defined Radio Kernel. In *Proceedings of the Karlsruhe Workshop on Software Radios (WSR)*, March 2010.

[Dic] Mickael L. Dickens. Conversation personnelle, juin et septembre 2011, article à venir.

[DLD11] Mickael L. Dickens, J. Nicholas Laneman, and Brian P. Dunn. Seamless Dynamic Runtime Reconfiguration in a Software-Defined-Radio. In *Proceedings of SDR'11-WinnComm-Europe*, June 2011.

[DMLP05] Jean-Philippe Delahaye, Christophe Moy, Pierre Leray, and Jacques Palicot. Managing dynamic partial reconfiguration on heterogeneous SDR platforms. In *SDR 05 Technical Conference and Product exposition*, 2005.

[DPML07] J.-P. Delahaye, J. Palicot, C. Moy, and P. Leray. Partial Reconfiguration of FPGAs for Dynamical Reconfiguration of a Software Radio Platform. *Mobile and Wireless Communications Summit, 2007. 16th IST*, pages 1–5, July 2007.

[DZD+05] Ioannis Dagres, Andreas Zalonis, Nikos Dimitriou, Konstantinos Nikitopoulos, and Andreas Polydoros. Flexible Radio : A Framework for Optimized Multimodal Operation via Dynamic Signal Design. *EURASIP Journal on Wireless Communications and Networking*, 3 :284–297, 2005.

[fIT03] IEEE Standard for Information Technology. 1003.26 - Portable Operating System Interface (POSIX) - Part 26 : Device Control Application Interface (API) [C Language], 2003.

[GC97] Andrea J. Goldsmith and Soon-Ghee Chua. Variable-Rate Variable-Power MQAM for Fading Channels. *IEEE Transactions on Communications*, Vol. 45, October 1997.

[GMBG10] Ismael Gomez, Vuk Marojevic, Jordi Bracke, and Antoni Gelonch. Performance and Overhead Analysis of the ALOE Middleware for SDR. In *The 2010 Military Communications Conference*. IEEE, 2010.

[GMSG08] Ismael Gomez, Vuk Marojevic, Jose Salazar, and Antoni Gelonch. A Lightweigh Operating Environment for Next Generation Cognitive Radios. In *11th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, 2008.

[gnua] Beagle Board SDR. `www.opensdr.com/node/10`, visité le 05/07/2011.

[gnub] GNU Radio. `http://gnuradio.org`.

[Gue10] Xavier Guerin. *Approche efficace de développement de logiciel embarqué pour des systèmes multiprocesseurs sur puces*. PhD thesis, Institut National Polytechnique de Grenoble, 2010.

[HCS11] Ke He, Louise Crockett, and Robert Stewart. Dynamic Reconfiguration Technologies Based on FPGA in Software Defined Radio System. In *Proceedings of SDR'11-WinnComm-Europe*, June 2011.

[HHP+09] D. Hedde, P.-H. Horrein, F. Petrot, R. Rolland, and F. Rousseau. A MPSoC Prototyping Platform for Flexible Radio Applications. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 559 –566, 27-29 2009.

[HPA01] Denis Hommais, Frédéric Pétrot, and Ivan Augé. A practical tool box for system level communication synthesis. In *CODES*, pages 48–53, 2001.

[HSM+08] George Harrison, Ambrose Sloan, Wilbur Myrick, Joe Hecker, and David Eastin. Polyphase Channelization utilizing General-Purpose computing on a GPU. In *SDR 2008 Technical Conference and Product Exposition*, 2008.

[JCS11] Hyunwoo Ju, Junho Cho, and Wonyong Sung. Memory Access Optimized Implementation of Cyclic and Quasi-Cyclic LDPC Codes on a GPGPU. *Springer Journal of Signal Processing Systems*, 64 :149–159, 2011.

[Jon02] Friedrich Jondral. Parametrization - A technique for SDR implementation. In Walter Tuttlebee, editor, *Software Defined Radios : Enabling Technologies*, pages n–m. Wiley, 2002.

[Jon05] Friedrich K. Jondral. Software-Defined Radio–Basics and Evolution to Cognitive Radio. *EURASIP Journal on Wireless Communications and Networking*, Vol.3 :275–283, 2005.

[JXJQ09] Guan Jianxin, Ye Xiaohui, Gao Jun, and Liu Quan. The Software Communication Architecture specification : Evolution and trends. In *Computational Intelligence and Industrial Applications, 2009. PACIIA 2009. Asia-Pacific Conference on*, volume 2, pages 341 –344, nov. 2009.

[KB05] J. Kulp and M. Bicer. Integrating specialized hardware to JTRS/SCA software defined radios. In *Military Communications Conference, 2005. MILCOM 2005. IEEE*, pages 839 –844 Vol. 2, oct. 2005.

[KH00] Thomas Keller and Lakos Hanzo. Adaptive Modulation Technique for Duplex OFDM Transmission. *Transactions on Vehicular Technology*, vol. 47, september 2000.

[Kha02] Shoab Ahmed Khan. *Digital Design of Signal Processing Systems : A Practical Approach*. Wiley, 2002.

[KHC10] June Kim, Seungheon Hyeon, and Seungwon Choi. Implementation of an SDR System Using Graphics Processing Unit. *IEEE Communication Magazine*, pages 156–162, March 2010.

[KM02] Apostolos A. Kountouris and Christophe Moy. Reconfiguration in software radio systems. 2nd Karlsruhe Workshop on Software Radios, 2002.

[KMR00] Apostolos A. Kountouris, Christophe Moy, and Luc Rambaud. Reconfigurability : A key property in software radio systems. 1st Karlsruhe Workshop on Software Radios, 2000.

[Kut08] Rade Kutil. Parallelization of IIR filters using SIMD extensions. In *15th International Conference on Systems, Signals and Image Processing (IWSSIP)*, pages 65–68, June 2008.

[LNT+09] D. Liu, A. Nilsson, E. Tell, D. Wu, and J. Eilert. Bridging dream and reality : Programmable baseband processors for software-defined radio. *Communications Magazine, IEEE*, 47(9) :134 –140, sep. 2009.

[LP08] Enno Lubbers and Marco Platzner. A portable abstraction layer for hardware threads. In *International Conference on Field Programmable Logic and Applications*, pages 17–22, 2008.

[mal] Annonce ARM Mali T604. `http://blogs.arm.com/multimedia/318-arm-mali-t604-new-gpu-architecture-for-highest-performance-flexibility/`, Visité le 17/08/2011.

[MBGS03] Klaus Moessner, Didier Bourse, Dieter Greifendorf, and Joerg Stammen. Software radio and reconfiguration management. *Computer Communications*, 26(1) :26 – 35, 2003.

[MGT01] K. Moessner, S. Gultchev, and R. Tafazolli. Software Defined Radio reconfiguration management. In *Personal, Indoor and Mobile Radio Communications, 2001 12th IEEE International Symposium on*, volume 1, pages D–91 –D–95 vol.1, sep. 2001.

[MHV+09] Benoît Miramond, Emmanuel Huck, François Verdier, Amine Benkhelifa, Bertrand Granodo, Thomas Lefebvre, Mehdi Aichouch, Jean-Christophe Prevotet, Yaset Oliva, Daniel Chillet, and Sébastien Pillement. OveRSoC : A Framework for the Exploration of RTOS for RSoC Platforms. *International Journal of Reconfigurable Computing*, 2009.

[Mil10] Joel Gregory Millage. GPU integration into a Software Defined Radio Framework. Master's thesis, Iowa State University, 2010.

[MIP92] MIPS. Assembly Language Programmer's Guide, 1992.

[Mit92] III Mitola, J. Software radios-survey, critical evaluation and future directions. *Telesystems Conference, 1992. NTC-92., National*, pages 13/15–13/23, May 1992.

[MM99] Joseph Mitola and Gerald Q. Maguire. Cognitive Radio : Making Software Radios More Personal. *IEEE Personal Communications*, pages 13–18, August 1999.

[MSA06] P. Murphy, A. Sabharwal, and B. Aazhang. Design of WARP : A Flexible Wireless Open-Access Research Platform. In *Proceedings of EUSIPCO*, 2006.

[NKM+09] Dominique Nussbaum, Karim Kalfallah, Christophe Moy, Amor Nafkha, Pierre Lerary, Julien Delorme, Jacques Palicot, Jerome Martin, Fabien Clermidy, Bertrand Mercier, and Renaud Pacalet. Open Platform for Prototyping of Advanced Software Defined Radio and Cognitive Radio Techniques. In *Digital Systems Design, Euromicro Symposium on*, volume 0, pages 435–440, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[NVI10] NVIDIA. OpenCL Programming Guide for the CUDA Architecture, August 2010.

[NVI11] NVIDIA. NVIDIA CUDA C Programming Guide Version 4.0, June 2011.

[OMG08] OMG. Common Object Request Broker Architecture (CORBA/IIOP), April 2008.

[ope10] The OpenCL Specification, Version 1.1, September 2010.

[ora] Oracle. `http://cordis.europa.eu/fetch?CALLER=PROJ_ICT&ACTION=D&CAT=PROJ&RCN=80719` Visité le 11/07/2011.

[Pra10] Ramjee Prasad. *My personal Adaptive Global NET (MAGNET)*. Springer, 2010.

[PRR+03] Andreas Polydoros, Jukka Rautio, Giuseppe Razzano, Hanna Bogucka, Diego Ragazzi, Panos I. Dallas, Aarne Mämmelä, Michael Benedix, Manuel Lobeira, and Luigi Agarossi. WIND-FLEX : Developing a Novel Testbed for Exploring Flexible Radio Concepts in an Indoor Environment. *IEEE Communications Magazine*, July 2003.

[PZB+11] William Plishker, George F. Zaki, Shuvra S. Bhattacharyya, Charles Clancy, and John Kuykendall. Applying Graphics Processor Acceleration in a Software Defined Radio Prototyping Environment. In *Proceedings of the International Symposium on Rapid System Prototyping, Karlsruhe, Germany*, May 2011.

[qos] Qosmos. `http://www.ict-qosmos.eu/` Visité le 11/07/2011.

[Ram07] Ulrich Ramacher. Software-Defined Radio Prospects for Multistandard Mobile Phones. *Computer*, pages 62–67, October 2007.

[RGMF05] Xavier Revés, Antoni Gelonch, Vuk Marojevic, and Ramon Ferrús. Software Radios : Unifying the Reconfiguration Process over Heterogeneous Platforms. *EURASIP Journal on Applied Signal Processing*, 2005(16) :2626–2640, 2005.

[sca01] Software Communications Architecture Specifications - MSRC-5000SCA - V2.2, 2001.

[SCA11] Software Communications Architecture Specifications - Version Next(draft) 1.0.0.2, 2011.

[SH09] Piotr Szegvari and Christian Hentschel. Scalable Software Defined FM-Radio Receiver Running on Desktop Computers. In *The 13th IEEE Internation Symposium on Consumer Electronics*, 2009.

[SMM05] Jeff Smith, David Murotake, and Antonio Martin. Software communication architecture : Evolution and status update. *Military Embedded Systems*, pages 28–31, October 2005.

[Tec] Virginia Tech. OSSIE - SCA-Based Open Source Software Defined Radio. `http://ossie.wireless.vt.edu/`.

[TR08] Yahia Tachwali and Hazem Refai. Implementation of a BPSK Transceiver on Hybrid Software Defined Radio Platforms. In *3rd International Conference on Information and Communication Technologies From Theory to Applications*. IEEE, 2008.

[Tut02] Walter Tuttlebee. *Software Defined Radio - Enabling Technologies*. Wiley, 2002.

[WCW+09] Ying Wang, Wei-Nan Chen, Xiao-Wei Wang, Hon-Jun You, and Cheng-Lian Peng. The Hardware Thread Interface Design and Adaptation on Dynamically Reconfigurable SoC. In *International Conference on Embedded Software and Systems*, pages 173–178, 2009.

[WEL11] Di Wu, Johan Eilert, and Dake Liu. Implementation of a High-Speed MIMO Soft-Output Symbol Detector for Software Defined Radio. *Springer Journal of Signal Processing Systems*, 63 :27–37, January 2011.

[WSC10] Michael Wu, Yang Sun, and Joseph R. Cavallaro. Implementation of a 3GPP LTE turbo decoder accelerator on GPU. In *IEEE Workshop on Signal Processing Sytems (SIPS)*, pages 192–197, October 2010.

[WSGC11] Michael Wu, Yang Sun, Siddharth Gupta, and Joseph R. Cavallaro. Implementation of a High Throughput Soft MIMO Detector on GPU. *Springer Journal of Signal Processing Systems*, 64 :123–136, 2011.

[Xil10] Xilinx. Partial Reconfiguration User Guide, May 2010.

Résumé :

L'utilisation de la radio flexible permet d'envisager des réseaux sans fil évolutifs, plus efficaces et plus intelligents. Le terme « radio flexible » est un terme très général, qui peut s'appliquer à une implantation logicielle des opérations, à une implantation matérielle adaptable basée sur des accélérateurs matériels, ou encore à des implantations mixtes. Il regroupe en fait toutes les implantations de terminaux radio qui ne sont pas figées.

Les travaux réalisés durant cette thèse tournent autour de deux points. Le premier est l'utilisation des processeurs graphiques pour la radio flexible, et plus particulièrement pour la radio logicielle. Ces cibles offrent des performances impressionnantes en termes de débit brut de calcul, en se basant sur architecture massivement parallèle. Le parallélisme de données utilisé dans ces processeurs diffère cependant du parallélisme de tâches souvent exploitées dans les applications de radio logicielle. Différentes approches pour résoudre ce problème sont étudiées. Les résultats obtenus sur ce point permettent une nette amélioration du débit de calcul atteignable avec une implantation logicielle, tout en libérant le processeur pour d'autres tâches.

Le deuxième point abordé dans cette étude concerne la définition d'un environnement permettant de supporter différentes possibilités d'implantation de la radio flexible. Cet environnement englobe le support de la plateforme hétérogène, et la gestion des applications sur ces plateformes. Bien qu'encore expérimental, les premiers résultats obtenus avec l'environnement montrent ses capacités d'adaptation, et le rendent utilisable pour des applications radio variées sur des plateformes hétérogènes.

Abstract :

The development of flexible radio may lead to evolving wireless networks. This characteristic enables more efficient and smarter networks. Flexible radio is not a precise definition. It can be used to describe a software implementation of radio operations, as well as a hardware implementation based on configurable hardware coprocessors. It can also refer to heterogeneous implementations. It describes any implementation which can evolve.

During this PhD, we focused on two different aspects of flexible radio. First, the use of graphical processors for flexible radio (and especially software radio) is studied. These execution targets enable impressive performances, when studying raw attainable processing throughput, through the use of massively parallel architectures. The problem is that the data parallelism exhibited by these processors does not match the task parallelism of software radio applications. Different approaches to correct this mismatch are studied in this work. The displayed results show an improvement in the attainable software implementation, while letting the processor process other tasks.

The other issue addressed in this work is the definition of an environment able to support different implementation choices for flexible radio. Support for multiple implementations includes heterogeneous platforms support, as well as application management on these platforms. While this environment is still in early development stage, preliminary results demonstrate its adaptability, and eases development of applications on different heterogeneous platforms.

ISBN : 978-2-84813-180-1