UNIVERSITE PARIS-SUD

ECOLE DOCTORALE : Sciences et Technologies de l'Information des Télécommunications et des Systèmes

DISCIPLINE : PHYSIQUE

THESE DE DOCTORAT

Soutenue le 20/05/2014

présentée par

Haixiong YE

Impact des transformations algorithmiques sur la synthèse de haut niveau: application au traitement du signal et des images

Examinateur	Corinne Ancourt	Chercheur, ENSMP/CRI, Fontainebleau
Examinateur	Daniel Etiemble	Professeur, LRI, Université Paris-Sud
Examinateur	Alain Mérigot	Professeur, LRI, Université Paris-Sud
encadrant ST	Olivier Florent	Ingénieur-docteur, ST Microelectronics
Directeur de thèse	Lionel Lacassagne	Maitre de Conférences HDR, LRI, Université Paris-Sud
Rapporteur	Antoine Manzanera	Enseignant-chercheur HDR, ENSTA Paristech
Rapporteur	Daniel Ménard	Professeur, IETR, Université de Rennes 1

TABLE DES MATIÈRES

Table des matières **3**

Remerciements **i**

Introduction **iii**
Les contextes de cette thèse CIFRE . . . iii
Les besoins des applications enfouies . . . iii
Le *time to market* . . . iv
Le contexte professionnel de cette thèse CIFRE : ST Micro-electronics . . . iv
La synthèse automatique de haut niveau ou HLS (*High Level Synthesis*) . . . v
Les transformations algorithmiques de haut niveau ou HLT (*High Level Transforms*) vi
Objectifs de la thèse . . . vii
Contribution de la thèse . . . vii
Plan de cette thèse . . . viii

1 Synthèse automatique de haut niveau et cibles architecturales **1**
1.1 La synthèse automatique de haut niveau . . . 1
1.1.1 Historique et outils . . . 1
1.1.2 Catapult-C . . . 3
1.1.3 Optimisation logicielle ou matérielle ? . . . 6
1.2 Méthodologie d'évaluation de performance . . . 6
1.3 Cibles architecturales . . . 6
1.3.1 Processeur ST XP 70 . . . 7
1.3.2 Processeur ARM Cortex-A9 . . . 8
1.3.3 Processeur Intel Penryn ULV . . . 9
1.3.4 Autres processeurs ? . . . 10
1.4 Conclusion du chapitre . . . 10

2 Métaprogrammation **11**
2.1 Principes de la métaprogrammation . . . 11
2.1.1 L'évaluation partielle . . . 12
2.1.2 Outils pour la métaprogrammation . . . 12
2.2 Fonctionnement du préprocesseur . . . 13
2.2.1 Eléments de bases du préprocesseur . . . 14
2.2.2 Fonction de base . . . 15
2.3 Application au design d'interface modulaire . . . 16
2.3.1 Implémentation . . . 16
2.4 Conclusion du chapitre . . . 22

3 Filtrage non récursif **23**

3.1 Optimisation de l'implantation d'un filtre FIR3 . . . 23
3.1.1 Optimisations logicielles . . . 24
3.1.2 Optimisations matérielles . . . 25
3.1.3 Résultats et analyse . . . 26
3.2 Optimisation de l'implantation de deux filtres FIR en cascade . . . 30
3.2.1 Implantation . . . 30
3.2.2 Résultats et analyse . . . 32
3.3 Evaluation de l'impact des transformations pour les processeurs généralistes . . . 34
3.3.1 Résultats et analyse . . . 36
3.3.2 ASIC vs GPP : comparaison en temps et en énergie . . . 40
3.4 Conclusion du chapitre . . . 41

4 Filtrage récursif 43
4.1 Introduction . . . 43
4.2 Transformations algorithmiques : les différentes formes du filtre de FGL . . . 44
4.2.1 Stabilité numérique du filtre . . . 46
4.3 Benchmarks . . . 48
4.3.1 Benchmarks des trois formes sans optimisation . . . 48
4.3.2 Benchmarks des trois formes : optimisations logicielles ou matérielles . . . 50
4.4 Optimisation de l'implantation de deux filtres IIR en cascade . . . 56
4.4.1 Implantations . . . 56
4.5 Portage sur processeurs généralistes . . . 60
4.5.1 Comparaison en *cpp* . . . 60
4.5.2 Comparaison en temps et en énergie . . . 61
4.6 Conclusion du chapitre . . . 62

5 Détection de mouvement 63
5.1 Introduction . . . 63
5.1.1 Filtrage Sigma-Delta . . . 63
5.1.2 Post-traitement morphologique . . . 66
5.2 Optimisations et transformations de haut niveau . . . 69
5.2.1 Opérateur morphologique . . . 69
5.2.2 Optimisations logicielles . . . 72
5.2.3 Optimisations matérielles . . . 74
5.2.4 Résultats et analyse . . . 74
5.3 Evaluation de l'impact des transformations pour les processeurs généralistes . . . 77
5.3.1 Résultats et analyse . . . 77
5.4 Conclusion du chapitre : *higher, faster and greener* . . . 79

Conclusion et perspectives de recherche 81

Annexe FIR 83

Annexe IIR 85

Annexe Sigma-Delta 95

Annexe Erosion 107

Bibliographie 119

REMERCIEMENTS

C'est dans le cadre d'un contrat CIFRE entre les Laboratoires IEF et LRI de l'Université Paris-Sud et la division de HED de ST Microelectronics à Grenoble que s'inscrit ma thèse. Elle s'est déroulée au sein de l'équipe CAD support de ST en collaboration avec le département AXIS de IEF puis l'équipe Archi du LRI.

Je tiens à adresser mes remerciements à l'entreprise STMicroelectronics pour avoir permis la réalisation de mes travaux dans les meilleures conditions.

Je tiens à exprimer ma profonde gratitude à Monsieur Lionel Lacassagne, directeur de la thèse au LRI qui a suivi au jour le jour ce travail de thèse. Merci Lionel pour ta disponibilité, ta gentillesse, tes conseils avisés, la patience avec laquelle tu ma expliqué les différentes transformations algorithmiques pour la HLS et le fonctionnement des processeurs généralistes SIMD.

Je tiens à remercier très vivement Monsieur Olivier Florent, manager de l'équipe CAD support et encadrant de ma thèse, pour son accueil, la confiance qu'il m'a accordé dès mon arrivée dans l'entreprise, son soutien constant, ses précieux conseils, sa responsabilité et le temps qu'il m'a consacré tout au long de cette période, sachant répondre à toutes mes questions.

Je tiens à remercier Monsieur Joel Falcou, co-encadrant de la thèse, au LRI pour la confiance qu'il m'a accordé, son encadrement et ses précieux conseils en méta-programmation.

Je tiens à remercier Messieurs Bernard Caubet, Laurent Chalet, Guy Chambonnières, Lionel Picandet, Cyril Roux, Cyril Vartanian, membres de l'équipe IDT, qui m'ont aidé au cours de ma thèse.

Je tiens à remercier les membres du jury pour avoir accepté de juger mon travail : Messieurs Daniel Ménard et Antoine Manzanera, en qualité de rapporteurs pour leurs remarques pertinentes sur le manuscrit et les corrections apportées. Madame Corinne Ancourt en tant qu'examinatrice et Messieurs Daniel Etiemble et Alain Mérigot en tant qu'examinateurs.

INTRODUCTION

Les contextes de cette thèse CIFRE

Les besoins des applications enfouies

Si les performances des ordinateurs et des systèmes enfouis et embarqués progressent de manière exponentielle depuis des dizaines d'années en s'appuyant sur les progrès de la technologie VLSI qu'exprime la loi de Moore, les performances qu'exigent un grand nombre d'applications restent très supérieures aux possibilités actuelles des processeurs d'usage général (*General Purpose Processor* ou GPP), ou des processeurs spécialisés existant aujourd'hui. Des domaines d'application très importants, comme les applications audio, les applications vidéo, les applications graphiques, les applications de communication ou de reconnaissance ont des besoins considérables en puissance de calcul, qui vont de dizaine de millions d'opérations par seconde à la centaine de milliards d'opérations par seconde, comme le montre la figure 1.

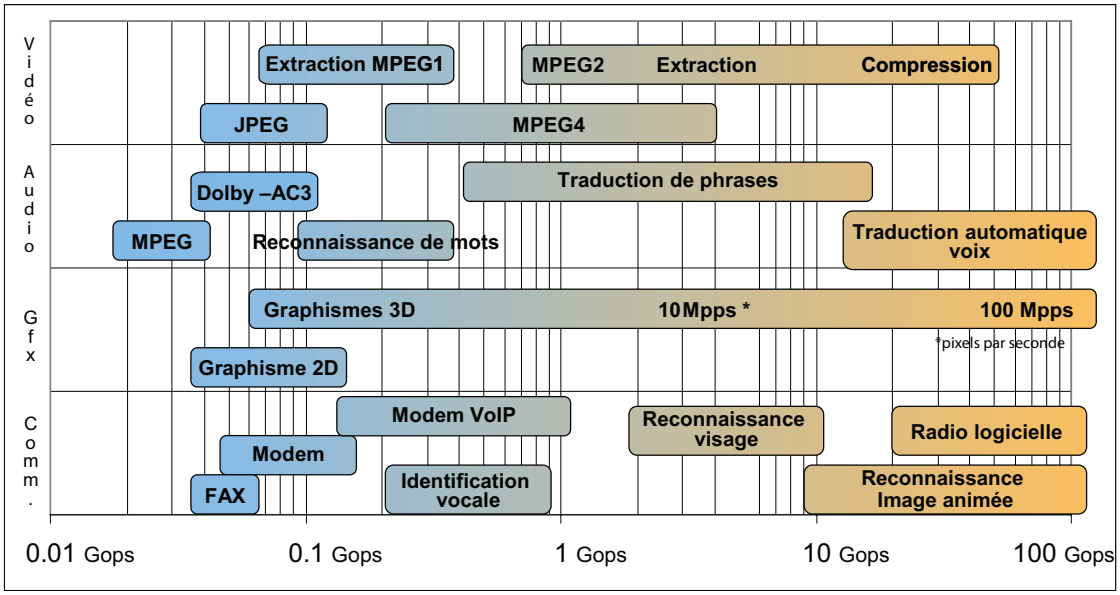

FIGURE 1 – besoin des applications en puissance de calcul, d'après ITRS design ITWG Juillet 2003

La table 1 présente la *roadmap* des applications enfouies pour les technologies futures. Par rapport à la figure 1 qui présente les besoins en puissance de calcul, elle ajoute un certain

nombre de caractéristiques associées aux différentes générations de technologie, comme la puissance dissipée (crête et au repos), les capacités des batteries, les vitesses de communications et l'efficacité énergétique. On peut constater qu'alors que les puissances dissipées doivent rester pratiquement constantes, la puissance de calcul doit passer de 2 GOPS (technologie 90 nm) à 77 GOPS (technologie 45 nm), soit un gain d'environ $\times 40$. On remarque par ailleurs qu'à partir de la technologie 90 nm, le facteur *parallélisme* intervient, qui est l'équivalent pour les systèmes enfouis de l'irruption des multi-cœurs dans les microprocesseurs des PC.

Technologie	130 nm	90 nm	65 nm	45 nm	32 nm	22 nm
année	2002	2004	2006	2008	2010	2012
nombre de portes/mm^2	80k	150k	300K	600k	1.2M	2.4M
Tension d'alimentation	1.2 V	1 V	0.8 V	0.6 V	0.5 V	0.4 V
Fréquence d'horloge	150 Mhz	300 Mhz	450 Mhz	600 Mhz	900 Mhz	1.2 Ghz
Application (Performance max requise)	Traitement d'images	Codec vidéo temps réel MPEG4/CIF		Interprétation temps réel		
Application (Autres)	Browser Web Courrier électronique Ordonnanceur	TV téléphone (1 :1) Reconnaissance voix (entrée)		TV téléphone (>3 :1) Reconnaissance voix (opération)		
	Authentification (moteur cryptographie)					
Puissance calcul (GOPS)	0.3	2	14	77	461	2458
degré de Parallélisme	1	4	4	4	4	4
Vitesse communication (Kbps)	64	384	2304	13824	82944	497664
Efficacité énergétique (MOPS/mW)	3	20	140	770	4160	24580
Puissance crête (mW)	100	100	100	100	100	100
Puissance au repos (mW)	2	2	2	2	2	2
Capacité batterie (Wh/kg)	120	200		400		

TABLE 1 – La *Roadmap* du traitement enfoui pour les technologies futures

Le *time to market*

Le *time to market* ou délai de mise sur le marché est une expression anglo-saxonne utilisée pour exprimer le délai nécessaire pour le développement et la mise au point d'un projet ou d'un produit, avant qu'il puisse être lancé sur le marché. Avec le raccourcissement des cycles de vie des produits, il est devenu un facteur stratégique majeur, en ce sens où une réduction caractéristique du *time to market* peut permettre à une entreprise d'améliorer de manière significative sa rentabilité mais aussi lui donner la possibilité de prendre un avantage concurrentiel décisif.

Comme le montre la table 1, d'une part, la technologie de gravure change environ tous les deux ans. Et d'autre part des applications – toujours plus complexes – doivent être développées, validées et implantées sur de nouvelles architectures, dans ce même laps de temps de deux ans. Pour assurer un *time to market* constant (voire le faire diminuer), il est donc nécessaire d'avoir des équipes de développement de plus en plus grosses – ce qui est difficilement viable économiquement – soit de réduire les temps de développement en changeant la manière de développer le logiciel et/ou le matériel.

Le contexte professionnel de cette thèse CIFRE : ST Micro-electronics

Cette thèse CIFRE a été réalisée dans la division HED (*Home and Entertainment Division*) de STMicro-electronics à Grenoble. Cette division est en charge du développement des ASICs (*Application Specific Integrated Circuit*) intégrant pour des codecs audio-vidéo MPEG et H264 qui seront par la suite intégrés dans des set-top boxes, mais aussi des smartphones ou des tablettes.

La synthèse automatique de haut niveau ou HLS (*High Level Synthesis*)

Le but de cette thèse est de proposer une méthodologie et des outils permettant d'accélérer l'implémentation matérielle d'algorithmes complexes de traitement de flux vidéo. Ces algorithmes sont habituellement décrits en langage dit de *haut niveau* (comme le C ou le C++) et validés au travers d'une chaine de simulation contenant l'algorithme lui-même ainsi que d'autres composants.

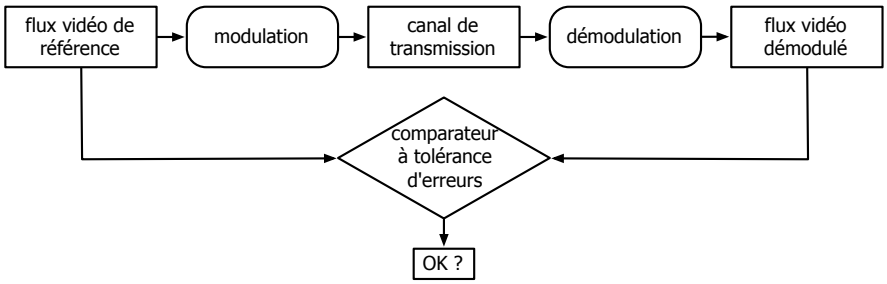

FIGURE 2 – environnement de simulation d'un démodulateur de flux vidéo

La figure 2 représente par exemple un environnement complet en C++ destiné à valider des algorithmes de démodulation de flux vidéo transmis par une antenne, en vue de les implémenter dans un circuit. L'utilisation d'un langage de haut niveau est ici indispensable pour avoir des performances de simulation acceptable. Il permet ici de simuler 1.5 seconde de transmission réelle en 4 heures, sachant qu'il faut probablement simuler plusieurs heures de transmission pour valider correctement l'algorithme.

Une fois l'algorithme validé, une étude micro architecturale doit permettre éventuellement de déterminer quelles sont les parties destinées à être simplement exécutées sur un processeur embarqué (SW). Les parties devant être implémentées en matériel (HW) doivent ensuite être traduites dans un langage de description adapté (RTL).

Pour arriver à ce résultat, plusieurs étapes peuvent être nécessaires comme le montre la figure 3 : séparation entre la logique de calcul et la logique de contrôle, passage des calculs d'une arithmétique flottante à une arithmétique bornée, définition des interfaces du composant matériel, séquencement des calculs, allocation des ressources, et enfin écriture du RTL, tout en prenant en compte les contraintes de surface, de performance et de consommation visées. Enfin la fonctionnalité du code RTL obtenu doit être vérifiée par rapport à l'algorithme original, soit par preuve formelle, soit en réutilisant l'environnement de simulation original.

Il existe aujourd'hui des solutions logicielles, appelés outils de synthèse de haut niveau (HLS), qui permettent d'exécuter plus ou moins automatiquement certaines de ces étapes, et sous certaines conditions. Ces solutions sont aujourd'hui assez efficaces lorsqu'elles s'appliquent aux parties de calcul de données une fois le partitionnement effectué, sur des algorithmes déjà exprimé en algorithmique fixe et dont les interfaces sont normalisées. Cela ne

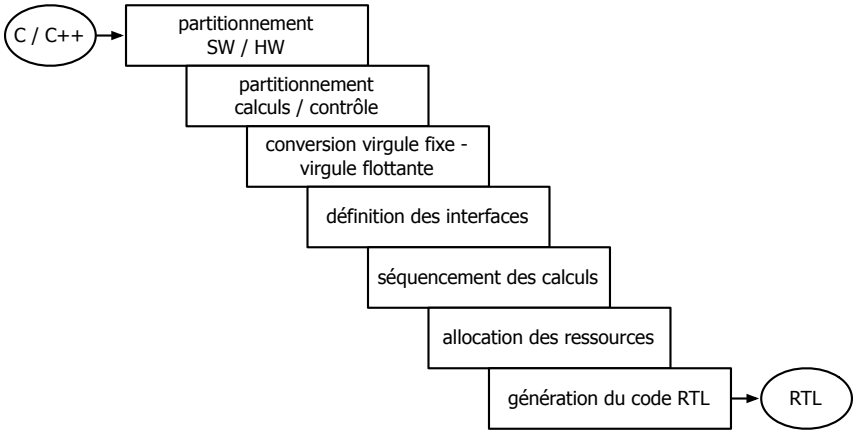

FIGURE 3 – étapes pour passer d'un code C++ algorithmique à une description RTL

couvre donc qu'une partie des étapes décrites en figure 3.

A noter que ces outils de synthèse sont, à l'instar des compilateurs classiques, capables de réaliser de plus en plus d'optimisations logicielles comme le déroulage de boucle, et le pipeline logiciel, qui dans le cas de la synthèse d'architecture dataflow a un impact très important.

Les transformations algorithmiques de haut niveau ou HLT (*High Level Transforms*)

Si les compilateurs *optimisants* continuent de progresser, notamment dans leurs capacités à paralléliser (pour les processeurs multi-cœurs) et à vectoriser (pour les processeurs avec extensions multimédia SIMD), certaines transformations restent hors de leur portée. Cela peut être dû à un manque de sémantique et alors cela peut être résolu par des extensions du langage. Cette approche est actuellement très active. Mais cela peut aussi être dû à l'obligation qu'a le compilateur de respecter la sémantique du programme.

Ces transformations peuvent être classées en trois groupes, de complexité croissante.

Au niveau le plus simple, cela concerne la ré-écriture d'expressions algébriques. Ces expressions algébriques – entières ou flottantes – peuvent par exemple être factorisées pour obtenir le résultat plus rapidement. Le problème est que ces transformations algébriques peuvent influer sur la précision des calculs et sur leur reproductibilité car les calculs sont faits avec une précision finie. Les additions sont source d'*absorption* et les soustractions, sources de *cancellation*. Ces ré-écritures peuvent être autorisées via des options de compilation dédiées (comme `-fast-math`).

Au second niveau, cela concerne des transformations algébriques liées à un domaine applicatif (typiquement signal ou image) qui se baseront sur la connaissance de propriétés mathématiques pour simplifier certains opérateurs. Par exemple la décomposition de convolution 2D en deux convolutions 1D. Le projet Spiral (`www.spiral.net`) permet de générer directement un code optimisé pour une FFT particulière, code qui sera bien plus rapide qu'un code généraliste.

Enfin, le troisième niveau concerne des transformations qui nécessitent l'allocation mémoire de tableaux supplémentaires pour des calculs intermédiaires. Un cas très courant est le calcul d'image intégrale permettant de calculer la somme de n'importe quel ensemble de

point d'un domaine carré avec une complexité constante de 4 opérations.

Ces transformations sont relativement simples à expliquer à tout développeur chevronné (*hacker*) qui pourra par la suite les implémenter dans de nombreux cas.

Objectifs de la thèse

Le but de cette thèse est de définir une méthodologie et des outils pour améliorer les capacités et les performances de synthèse d'outils de synthèse automatique. L'idée est de se baser sur les capacités de méta-programmation du C++ et d'utiliser une description unique d'une chaîne algorithmique pour réaliser l'analyse et l'optimisation de cette chaîne via des transformations de haut niveau.

Concernant la partie analyse, il s'agit de concevoir et valider les algorithmes sur une station de travail classique dotée d'un compilateur C++ et que ces codes servent à la fois pour la validation et la synthèse architecturale.

Concernant la partie optimisation et synthèse, il s'agit de :

- proposer une méthode d'écriture du C++ permettant une meilleure séparation (explicite ou implicite) des parties calcul et contrôle sans sacrifier à la souplesse d'écriture, aux performances de la simulation ou à la qualité du matériel synthétisé,

- définir les interfaces d'entrée sorties des éléments algorithmes de calcul (fonction C) d'une façon compatible avec les outils de synthèse de haut niveau et de mixer descriptions logicielle et matérielle (permettant de remplacer dans la chaîne de simulation tout ou une partie de l'algorithme par une description plus proche du matériel),

- réaliser des transformations algorithmiques de haut niveau (HLT) sur le graphe associé à la description de la chaîne de traitement. Ces transformations d'abord faites manuellement devront par la suite être encapsulées dans un système de méta-programmation par macros.

La démarche sera validée sur un ensemble représentatif d'algorithmes du traitement du signal et des images.

Contribution de la thèse

La contribution de cette thèse est la proposition d'une méthodologie de synthèse architecturale fortement automatisée grâce à de la méta-programmation par macros. L'outil de métaprogrammation développé condense l'expertise d'ingénieurs et de chercheurs dans les domaines de la compilation (pour les optimisations logicielles), de l'adéquation algorithme-architecture (pour les transformations algorithmiques de haut niveau) et de l'électronique (pour les optimisations matérielles).

Le but est que l'utilisateur final puisse – sans être un expert des domaines précédemment cités – activer des optimisations logicielles et matérielles (combinées à des transformations algorithmiques afin d'amplifier leur impact) pour un opérateur donné et ainsi réaliser simplement et automatiquement une exploration de l'espace des configurations des paramètres de l'algorithme (et de son instanciation matérielle) pour minimiser des contraintes (surface ou énergie),

ou pour trouver des compromis surface/énergie en un temps très court (*time to market*).

Le code C est unique et sert à la fois à la validation fonctionnelle de l'algorithme sur un PC et à la synthèse de l'architecture ASIC. Mais grâce aux macros développées, il sert aussi à générer du code C scalaire ou SIMD pour faire des benchmarks comparatifs.

Là encore, l'utilisateur n'a pas besoin d'être un expert en codage SIMD (certains *design patterns* sont à connaitre pour implanter efficacement les transformations algorithmiques de haut niveau) ni de maitriser les jeux d'instructions SSE, VECx et Neon, qui sont les extensions SIMD pour lesquel l'outil est capable de générer un code optimisé.

Plan de cette thèse

Le fil conducteur de cette thèse va être l'augmentation croissante de la complexité des algorithmes utilisés.

Le chapitre 1 fait un rapide état de l'art des outils de synthèse automatique et présente en détail Catapult-C qui est utilisé pour l'ensemble des synthèses réalisées. Il se termine par un descriptifs des processeurs généralistes qui seront utilisés pour comparer l'impact des transformations algorithmes sur ces cibles.

Le chapitre 2 présente les techniques de méta-programmation en se restreignant à la méta-programmation par macro qui permet de s'interfacer avec le compilateur C++ de Catapult-C.

Le chapitre 3 présente l'optimisation de la synthèse de filtres non récursifs, ainsi que – dans une seconde partie - l'optimisation de leur utilisation en cascade et se termine par une comparaison avec des processeurs généralistes.

Le chapitre 4 fait de même pour les filtres récursifs. Les problèmes de stabilité de ces filtres sont présentés, et la difficulté d'optimisation pour un ASIC et pour un processeur généraliste est étudiée.

Enfin le chapitre 5 présente une application de détection de mouvement avec post-traitement morphologique qui permet d'aller plus loin dans l'utilisation des HLT.

CHAPITRE 1

SYNTHÈSE AUTOMATIQUE DE HAUT NIVEAU ET CIBLES ARCHITECTURALES

Ce chapitre est composé de deux parties. La première partie présente la synthèse automatique de haut niveau comme une alternative aux approches classiques de conception d'architectures matérielles pour une utilisation en production où les délais de conception et réalisation (*time to market*) nécessitent de concevoir rapidement des circuits où les contraintes antagonistes de surface et de consommation imposent de trouver rapidement des compromis entre l'efficacité énergétique et le coût de production du circuit (proportionnel à sa surface). La seconde partie présente trois processeurs généralistes qui serviront à positionner les performances de l'ASIC et qui permettront aussi d'évaluer l'impact des transformations de haut niveau sur les deux métriques de performance que sont la surface et la consommation énergétique.

1.1 La synthèse automatique de haut niveau

1.1.1 Historique et outils

Les deux langages *historiques* pour la description d'architectures matérielles (*Hardware Description Language*) sont VHDL (*VHSIC HDL*) et Verilog. Le langage VHDL est issu de l'initiative VHSIC (*Very High Speed Integrated Circuit*) du Département de la Défense américain. Il a une syntaxe proche d'Ada, un autre langage de programmation initié aussi par le DoD. Verilog, de son nom complet Verilog-HDL était à l'origine un langage propriétaire développé par la société Cadence et a une syntaxe proche du C. Ces langages permettent une plus grande abstraction que le RTL (*Register Transfer Level*).

Une étape importante vers l'utilisation de langages de haut niveau a été le développement de SystemC, ensemble de bibliothèques d'objets C++ (bus, ALU, mémoire, processeurs, ...) permettant de concevoir des architecture et de simuler rapidement les différentes fonctions implantées. Si la simulation fonctionnelle a pu être simplifiée, la synthèse d'architecture n'est pas systématiquement possible, car certains composants ne sont pas synthétisables.

La synthèse (automatique) de haut niveau ou HLS (*High Level Synthesis*) est une nouvelle approche dans la conception de circuits VLSI. Elle vise à diminuer le nombre d'étapes de

transition/adaptation qui existent entre la description d'un algorithme dans un langage de haut niveau et la description matérielle de l'architecture réalisant cet algorithme. Elle est aussi appelé *synthèse C* car le langage utilisé en entrée de l'outil est le langage C ou un sous ensemble de celui ci. Certains outils acceptent aussi du C++. C'est un domaine recherche très actif [69] [10], [11], [15], [16], [47], [42], [30].

Les principaux avantage de la synthèse HLS sont :

- la rapidité de conception/prototypage. Les algorithmes sont plus faciles et plus rapides à écrire dans des langage de haut-niveau. Le délai du développement d'une nouvelle *IP* (*Intellectual Property*) [1] grâce à la synthèse HLS est 2 à 3 fois plus rapide qu'avec une description manuelle en RTL. Les IP sont conçues pour être ré-utilisables.

- La co-simulation : les concepteurs peuvent faire une co-simulation logicielle/matérielle en ajoutant une couche de *wrappers* pour réaliser une simulation fonctionnelle en SystemC.

- La durée des simulations : le délai de simulation et de vérification du système HLS est au moins 10 fois plus rapide que celui en RTL. La durée de simulation de systèmes complexes au niveau RTL devient même problématique. Enfin la synthèse HLS permet d'obtenir rapidement des estimations des caractéristiques du circuit (surface, puissance consommée, énergie consommée).

Il y a de plus en plus d'outils disponibles, aussi bien des outils commerciaux que des outils issus du monde de la recherche. Parmi les outils commerciaux, nous pouvons citer : Catapult-C (Calypto) [8], Pico(Synfora) [54], Cynthesizer(Forte Design system) [61], Cascade(critical Blue) [5], Synphony-C-Compiler [60], C-to-Silicon [59], Bleupec [6]. A noter qu'il existe aussi des approches Matlab-VHDL.

Parmi les outils académiques, nous pouvons citer : Gaut [14, 55] , GraphLab [41], Spark [29],Compann/Laura [58], ESPAM [52], MMAlpha [51], Paro [4], UGH [3], Streamroller [35], xPilot [12], Array-OL.

Les outils Compaan, Paro et MMAlpha sont focalisés sur la compilation efficace de boucles, et utilisent le modèle polyédrique pour faire les analyses et les transformations. A noter aussi que Array-OL, initialement développé pour des algorithmes réguliers, manipulant des tableaux – d'où son nom (*Array Oriented Language*) a évolué ces dernières années pour être capable de déployer des applications sur des architectures matérielles distribuées et hétérogènes [7] et pour générer du VHDL [70].

Nous allons maintenant regarder en détail Catapult-C qui est utilisé en production dans la division HED de STMicroelectronics.

1. au sens composant logiciel à forte valeur ajoutée dont la propriété intellectuelle est protégée, même s'il existe des IP libres de droits

1.1.2 Catapult-C

La simulation système en SystemC souffre d'un défaut majeur : certains composants ne sont pas synthétisables. Il faut alors de nouveau passer du temps à décrire l'architecture en VHDL ou en Verilog, ce qui va à l'encontre de l'un des objectifs initiaux : réduire le temps de conception. Ce problème existe aussi avec la synthèse HLS car certaines constructions en C ne sont pas synthétisables (souvent lié à l'utilisation de pointeurs, comme l'*aliasing* de pointeurs et les pointeurs de fonctions). Pour régler ce problème, seul un sous-ensemble du C est alors autorisé . Afin de guider le développeur, Catapult-C propose donc un ensemble de classes dites algorithmiques.

1.1.2.1 Les classes de Catapult-C

Il existe deux classes de nombres pour les calculs en entier et les calculs en virgule fixe :
- `ac_int<W, false>` pour les entiers non signés codés sur W bits et `ac_int<W, true>` pour les entiers signés.
- `ac_fixed<W, I, sign>` pour les entiers en virgule fixe codés sur W bits avec I bits pour la partie fractionnaire, c'est à dire $Q_{W.I}$.

Il existe aussi une classe template de nombres complexes `ac_complex<T>` pouvant être paramétrée avec des types C (`int`, `float`) ou des types de Catapult-C (`ac_int`, `ac_fixed`). A l'heure actuelle il n'y a pas de type flottant paramétrique (`ac_float(W)`). Cette classe permettrait d'intégrer l'étape de conversion de calculs en virgule flottante → calculs en virgule fixe dans le flux de conception.

Si les tableaux de données peuvent être manipulés avec des boucles classiques, il existe en plus une classe permettant d'implémenter des FIFO, c'est la classe `ac_channel<T>` qui propose deux méthodes assurant l'ordre de lecture et d'écriture dans les flux : `read` et `write`.

1.1.2.2 Les optimisations logicielles

A l'instar d'un compilateur classique optimisant, le compilateur matériel de Catapult-C doit être capable de réaliser efficacement les optimisations logicielles les plus importantes.

Les optimisations peuvent être classifiées en trois catégories :
- les transformations de haut niveau ou HLT (*High Level Transforms*,
- les optimisations logicielles,
- les optimisations matérielles.

Les transformations de haut niveau, aussi appelées transformations algorithmiques, sont des optimisations spécifiques à un domaine, comme le traitement du signal ou le traitement d'images. Les exemples les plus courants sont la décomposition de filtres séparables et la factorisation des calculs, ce qui permet à la fois de diminuer la complexité de calcul et le nombre d'accès mémoire. Ces différentes transformations seront présentées au fur et à mesure dans chaque chapitre applicatif.

Les principales optimisations logicielles [2] sont le déroulage de boucle, la rotation de registres et le pipeline logiciel. Ces optimisations sont disponibles dans Catapult-C. Concernant la rotation de registres, Catapult-C est d'une part capable de générer du code RTL réalisant une rotation de registres, mais aussi de comprendre sémantiquement qu'un bout de code C réalise une rotation de registres (entre registres ou entre cases mémoire d'un tableau local). Dans ce cas, la rotation (la recopie case par case) qui été réalisée séquentiellement est réalisée en parallèle et

la boucle réalisant les itérations disparait : il n'y a pas création d'un automate à états pour faire la rotation, d'où un gain de surface. A noter que lorsque l'espace d'itération d'une boucle est connu à la compilation, Catapult-C peut totalement dérouler cette boucle (*loop-unwinding*). Ces différentes optimisations sont largement détaillées et mise en applications sur de nombreux cas standards dans le *Blue Book* de Catapult-C [24].

Les optimisations matérielles concernent principalement l'interconnexion avec différentes mémoires, plus ou moins rapides. Cette mémoire peut être simple port (SP) et permettre un accès par cycle, ou être double port et permettre deux accès par cycle. Nous utiliserons les mémoires simple port et double port fournies par ST en techno 65 nm.

Lorsque plus de deux accès mémoire par cycle seront nécessaires, une mémoire entrelacée par bancs, composée de bancs mémoire simple port sera utilisée. L'automate d'adressage mémoire sera écrit en C et intégré dans le flot de Catapult-C afin de prendre en compte son coût (surface, consommation) dans le total.

Les optimisations par défaut de Catapult-C sont l'optimisation en surface et l'optimisation en puissance. Restent donc à disposition de l'utilisateur les optimisations logicielles et matérielles.

L'utilisation typique de Catapult-C consiste à fournir un code C ou C++ et à spécifier une fréquence minimale de fonctionnement. Si par défaut, quasiment tous les paramètres peuvent être explorés automatiquement par Catapult-C – nous appellerons par la suite ce mode de fonctionnement, le mode *auto* – au moins un paramètre peut être contrôlé par l'utilisateur : l'intervalle de lancement (*initiation interval* ou ii) qui spécifie la latence en cycles entre le lancement de deux itérations d'un boucle. Dans l'hypothèse d'un fonctionnement régulier de type flot de données, toutes les itérations ont la même durée. Ainsi la différence de temps entre le début de deux itérations de boucle est égale à la différence de temps de fin de ces deux même boucles.

Nous verrons que l'intervalle de lancement a un impact sur tous les paramètres (surface, vitesse, consommation) du circuit généré et va de pair avec le pipeline logiciel (*software pipeline*) qui est l'optimisation clé pour la synthèse HLS.

1.1.2.3 L'*initiation interval*

On souhaite par exemple additionner quatre valeurs : $t = a+b+c+d$ (exemple issu de [24]) avec des additionneurs à deux entrées. Nous faisons en plus l'hypothèse qu'à la fréquence de synthèse voulue, chaque addition ne dure qu'un cycle. En fonction de l'intervalle de lancement (ii), nous obtenons les trois cas de la figure 1.1.

Dans le premier cas lorsque $ii = 3$, (Fig. 1.1 haut), un seul additionneur suffit pour réaliser séquentiellement une addition par cycle : il est donc ré-utilisé à chaque cycle et les données en entrée sont sélectionnées via un ensemble de multiplexeurs. Dans le second cas, lorsque $ii = 2$, (Fig. 1.1 millieu), deux additionneurs sont nécessaires : le premier pour additionner le résultat temporaire t_2 avec d de la première itération de boucle, le second pour additionner a et b de la seconde itération. Enfin dans le troisière cas, lorsque $ii = 1$, (Fig. 1.1 millieu), trois additionneurs sont nécessaires.

Ce fonctionnement est le pendant matériel du pipeline logiciel pour processeur VLIW comme le DSP C6x de Texas Instrument où quatre additionneurs et deux multiplieurs 32 bits - complètement pipelinés - permettent à chaque cycle de réaliser six opérations. A noter que sur les processeurs généralistes superscalaires, il est certes possible de faire du pipeline logiciel, mais

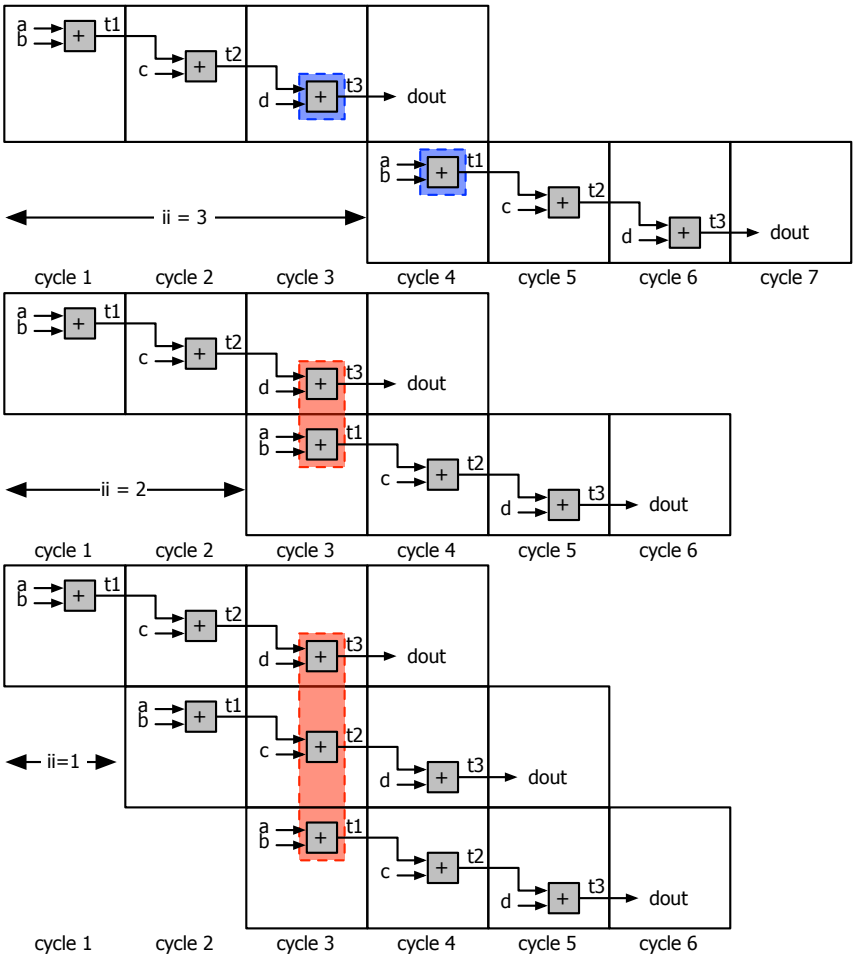

FIGURE 1.1 – Pipeline logiciel et *initiation interval*. En haut : ii=3 $\Rightarrow$ 1 seul *adder* est nécessaire. Au millieu : ii=2 $\Rightarrow$ au moins deux *adders* sont nécessaires. En bas : ii=1 $\Rightarrow$ trois *adders* sont nécessaires (version totalement pipelinée)

avec une écriture de cases mémoires (d'indices de tableaux) et non des registres. A la charge du compilateur de réaliser la mise en registres (*scalarisation*).

Dans la réalité, il en va tout autrement. Catapult-C réalisant des optimisations globales (avec entre autres, partage de ressources), il est tout à fait concevable que pour réaliser plusieurs fois une même opération (addition, multiplication), Catapult-C utilise des opérateurs de performances différentes (surface et vitesse). Cela peut se traduire – typiquement à haute fréquence – par des multiplieurs rapides (et de grande surface) sur le chemin critique et des multiplieurs "lents" (et petits) à d'autres endroits. Cela peut aussi se traduire, malgré une valeur basse de ii pour contraindre à utiliser plusieurs opérateurs, à la synthèse d'un unique opérateur très rapide réutilisé via un ensemble de multiplexeurs. Catapult-C réalisant des optimisations globales de l'ensemble du circuit, la synthèse d'un unique opérateur et l'utilisation de plusieurs multiplexeurs peut être plus efficace (en surface, en puissance) que la solution qui semble évidente.

1.1.2.4 Autres capacités de Catapult-C

Catapult-C permet de faire intervenir le temps dans l'exécution d'une boucle. Partant d'une

boucle produisant un résultat par itération, il est possible de contraindre la boucle pour produire deux résultats tous les deux cycles (le débit moyen est respecté), soit de produire deux résultats tous les cycles, ce qui correspond à un déroulage de boucle d'un facteur deux. Ces capacités ont été validées en amont de la thèse, mais n'ont pas été utilisées par la suite.

1.1.3 Optimisation logicielle ou matérielle ?

Cette thèse a pour but de fournir une évaluation de l'impact de transformations HLT pour la synthèse de haut niveau. On peut toutefois se poser la question de savoir quelle serait – s'il fallait choisir – le type d'optimisation le plus efficace : vaut-il mieux faire des optimisations logicielles ou des optimisations matérielles ? Vaut-il mieux faire du déroulage de boucle ou s'interfacer avec de la mémoire rapide double port, voir créer un automate à états pour réaliser plusieurs lectures (ou écritures par cycle) dans des bancs mémoires entrelacés ? Dans les chapitres suivant, nous tenterons aussi de répondre à ce type de question.

1.2 Méthodologie d'évaluation de performance

L'environnement d'évaluation est le suivant : la bibliothèque 65-nm CMOS *General Power* de ST a été utilisée avec Catapult-C pour générer le code RTL. L'estimation de la puissance consommée ainsi que la surface est réalisée avec Design Compiler de Synopsis, avant placement-routage et sans activer les capacités pour Catapult-C de réduction de la consommation du circuit en faisant du *clock gating* [2]. Enfin nous supposons que le circuit fonctionne en continu, c'est à dire que la réalisation du calcul prend tout le temps disponible et qu'ainsi, la puissance consommée est la somme de la puissance dynamique et de la puissance statique. L'énergie consommée sur cet intervalle de temps est donc le produit du temps par la puissance (régime permanent).

Quant à l'exploration de l'espace des configurations des différents paramètres, c'est une exploration exhaustive qui a été réalisée sur la ferme de calcul de ST, afin d'analyser l'impact des différentes configurations dans tous les cas, et non pas seulement ceux qu'on pourrait penser être intéressants. Les fréquences de synthèse vont de 100 MHz à 600 MHz par pas de 100 MHz, Les valeurs de ii vont de 1 à 8 (cycles) et le mode *auto* indique dans les tableaux que le choix est laissé à Catapult-C.

1.3 Cibles architecturales

Trois processeurs généralistes ont été utilisés pour évaluer l'impact de ces transformations :
- le microcontroleur / microprocesseur ST XP70 de STMicroelectronics,
- le processeur Cortex-A9 de ARM, dans son instance OMAP4 de Texas Instrument,
- le processeur basse consommation Penryn SU9300 d'Intel.

2. méthode de réduction de la consommation dynamique qui consiste à couper le signal d'horloge d'une partie du circuit, lorsque celle-ci est inactive

1.3.1 Processeur ST XP 70

1.3.1.1 Présentation du coeur XP70

Le processeur XP70 est un microprocesseur RISC 32 bits développé par ST et utilisé dans de nombreux produits. Il est actuellement dans sa quatrième version. En technologie 60 nm, il a une surface d'environ 40 KGates. Sa fréquence maximale de fonctionnement est de 450 MHz en 65 nm et de 600 MHz en 32 nm. Sa consommation électrique est particulièrement basse : $0,05mW$ par MHz, soit 23 mW à 450 MHz. En plus de sa faible consommation électrique, son autre particularité est d'être très paramétrable, ce qui va permettre de tester l'impact de choix architecturaux sur la performance des algorithmes étudiés.

Il a en plus des éléments qu'on ne retrouve pas sur les autres processeurs testés. Ainsi Il possède des compteurs de boucle matériels, ce qui évite les erreurs de prédiction de branchement. Cela évite aussi d'avoir à développer des prédicteurs de branchement de plus en plus gros pour minimiser le pourcentage d'erreur de branchement.

Il n'a pas de cache, mais des mémoires rapides (pour les instructions et pour les données, il a donc une architecture de type Harvard) appelées TCPM (*Tightly Coupled Program Memory*) et TCDM (*Tightly Coupled Data Memory*). Ces mémoires ont des temps d'accès réglables à 1 cycle, 2 cycles ou 10 cycles. Le bus mémoire est d'une largeur de 64 bits.

La taille du banc de registres est paramétrable avec 16 ou 32 registres. Enfin le pipeline ne comporte que 7 étages : *fetch*, *decode*, *align*, *execute*, mem_1, mem_2, *writeback*.

1.3.1.2 Extensions du jeu d'instructions du coeur XP70

Plusieurs extensions du jeu de base du XP70 sont disponibles :

- FPx : extension flottante 32 bits respectant la norme IEEE754, mais avec un seul mode d'arrondi (arrondi par défaut) et sans nombre dénormalisé pour des raisons d'optimisation en surface,

- MPx : extension Multimédia pour l'arithmétique en virgule fixe en 16, 32 et 64 bits (Q1.63, Q1.31 et Q1.15), pour l'arithmétique saturée et dotée d'un accélérateur Viterbi.

- VECx : extension vectorielle 64/128 bits pour du calcul SIMD.

Le jeu d'instructions VECx est particulièrement complet. Par son contenu, il est proche du jeu d'instructions Neon d'ARM, ce qui est logique vu les domaines applicatifs ciblés : traitement du signal et applications multimédia pour processeur embarqué (faible consommation). Il a donc en plus des instructions généralistes (accès mémoire, arithmétique, comparaison, mélange), des instructions spécialisées. Il a entre autres un MAC (*Multiplier-Accumulator*) 16 bits, un moyenneur adapté au traitement du signal. La raison est la suivante : lors de l'utilisation de filtres, la première opération à laquelle on pense – pour accélérer un code – est l'instruction MAC qui multiplie des données 8 bits et les accumule dans un registre 16 bits : $acc_{16} \leftarrow h_8 \times x_8 + acc_{16}$. Si le format de sortie du filtre est de 8 bits, il est donc nécessaire d'arrondir le résultat de l'accumulateur et de le convertir en 8 bits : $y_8 = (acc_{16} + 128)>>8$. Il existe une instruction qui réalise ces deux opérations en VECx.

Dans ce même objectif d'efficacité énergétique et de minimisation de surface, les instruc-

tions de LOAD et STORE SIMD sont 64 / 128 bits. L'instruction LOAD lit 64 bits depuis la mémoire (8×8 bits) et les stocke dans un registre 128 bits (8×16 bits), en faisant à la volée une conversion 8 bits $\rightarrow$ 16 bits. L'instruction STORE possède le mécanisme inverse.

On évite ainsi d'ajouter des instructions de conversions (entrelacement du registre contenant les données 8 bits avec un registre contenant des zéros pour la promotion pour les données 8 bits non signées, ou extension du bit de signe pour les données signées via une instruction dédiée). Cela permet aussi d'avoir un code plus compact. Si le LOAD avait chargé 128 bits (16×8 bits), il aurait fallu avoir deux registres pour stocker les 16 valeurs 16 bits. Cela aurait impliqué de dupliquer l'ensemble du code de calcul d'un facteur deux et donc d'avoir une empreinte mémoire deux fois plus grande. Cette astuce architecturale est aussi présente en Neon, mais est absente en Altivec, SSE et AVX. S'il s'avérait que cela pose des problèmes de performance, il est toujours possible de dérouler d'un facteur 2 la boucle (*unroll & jam*) et de masquer en partie la latence des instructions les plus longues.

Le bloc matériel VECx est composé de 190 KGates pour une surface de 0.42 mm^2 (pour une fréquence de fonctionnement de 420 MHZ, et 0.30 mm^2 à 200 MHz). Si l'on fait l'hypothèse que la consommation du bloc est proportionnelle à sa surface – hypothèse fausse, mais la consommation de l'extension n'était pas mesurable simplement – alors VECx doit apporter une accélération d'un facteur supérieur au ratio du nombre de portes soit : $190/40 = \times 4.75$ pour être utile. Un présentation de l'environnement est disponible dans [32].

1.3.2 Processeur ARM Cortex-A9

La famille de processeurs ARM Cortex-A est très répandue dans le monde de l'embarqué. Elle est ultra-majoritaire dans les téléphones portables haut de gamme de type *smartphone* et dans les tablettes. Ces processeurs sont en fait instanciés sous forme de SoC (*System on Chip*) par rapport à un design de référence fourni par ARM. Les enjeux commerciaux sont très importants. La guerre commerciale qui opposent les constructeurs de téléphone (Apple, Samsung, HTC, Google) et la guerre commerciale qui opposent les concepteurs de SoC (Samsung, Samsung pour Apple, Qualcomm, Texas Instrument, ST) et les fondeurs font qu'il n'est pas possible d'avoir de chiffres précis. Pour un même design, la fréquence de fonctionnement peut varier (tout comme l'asservissement de la fréquence, vis à vis de la puissance dissipée), la taille du cache (L1 et L2) peut être différente.

La carte utilisée pour les benchmarks est une carte Pandaboard-ES à 1.2 GHz avec un OMAP4 4460 de Texas Instrument en techno 45 nm. La consommation moyenne (TDP) de la version précédente de la carte, était de 1 W pour un processeur OMAP4 4430 à 1 GHz. Par la suite nous feront l'hypothèse que la consommation de notre SoC est de 1.2 W, même lors de calculs sur l'unité SIMD.

En plus de leur faible consommation, ces processeurs possèdent des instructions à prédicat qui sont exécutées conditionnellement à l'état d'un bit du registre d'état (*status register*). Le fragment de programme 1.1 correspond à l'algorithme de calcul du PGCD de deux entiers par l'algorithme d'Euclide à base de soustraction. Ce code est un bon exemple de *pire code* pour un processeur RISC, car la mise à jour des deux entiers se fait via une structure de contrôle `if-then-else` et que le nombre d'itération de la boucle n'est pas déterministe.

```
1  int gcd(int a, int b)
2  {
3     while (a != b) {
4        if (a > b) {
5           a = a - b;
6        } else {
7           b = b - a;
8        }
9     }
10    return a;
11 }
```

Prog 1.1– code C de l'algorithme d'Euclide par soustraction ()

Le fragment de programme 1.2 correspond au code C précédent. On peut voir les trois instructions de branchement `B` pour les clauses vraie et fausse du test et l'instruction `BEQ` pour la boucle.

```
1  gcd CMP r0, r1
2      BEQ end
3      BLT lt
4      SUB r0, r0, r1
5      B   gcd
6  lt  SUB r1, r1, r0
7      B   gcd
8  end
```

Prog 1.2– code assembleur classique de l'algorithme d'Euclide par soustraction

Avec les instructions à prédicat, les problèmes de prédiction et de branchement disparaissent (fragment de code 1.3). Tant que le résultat n'est pas égal à zéro (`BNE`) c'est soit la première soustraction (ligne 2) soit la seconde (ligne 3) qui est réalisée.

```
1  gcd CMP r0, r1
2      SUBGT r0, r0, r1
3      SUBLT r1, r1, r0
4      BNE  gcd
5  end
```

Prog 1.3– code assembleur avec instructions à exécution conditionnelle de l'algorithme d'Euclide par soustraction

1.3.3 Processeur Intel Penryn ULV

Le processeur Penryn ULV (*Ultra Low Voltage*) SU9300 est un processeur basse consommation d'Intel destiné au marché des portables haut de gamme de type *ultrabook*. Il est en technologie 45 nm et sa fréquence d'horloge nominale est de 1.2 GHz. De ce fait, ses performances sont directement comparables à celles du Cortex-A9. Afin de le comparer au processeur XP70 un facteur correctif sera appliqué pour prendre en compte la différence de technologie de gravure.

Il appartient à la première génération d'architecture Core d'Intel et peut exécuter les jeux d'instructions SIMD SSE jusqu'à SSSE3. Cela est important car à partir de cette version du jeu d'instructions SIMD, le calcul des vecteurs non alignés (voir chapitre sur les filtres non récursifs) se fait avec une seule instruction (`_mm_alignr_epi8`) et non plus trois, comme c'était le cas en SSE2 et SSE3 (les registres sont décalés à gauche et à droite grâce à des *shift* 128 bits et combiné avec un OU bit-à-bit). Grâce à cela, il n'est plus handicapé comme pouvaient l'être ses

prédécesseurs.

Enfin son TDP est de 10 W. Il est donc intéressant de voir si le rapport de performance avec le Cortex-A9 est du même ordre de grandeur ou pas. C'est à dire si les optimisations matérielles présentes (pipeline avec de nombreux *bypasses*, prédicteurs de branchement, *prefetch* de cache, débit d'instructions SIMD de 1 contre 2 pour le Cortex-A9, ...) permettent de compenser la différence de taille et de consommation. Notons toutefois que le fait qu'il soit doté d'un cache plus grand (2Mo de L2 contre typiquement 512 Ko pour un Cortex-A9) n'est pas pris en compte pour corriger la surface "utile" du processeur (de 107 mm^2) ni le nombre total de transistors (410M) principalement utilisés pour le cache.

1.3.4 Autres processeurs ?

D'autres processeurs auraient pu être évalués. Notamment des processeurs moins généralistes et plus adaptés au domaine du traitement du signal comme les DSP (*Digital Signal Processor*) C6x de Texas Instrument. Ces processeurs consomment peu d'énergie (1 W pour le C64x en techno 90 nm), sont doté d'un micro-SIMD 32 bits (*Sub-Word Parallelism*) et sont capables de faire deux accès mémoire par cycle sans contrainte (la mémoire étant composée de mots 32 ou 64 bits selon les versions, mais permettant deux accès concurrents de 8 bits sur le même mot, sans pénalité pour le bus de donnée). Enfin la nouvelle génération (C66x) est multi-coeurs ce qui les rend encore plus compétitifs pour des tâches de calcul intensif dans le domaine de l'embarqué. Si cette famille de DSP n'a pas été évaluée, c'est donc uniquement pour des raisons de temps.

1.4 Conclusion du chapitre

Ce chapitre a présenté les objectifs visés par la synthèse automatique de haut niveau (HLS) et en particulier l'outil Catapult-C de Capylto, utilisé par STMicroelectronics en production.

Afin d'évaluer l'impact des transformations de haut niveau (qui seront détaillées dans les trois chapitres applicatifs), trois processeurs généralistes ont été présentés. Il permettront aussi de positionner les performances de l'ASIC par rapport à des composants généralistes COTS.

CHAPITRE 2

MÉTAPROGRAMMATION

Ce chapitre présente la stratégie de génération de code utilisée pour automatiser l'écriture de codes de traitement du signal et des images (algorithmes réguliers) tout en prenant en compte à la fois les spécificités de Catapult-C et les optimisations afférentes à ce type de code. Ce chapitre couvrira dans un premier temps les principes généraux de la métaprogrammation au sens large avant de se focaliser sur la métaprogrammation par préprocesseur. Enfin, nous entrerons dans les détails des choix d'implémentation de notre outil.

2.1 Principes de la métaprogrammation

On peut définir simplement la métaprogrammation comme l'écriture de programmes (les « méta-programmes ») capables, au sein d'un système adéquat, d'extraire, de transformer et de générer des fragments de code afin de produire de nouveaux programmes (les programmes « objets ») répondant à un besoin précis. L'exemple le plus simple de méta-programme est le compilateur qui, à partir de fragments de code dans un langage donné, est capable d'extraire des informations et de générer un nouveau code, cette fois en langage machine, qui sera plus tard exécuté par le processeur. En pratique, on distingue souvent deux types de systèmes de métaprogrammation :

– *les analyseurs/transformateurs de code* qui utilisent la structure et l'environnement d'un programme afin de calculer un nouveau fragment de programme. On retrouve dans cette catégorie des outils comme PIPS [31] ;

– *les générateurs de programmes* dont l'utilisation première est de résoudre une famille de problèmes connexes dont les solutions sont souvent des variations d'un modèle connu. Pour ce faire, ces systèmes génèrent un nouveau programme capable de résoudre une instance particulière de ces problèmes et optimiser pour cette résolution. On distingue plus précisément les générateurs *statiques* qui génèrent un code nécessitant une phase de compilation classique et les générateurs capables de construire des programmes et de les exécuter à la volée. On parle alors de *multi-stage programming* [62].

L'avantage principal est le fait que la métaprogrammation permet, à partir d'une description générique et adaptée au problème traité, de produire un nouveau programme dont les performances sont plus élevées. De manière empirique, cette augmentation des performances est proportionnelle à la quantité d'éléments évaluables statiquement au sein du programme initial.

2.1.1 L'évaluation partielle

Parmi les différentes techniques de métaprogrammation, on s'intéresse plus particulièrement aux techniques d'*évaluation partielle* [26, 33]. Effectuer l'évaluation partielle d'un code consiste à discerner les parties statiques du programme, c'est-à-dire les parties du code calculables à la compilation, des parties dynamiques, calculables à l'exécution. Le compilateur est alors amené à évaluer cette partie statique et à générer un code – dit code *résiduel* – ne contenant plus que les parties dynamiques. On retrouve donc ici une définition de la métaprogrammation – la programmation par extraction et génération de codes à partir d'une description générique – dans un cadre particulier : au lieu de nécessiter à la fois un langage de spécification et un langage pour le code résiduel, l'évaluation partielle ne nécessite qu'un seul et unique langage. Mais, pour permettre l'évaluation partielle, ce langage doit permettre l'annotation explicite des structures statiques. Ceci nécessite donc un langage à deux niveaux dans lequel il est possible de manipuler de tels marqueurs.

De manière plus formelle, on considère qu'un programme est une fonction Φ qui s'applique à la fois à des entrées définies au moment de la compilation et des entrées définies au moment de l'exécution afin de produire un résultat à l'exécution.

$$\Phi \ : \ (I_{static},\ I_{dynamic}) \rightarrow O$$

On définit alors un *évaluateur partiel* Γ comme une fonction qui transforme le couple $(\Phi,\ I_{static})$ en un nouveau programme – le code résiduel – Φ^* dans lequel I_{static} a été entièrement évalué.

$$\Gamma \ : \ (\Phi,\ I_{static}) \rightarrow \Phi^*$$

Bien entendu, Φ^* est tel que Φ et Φ^* restent fonctionnellement équivalents mais, en général, les performances de Φ^* sont supérieures à celle de Φ.

$$\Phi \ \equiv \ \Phi^* \ : \ (I_{dynamic}) \rightarrow O$$

Afin de déterminer les portions de code à évaluer statiquement, un évaluateur partiel doit analyser le code source original pour y détecter des structures et des données statiques. Cette détection permet par la suite de spécialiser les fragments de code.

2.1.2 Outils pour la métaprogrammation

La mise en œuvre d'un mécanisme d'évaluation partielle en C++ revient à fournir un outil capable d'analyser un code contenant des éléments statiques. Il est évidemment possible de définir un outil externe au langage qui effectuerait cette tâche au niveau du code source et génèrerait un nouveau code source partiellement évalué comme le fait par exemple l'outil Tempo [13]. Cette solution, bien qu'envisageable, nous paraît peu élégante dans le sens où elle complexifie la chaîne de développement. Il nous semble préférable de trouver un moyen d'annoter directement *au sein du langage* C++ ces éléments statiques : la métaprogrammation par template et la métaprogrammation via le préprocesseur.

2.1.2.1 Métaprogrammation par template

Les *templates* (ou patrons) sont un mécanismes du langage C++ dont le but principal est de fournir un support pour la programmation dite « générique », favorisant ainsi la réutilisation de code. Un patron définit une famille de classes ou de fonctions paramétrées par une liste de valeurs ou de types. Fixer les paramètres d'une classe ou d'une fonction *template* permet de *l'instancier* et donc de fournir au compilateur un fragment de code complet et compilable. Les *templates* supportent aussi un mécanisme de *spécialisation partielle*. Le résultat d'une telle spécialisation est alors non pas un fragment de code compilable mais un nouveau patron devant lui-même être instancié. Ainsi, il devient possible de fournir des patrons dépendant de certaines caractéristiques paramétrables et d'optimiser le code ainsi généré en fonction de ses caractéristiques. Lorsque le compilateur tente de résoudre un type *template,* il commence par chercher la spécialisation la plus complète et explore le treillis des spécialisations partielles jusqu'à trouver un cas valide. Nous retrouvons donc ici la définition même de l'évaluation partielle : la programmation par spécialisation. En effet, l'instanciation d'une classe *template* correspond à l'application d'une fonction qui calcule une nouvelle classe à partir d'un patron et de ses paramètres. De la même manière, la spécialisation partielle de *template* fournit un équivalent du mécanisme de *filtrage* (*pattern matching*) présent dans de nombreux langages fonctionnels (ML, Haskell…).

On peut en fait démontrer, par construction [66], que les *templates* forment un sous-ensemble Turing-complet du C++. Un des premiers exemples de tels programmes fut proposé par Erwin Unruh [64]. Ce programme ne s'exécutait pas mais renvoyait une série de messages à la compilation qui énumérait la liste des N premiers nombres premiers. Il démontrait ainsi que les patrons permettaient d'exprimer plus que la simple généricité des classes et des fonctions.

2.1.2.2 Métaprogrammation via le préprocesseur

De manière similaire à la métaprogrammation par template, la métaprogrammation par le préprocesseur repose sur le détournement des macros et macro-fonctions du préprocesseur afin de définir des pseudos-structures de données et de contrôles.

Contrairement aux templates, les macros et macro-fonctions sont résolues en amont de toute analyse syntaxique ou sémantique et n'agissent que sur des tokens, limitant ainsi leurs applications à des remplacement textuels directs. Néanmoins, on montre qu'il est possible d'émuler des stratégies comme la récursivité ou la sélection conditionelle de macros. Des bibliothèques comme Boost.Preprocessor ou Chaos en sont la démonstration. En terme de complexité, si les templates sont effectivement Turing-complets, on démontre que la métaprogrammation par macros est elle aussi Turing-complète si on limite la taille de la mémoire de la machine de Turing équivalente au nombre maximal d'appels réentrants supportés par le préprocesseur (soit en général de 4096 à 65536 appels) avant de rendre le système inutilisable par son temps de compilation. Les bibliothèques `Boost.ConceptCheck` [56] ou `Boost.Local` utilisent un tel système afin d'émuler des constructions de haut-niveau et de fournir une interface familière à leur utilisateurs.

2.2 Fonctionnement du préprocesseur

Dans nos travaux, nous nous concentrerons sur la métaprogrammation via le préprocesseur du fait des limitations du compilateur CatapultC qui ne permet pas de gérer correctement les différents idiomes de la métaprogrammation par template de manière standard.

2.2.1 Eléments de bases du préprocesseur

Identifions tout d'abord les éléments essentiels du préprocesseur que sont les **tokens**, les **macros** et leurs **arguments**.

2.2.1.1 Token

L'unité fondamentale du préprocesseur est le **token**. Bien que légèrement différents des **tokens** classique tel que le standard les définit, les **tokens** du préprocesseur sont constitués des identifiants, symboles d'opérateurs et des littéraux.

2.2.1.2 Macros

Il existe deux types de macros : les macro-objets et les macro-fonctions. On définit un macro-objet via la syntaxe :

`#define` *identifier replacement-list*

avec *identifier* définissant le nom de la macro et *replacement-list* contenant une séquence de zéros ou plusieurs tokens. Lorsque *identifier* apparaît par la suite dans le texte du programme, il est remplacé par la séquence de tokens contenue dans *replacement-list*.

Les macro-fonctions, qui se comportent comme des méta-fonctions lors de la phase d'expansion des macros, se définissent comme suit :

`#define` *identifier(a1, a2, ... an) replacement-list*

où chaque a_i est un identifiant nommant un paramètre de macro. Lorsque l'identifiant de la macro apparait dans le source suivi d'une liste d'argument satisfaisante, la macro et sa liste d'arguments sont remplacés par les tokens contenus dans *replacement-list* une fois les arguments remplacés par leur valeurs effectives.

2.2.1.3 Arguments de macros

On définit un argument de macro comme une séquence non-vide composée de :

- Tokens de préprocesseur autres que le token virgule ou parenthèse.
- Tokens de préprocesseur entourés d'une paire de parenthèses cohérente.

Cette définition a des conséquences non-triviales pour la métaprogrammation par macros. Comme les symboles `,`, `(` et `)` ont un statut spécial, ils ne peuvent jamais apparaitre comme argument de macro. Par exemple, le code suivant est mal formé.

```
1 #define FOO(X) X // Macro identite
2 FOO(,)
3 FOO())
```

Prog 2.1– Jeu de paramètres de macros incorrects

Le premier appel de `FOO` est incorrect car la virgule peut être interprété comme soit une virgule sans parenthèse ou comme l'appel de `FOO` avec deux arguments vide. Le deuxième appel est incorrect car il comprend une parenthèse esseulée.

De manière similaire, le préprocesseur n'a aucune notion de paire de symboles comme les accolades (`{`, `}`), les crochets (`[`, `]`) ou les accolades anguleuses (`<`, `>`). Ainsi, le code suivant est

invalide :

```
1 FOO(std::pair<int, long>)      // Deux arguments
2 FOO({ int x = 1, y = 2; return x+y; }) // Deux arguments
```

Prog 2.2– Démonstration de l'ignorance des séparateurs

Le seul moyen est donc de passer un token unique en l'enveloppant de parenthèses.

```
1 FOO((std::pair<int,int>))       // Un argument
2 FOO(({ int x = 1, y = 2; return x+y; })) // Un argument
```

Prog 2.3– Passage d'argument contenant une virgule

2.2.2 Fonction de base

Le préprocesseur fournit deux opérations de base sur les tokens.

2.2.2.1 Transformation en chaîne de caractères

Le préprocesseur permet de transformer une liste de tokens en chaîne de caractères en utilisant le préfixe `#`.

```
1 #define MAKE_A_STRING(Arg) #Arg
```

Prog 2.4– Génération de chaine constante

Il faut noter neanmoins que les règles de substitutions des paramètres de macros interagissent de manière incongrues avec #. En effet, le code suivant :

```
1 #define FOO(A) -A
2
3 std::cout << MAKE_A_STRING(FOO(3)) << "\n";
```

Prog 2.5– Génération de chaine constante

a pour sortie la chaine `FOO(3)`. # empêche purement et simplement la substitution des arguments de s'effectuer. On peut néanmoins modifier légèrement la macro `MAKE_A_STRING` afin d'évaluer son argument avant sa transformation.

```
1 #define FOO(A) -A
2
3 #define MAKE_A_STRING_IMPL(Arg) #Arg
4 #define MAKE_A_STRING(Arg) MAKE_A_STRING_IMPL(Arg)
5
6 std::cout << MAKE_A_STRING(FOO(3)) << "\n";
```

Prog 2.6– Génération de chaine constante post-substitution

Cette version du code génère bien la chaine `-3`. En effet, avant de procéder au remplacement de la macro interne `MAKE_A_STRING_IMPL`, le préprocesseur procéde à l'évaluation complète de son paramètre `Arg`. Ainsi, # s'applique bien au substitué de `Arg`. Nous verrons par la suite que ce comportement va nous permettre d'émuler un certain nombre de structures intéressantes.

2.2.2.2 Concaténation de tokens

L'opérateur `##` permet de concaténer deux tokens.

```
1 #define version(symbole) symbole ## v123
2 int version(variable);
```

Prog 2.7– Concaténation de tokens

Le code généré est alors `variablev123`. De la même façon que #, l'application de ## n'effectue pas de substitution. Ce défaut est corrigeable de la même façon que pour #.

2.3 Application au design d'interface modulaire

Afin de fournir une interface de haut-niveau pour nos transformations de code, nous avons décidé d'émuler la notation IDL (*Interface Description Language*) pour les fonctions de filtrages. Cette notation a pour but de :

- Supporter les différentes structures de données fournies par Catapult-C : tableau et flux ;
- Encapsuler l'expertise issue de l'analyse du comportement de Catapult-C afin de reproduire fidèlement le comportement du code écrit "à la main" ;
- Permettre l'extension du système à de nouvelles stratégies d'optimisations.

2.3.1 Implémentation

Le code de génération des filtres FIR est basé sur l'analyse des différentes variantes couramment utilisées. Un filtre FIR se compose toujours de :

- Une phase d'initialisation des données dans une zone (mémoire ou temporaire) dépendante de la stratégie d'optimisation (avec ou sans rotation de registres) ;
- Un corps de boucle dans lequel les points de variations dépendent des types de données utilisées et de la stratégie d'optimisation ;
- Une phase de finalisation potentiellement vide.

Ces phases doivent bien entendu être paramétrées par la description des données d'entrées, de sorties et de la liste des coefficients du filtre. Le squelette de base d'un FIR est donc assimilable à une macro de la forme :

```
#define FIR( (Input, Coefficient, Output) )
```

2.3.1.1 Spécification des entrées sorties

la première étape consiste à définir un protocole afin de pouvoir spécifier le type, la taille et le nom des variables utilisées en entrée et en sortie d'une fonction de filtrage. Plusieurs caractéristiques sont à prendre en compte :

- Chaque paramètre du filtre peut être soit un simple tableau de taille variable soit une instance de la classe CatapultC `ac_channel` ;
- Chaque paramètre contient des valeurs typées ;
- Chaque paramètre possède une taille dans le cas des tableaux.

Afin de simplifier l'écriture de ces définitions, nous avons choisi de définir une série de macros imitant une interface de type IDL. Deux pseudo-types paramétriques sont donc définis :

– `array(T,N)` qui représente un tableau C de `N` éléments de type `T`
– `ac_channel(T)` qui représente un `ac_channel` contenant des éléments de type `T`
Par exemple, l'appel

```
FIR(((ac_channel(uint8),X), (array(sint16,M),H), (ac_channel(uint8),Y)))
```

génère le code nécessaire pour l'appel d'un filtre traitant des `ac_channel` d'entiers 8 bits non-signés en entrée et en sortie et récupérant les coefficients du filtre dans un tableau d'entiers 16 bits signés.

Le challenge est ensuite d'extraire facilement les données du pseudo-type. Il est important pour nous de récupérer chaque parcelle d'information encodée dans la macro définissant un paramètre du filtre, à savoir le type des données et leur taille. Nous définissons pour cela deux macros qui vont se charger de cette extraction.

La première macro `TYPE(X)` va prendre un descripteur de type IDL paramètrique et retourner le type de données sous-jacent. On remarque que la position de cette information dépend du type effectif du paramètre (tableau ou `ac_channel`). Une stratégie classique consiste à utiliser le fait que la construction `array(T,N)` ressemble à un appel de macro pour construire une série de tokens qui, une fois complétée, se résoudra comme un appel de macro spécifique. `TYPE` se définit donc comme suit :

```
1 #define CAT_IMPL(A,B) A ## B
2 #define CAT(A,B) CAT_IMPL(A,B)
3
4 #define TYPE_ac_channel(T) T
5 #define TYPE_array(T,N) T
6 #define TYPE(Type) CAT(TYPE_, Type)
```

Prog 2.8– Extraction des informations de type

Cette macro utilise une version correcte de la fonction de concaténation de tokens afin de générer un token étant lui même résolu comme un appel de macro. Le type IDL est utilisé comme suffixe afin de discriminer l'appel en cours. La même technique est appliquée pour l'évaluation de la taille d'un paramètre.

```
1 #define SIZE_array(T,N) T
2 #define SIZE(Type) CAT(SIZE_, Type)
```

Prog 2.9– Extraction des informations de taille

Notons néanmoins que ici, seul le cas tableau est géré. Cette technique est facilement extensible en ajoutant pour chacune des macros de nouveaux cas liés à d'éventuels nouveaux types de données.

2.3.1.2 Fonction de filtrage

L'interface principale du générateur de filtre se base sur la définition des trois types d'entrées/sorties. En fonction des types respectifs de ces paramètres, le squelette de base du filtre va se modifier. Dans le cas d'un calcul via des tableaux, le code est le suivant :

```
1 void fir(sint8 X[256], sint8 H[3], sint8 Y[256])
2 {
3    int i;
4    for(i=0; i<256; i++) {
```

```
5         sint8 x;
6         sint8 y;
7         sint8 r = 1<<7; // arrondi = 1 << (8-1) = 0.5
8         static sint8 RF[3];
9
10        x = X[i];
11        {
12           int k;
13           for(k=3-1; k>0; k--) RF[k] = RF[k-1];
14           RF[0]= x;
15        };
16        {
17           int k;
18           sint16 y16;
19           y16 = r;
20           for(k=0; k<3; k++) {
21              y16 += RF[k+i]*H[k];
22           }
23           y = (uint8) (y16>>8);
24        };
25        Y[i] = y;
26     }
27  }
```

Prog 2.10– Code FIR pour tableaux

Pour les cas utilisant le type `ac_channel` :

```
1  void fir(ac_channel<uint8>& X, sint8 H[3], ac_channel<uint8>& Y)
2  {
3     uint8 x;
4     uint8 y;
5     sint8 r=1<<7;
6     static sint8 RF[3];
7
8     x = X.load();
9     {
10       int k;
11       for(k=3-1; k>0; k--) RF[k] = RF[k-1];
12       RF[0]= x;
13    };
14    {
15       int k;
16       sint16 y16;
17
18       y16 = r;
19       for(k=0;k<3;k++) {
20          y16+=RF[k]*H[k];
21       }
22       y = (uint8) (y16>>8);
23    };
24    Y.write(y);
25 }
```

Prog 2.11– Code FIR utilisant `ac_channel`

On voit donc apparaitre un schéma mettant en jeu plusieurs parties :

– la déclaration des variables locales nécessaires au calcul du filtrage ;
– la déclaration de la zone nécessaire à la rotation de registres ;
– le code chargeant les nouvelles données depuis l'entrée ;
– le code prenant en charge la rotation de registres ;
– le code effectuant le calcul proprement dit ;
– le code stockant le résultat final.

Nous commençons donc par définir une macro générant le prototype de la fonction ainsi que le prologue nécessaire à son exécution.

```
#define FIR_GEN_array(T,N) FIR_GEN_array_IN
#define FIR_GEN_array_IN(ARGS)                                  \
{                                                               \
  int i;                                                        \
  for(i=0; i<N; i++)                                            \
    FILTER( AT(ARGS,0,3), AT(ARGS,1,3), AT(ARGS,2,3) , i)       \
}                                                               \
/**/

#define FIR_GEN_ac_channel(T) FIR_GEN_ac_channel_IN
#define FIR_GEN_ac_channel_IN(ARGS)                             \
{                                                               \
  FILTER( AT(ARGS,0,3), AT(ARGS,1,3), AT(ARGS,2,3) , ~)         \
}                                                               \
/**/

#define INOUT_array(T,N) INOUT_array_IN
#define INOUT_array_IN(ARGS)                                    \
      TYPE(AT(ARGS,0,2)) AT(ARGS,1,2)[SIZE(AT(ARGS,0,2))]       \
/**/

#define INOUT_ac_channel(T) INOUT_ac_channel_IN
#define INOUT_ac_channel_IN(ARGS)                               \
          ac_channel<TYPE(AT(ARGS,0,2))>& AT(ARGS,1,2)          \
/**/

#define INOUT(T) CAT(INOUT_,AT(T,0,2))(T)

#define FIR(ARGS)                                               \
void fir( INOUT(AT(ARGS,0,3))                                   \
        , INOUT(AT(ARGS,1,3))                                   \
        , INOUT(AT(ARGS,2,3))                                   \
        )                                                       \
CAT(FIR_GEN_, AT(AT(ARGS,0,3),0,2) )(ARGS)                      \
/**/
```

Prog 2.12– Génération d'un FIR

Cette macro et ses macros annexes extraient donc de sa liste d'argument les informations nécessaire à la génération de son code. Les paramètres de `FIR` sont passés sous la forme d'un triplet de paramètres séparés par des virgules. Chaque parcelle de cet argument est récupérable par la macro `AT` définie ainsi :

```
#define AT_0_1(T0)  T0
#define AT_0_2(T0,T1) T0
```

```
3 #define AT_1_2(T0,T1) T1
4 #define AT_0_3(T0,T1,T2) T0
5 #define AT_1_3(T0,T1,T2) T1
6 #define AT_2_3(T0,T1,T2) T2
7 // ... etc ...
8
9 #define AT(Tuple,Index,Size) AT_ ## Index ## _ ## Size Tuple
```

Prog 2.13– Macro AT - Accés énumérable

Assez trivialement, nous utilisons une énumération directe des cas d'appels de `AT` afin de récupérer l'élément correspondant dans la suite de valeurs séparées par des virgules passées en argument. La concaténation de l'index et de la taille du *pseudo-tuple* nous permet un accès en $O(1)$ à l'élément souhaité dans la macro.

`FIR` utilise alors la technique vue précédemment de génération de nom de macro basée sur les types afin de choisir quel corps de fonction est à produire. Au sein de ce bloc, la macro `FILTER` va générer la partie commune du code.

```
1  #define FILTER(Input, Coefficient, Output, Offset)          \
2  {                                                           \
3    TYPE(AT(Input,0,2)) x;                                    \
4    TYPE(AT(Output,0,2)) y;                                   \
5                                                              \
6    static TYPE(AT(Coefficient,0,2))                          \
7         RD[SIZE(AT(Coefficient,0,2))];                       \
8                                                              \
9    LOAD(Input,Offset,x);                                     \
10                                                             \
11   REGISTER_PUSH_VALUE(RD, x);                               \
12   APPLY_FILTER( AT(Input,0,2),RD, Coefficient, y, Offset);  \
13                                                             \
14   STORE(Output,Offset,y);                                   \
15 }                                                           \
16 /**/
```

Prog 2.14– Structure interne de la macro FIR

De manière similaire, les macros annexes utilisées par `FILTER` font usage des techniques mise en avant précédemment.

Les macros `LOAD` et `STORE` prennent en charge la lecture et l'écriture dans le tableau de données ou dans le `ac_channel` correspondant en utilisant leur paramètres contenant la variable utilisée, la valeur à lire ou écrire et un offset dans le tableau.

```
1  #define LOAD_ac_channel(T) LOAD_ac_channel_in
2  #define LOAD_ac_channel_in(Destination, Offset, Value)     \
3        Value = Destination.read()                           \
4  /**/
5
6  #define LOAD_array(T,N) LOAD_array_in
7  #define LOAD_array_in(Destination, Offset, Value)          \
8        Value = Destination[Offset]                          \
9  /**/
10
11 #define LOAD(Source, Offset, Value)                            \
```

```
12        CAT(LOAD_, AT(Source,0,2)) (AT(Source,1,2), Offset, Value) \
13  /**/
14
15  #define STORE_ac_channel(T) STORE_ac_channel_in
16  #define STORE_ac_channel_in(Destination, Offset, Value)     \
17        Destination.write(Value)                              \
18  /**/
19
20  #define STORE_array(T,N) STORE_array_in
21  #define STORE_array_in(Destination, Offset, Value)          \
22        Destination[Offset] = Value                           \
23  /**/
24
25  #define STORE(Dest, Offset, Value)                              \
26        CAT(STORE_, AT(Dest,0,2)) (AT(Dest,1,2), Offset, Value) \
27  /**/
```

Prog 2.15– Macros de lecture/écriture

La macro `REGISTER_PUSH_VALUE` gére le pré-remplissage du tableau utilisé pour la rotation de registres.

```
1  #define REGISTER_PUSH_VALUE(Register, Value)                \
2  {                                                           \
3    int k;                                                    \
4    for(k=M-1; k>0; k--) Register[k] = Register[k-1];         \
5    Register[0]= Value;                                          \
6  }                                                           \
7  /**/
```

Prog 2.16– Macro pour la rotation de registre

La macro `APPLY_FILTER` est en charge d'effectuer le calcul du filtrage en lui-même. Elle s'appuie sur une macro annexe `INDEX` qui permet la récupération d'un élément d'un tableau ou d'un `ac_channel` de manière générique.

```
1  #define INDEX_ac_channel(T) INDEX_ac_channel_in
2  #define INDEX_array(T,N) INDEX_array_in
3
4  #define INDEX_ac_channel_in(Register, Index, Offset) Register[Index]
5  #define INDEX_array_in(Register, Index, Offset) Register[Index+Offset]
6
7  #define INDEX(Type,Source, Index, Offset) CAT(INDEX_,Type)  \
8        (Source, Index, Offset)                               \
9  /**/
10
11 #define APPLY_FILTER(Type, Register, Coefficient, Output, Offset) \
12 {                                                                  \
13   int k;                                                        \
14   sint16 y16;                                                   \
15   y16 = 0;                                                      \
16   for(k=0; k<M; k++) {                                          \
17     y16 += INDEX(Type,Register,k,Offset)*AT(Coefficient,1,2)[k];    \
18   }                                                             \
19   Output = (uint8) (y16 >> 8);                                  \
20 }                                                               \
21 /**/
```

Prog 2.17– Macros pour le calcul du filtrage

2.4 Conclusion du chapitre

Dans ce chapitre, nous avons exposé les différentes stratégies nous permettant d'automatiser partiellement ou totalement la génération de code. De cette exposition, nous avons conclu que la métaprogrammation via le préprocessur C++ représentait la stratégie la plus à même de respecter les contraintes de l'environnement Catapult-C. Nous avons ensuite défini plusieurs stratégies et macros permettant de manière rapide et extensible de générer plusieurs variantes de filtres en fonctions des paramètres d'entrées.

CHAPITRE 3

FILTRAGE NON RÉCURSIF

Ce chapitre présente l'implantation optimisée de filtres non récursifs. Ce chapitre est découpé en trois parties principales. La première partie présente les optimisations possibles pour un filtre unique, ainsi que les problèmes classiques de codage en virgule fixe. La seconde présente l'optimisation de l'enchaînement de deux filtres. Et la troisième compare l'impact des transformations et optimisations proposées sur des codes s'exécutant sur des processeurs généralistes basse consommation, à la fois en scalaire et en SIMD

$$y(n) = \sum_{k=0}^{k=N-1} b_k x(n-k) \tag{3.1}$$

Les filtres non récursifs à réponse impulsionnelle finie (FIR) sont communs en traitement du signal. Leur forme générale est donnée par l'équation 3.1.

Si la représentation des données et des coefficients en virgule flottante ne pose pas trop de problèmes – à part les risques inhérents d'*absorption* et de *cancellation* lorsque les valeurs manipulées sont d'amplitude très différentes, il faut faire plus attention lors du codage et du calcul en virgule fixe.

3.1 Optimisation de l'implantation d'un filtre FIR3

Dans cette première partie, nous allons nous efforcer d'optimiser un filtre FIR à 3 points (eq. 3.2) en virgule fixe. Pour cela, nous allons maintenant présenter les différentes optimisations possibles ainsi que les problèmes qu'elles résolvent.

Nous faisons l'hypothèse simplificatrice que le codage utilisé est suffisamment précis. Nous ne cherchons pas non plus à optimiser le nombre de bits car comme nous souhaitons faire des comparaisons avec des processeurs généralistes, nous n'avons le choix qu'entre un codage sur 8 bits ou sur 16 bits. Le choix a été de faire les calculs en Q_8 et de choisir des données sur 8 bits (type image) afin que données et coefficients occupent la même place. les coefficients de l'expression "TS" du filtre (Eq. 3.2) sont donc multipliés par 2^8 et le résultat final est divisé par 2^8.

Ce qui est réalisé par un décalage à droite de 8. De plus, afin de réaliser des calculs avec arrondi - et non des calculs avec troncature, un arrondi r égal à la moitié de la division est additionné (soit $2^7 = 128$). Le pseudo-code associé est donné dans l'algorithme 2 avec $r = 128$.

A noter qu'un déroulage total de la boucle sur les coefficients sera systématiquement réalisé (*loop unwinding*) ainsi qu'une mise en registres (*scalarisation*) afin de maîtriser complètement la version initiale du filtre servant de référence dans les comparaisons par rapport à une expression manipulant des cases de tableau (algo. 1).

Pour une étude de l'optimisation du codage en virgule fixe dans le domaine du traitement du signal et des images, voir les travaux de Daniel Ménard et Olivier Sentieys [49], l'outil Fluctuat [17], [28] et l'ANR DEFIS [48].

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) \tag{3.2}$$

Algorithme 1 : filtre FIR3 – version tableau

$Y[0] \leftarrow X[0], Y[1] \leftarrow X[1], r \leftarrow 128$
for $i = 2$ **to** $n-1$ **do**
    $Y[i] \leftarrow (b_0 \times X[i] + b_1 \times X[i-1] + b_2 \times X[i-2] + r)/256$

Algorithme 2 : filtre FIR3 – version *Reg* (mise en registres)

$Y[0] \leftarrow X[0], Y[1] \leftarrow X[1], r \leftarrow 128$
for $i = 2$ **to** $n-1$ **do**
    $x_0 \leftarrow X[i-0]$
    $x_1 \leftarrow X[i-1]$
    $x_2 \leftarrow X[i-2]$
    $y \leftarrow (b_0 \times x_0 + b_1 \times x_1 + b_2 \times x_2 + r)/256$
    $Y[i] \leftarrow y$

3.1.1 Optimisations logicielles

Le principal problème d'optimisation des filtres FIR est leur faible complexité : il y a un MAC (*Multiply-Accumulate*) pour chaque accès mémoire (LOAD). Il en résulte que sur un processeur généraliste, leur implantation naïve est *memory bound* (limitée par la bande passante du système). Ce comportement va se retrouver sur ASIC avec de la mémoire simple port (SP) qui ne permet qu'un seul accès mémoire par cycle (LOAD ou STORE).

La première optimisation logicielle est donc la rotation de registres qui permet de mémoriser dans des registres les valeurs chargées précédemment et utiles pour le calcul courant et les calculs futurs. On obtient ainsi l'algorithme 3 où il n'y a plus qu'un LOAD et un STORE par itération de la boucle. Concernant le filtrage des premières valeurs du signal, le choix fait est de recopier les deux premières valeurs d'entrée sur la sortie.

La seconde optimisation logicielle est le déroulage de boucle (*Loop Unrolling*). Contrairement au cas général, lorsqu'on considère un filtre FIR, il existe des ordres de déroulages optimaux (au sens de la limitation du nombre d'accès mémoire) et qui sont les seuls à permettre

Algorithme 3 : filtre FIR3 – version *Rot*

```
1 x_2 ← X[0], x_1 ← X[1], r ← 128
2 for i = 2 to n − 1 do
3 |   x_0 ← X[i − 0]
4 |   y ← (b_0 × x_0 + b_1 × x_1 + b_2 × x_2 + r)/256
5 |   Y[i] ← y
6 |   [Rotation de registres]
7 |   x_2 ← x_1
8 |   x_1 ← x_0
```

une mise en registre sans aucun accès mémoire redondant. Le plus petit ordre de déroulage de la boucle est égal à la taille du filtre. Les autres ordres de déroulage sont les multiples de l'ordre de déroulage minimum (algo. 4).

A l'époque des tous premiers compilateurs, l'objectif du déroulage de boucle était de passer moins de temps dans le contrôleur de boucle (et ainsi de diminuer d'autant les aléas de branchement) et d'améliorer le flot d'instructions dans le pipeline : passer moins de temps à faire des tests, et plus à faire des calculs. Avec les compilateurs optimisants modernes et les architectures actuelles – dotées d'un grand nombre d'optimisations matérielles – comme l'exécution dans le désordre (*out of order*), les objectifs ont changé. Il s'agit de remplir la fenêtre des instructions *ready* (instructions décodées prêtes à être exécutées par les unités fonctionnelles du processeur) pour maximiser les possibilités de remplir les unités fonctionnelles du processeur.

Dans un pipeline moderne, l'unité de *dispatch* qui se trouve après les étages *fetch* et *decode* est en charge d'aiguiller chaque instruction prête à être exécutée vers son unité fonctionnelle. Afin de limiter les *stalls* du pipeline liés à des dépendances de données entre instructions, l'unité de *dispatch* fonctionne par anticipation : une centaine d'instructions sont récupérées et décodées, celles dont les dépendances sont résolues sont placées dans la fenêtre des instructions *ready*, les autres restent en attente. Lorsqu'un compilateur doit décider de l'ordre de déroulage d'une boucle, il choisi de telle façon que l'ordre de déroulage de la boucle multiplié par la taille de la boucle soit proche de la taille de la fenêtre. Ainsi tant que le processeur exécutera des instructions de la boucle, s'il y a des dépendances entre calculs, il pourra, si nécessaire, lancer des instructions d'un autre corps de boucle.

A noter aussi que la gestion de l'épilogue – pour les cas où la taille du tableau n'est pas égale à l'ordre de déroulage de la boucle – n'est pas traitée par soucis de simplification, afin de garder un pseudo-code petit et lisible. Une façon astucieuse de ne pas avoir à gérer un épilogue est d'écrire un *Duff's device* [23] qui inclu dans le corps de boucle la gestion de l'épilogue.

3.1.2 Optimisations matérielles

Les optimisations matérielles considérées sont l'interfaçage avec une mémoire permettant de faire plus d'accès par cycle. Si l'on se place dans le cas *memory bound*, on peut faire l'hypothèse que soit tous les calculs sont faits en 1 cycle, soit que chaque partie du calcul (une multiplication-addition) est masquée par un accès mémoire. Dans ce cas, la durée d'une itération de la boucle est égale au nombre de LOAD réalisés en série.

Ces valeurs serviront de borne inférieure dans l'analyse des résultats. On ne tient pas compte du STORE car on fait l'hypothèse que les écritures sont faites dans une mémoire diffé-

Algorithme 4 : filtre FIR3 – version LU avec *scalarisation*

```
1  x_2 ← X[0], x_1 ← X[1], r ← 128
2  Y[0] ← x_2, Y[1] ← x_1
3  for i = 2 to n − 1 step 3 do
4      x_0 ← X[i − 0]
5      y_0 ← (b_0 × x_0 + b_1 × x_1 + b_2 × x_2 + r)/256
6      Y[i + 0] ← y_0
7      x_2 ← X[i + 1]
8      y_1 ← (b_0 × x_2 + b_1 × x_0 + b_2 × x_1 + r)/256
9      Y[i + 1] ← y_1
10     x_1 ← X[i + 2]
11     y_2 ← (b_0 × x_1 + b_1 × x_2 + b_2 × x_0 + r)/256
12     Y[i + 2] ← y_2
```

rente de celle utilisée en lecture (modèle producteur-consommateur) et qu'il n'y a pas besoin de synchronisation en écriture.

Pour le cas d'un FIR3 nous avons donc les bornes suivantes :

1. mémoire simple port (SP) : 1 READ par cycle, soit un total de 3 cycles,
2. mémoire simple port *read-write* (SP RW) : 1 READ **et** 1 WRITE par cycle, soit un total de 3 cycles,
3. mémoire double port (DP) : 2 READ par cycle, soit un total de $\lceil 3/2 \rceil = 2$ cycles,
4. mémoire entrelacée (*banked memory*) : avec 3 bancs mémoire entrelacés (mémoire SP) il est possible de faire 3 READ par cycle et donc d'atteindre une cadence de 1 cycle par point.

Nous pouvons faire deux remarques : D'une part, dans le cas de la mémoire double port, afin d'éviter de passer de 1.5 cycle par point à 2 il aurait été possible de faire un déroulage d'ordre 6, (le $ppcm(2,3)$) et ainsi atteindre la valeur optimale de 1.5 cycle par point. D'autre part, concernant la mémoire entrelacée, il est nécessaire de concevoir un automate à états pour adresser – de manière circulaire – les différents bancs, ce qui ajoute de la surface (et de la consommation) à cette solution. Ainsi la valeur qui était dans la case $X[i]$ dans le cas d'une mémoire unique, se trouve dans le banc numéro $(i \mod 3)$ à l'indice $\lfloor i/3 \rfloor$.

3.1.3 Résultats et analyse

Les tableaux 3.1, 3.2, 3.3 et 3.4 présentent respectivement les résultats des configurations Reg+ mémoire SP, Reg + mémoire DP, Rot + mémoire SP et enfin, Reg + 3×SP. Pour chaque configuration, des synthèses ont été réalisées pour des fréquences allant de 100 à 600 MHz, par pas de 100 MHz. Pour chaque fréquence, nous indiquons la surface en μm^2, la puissance totale (somme de la puissance statique et de la puissance dynamique) en μW par point et l'énergie en pJ par point.

Tout d'abord, on peut observer qu'en mode $auto$, pour l'ensemble des 4 tableaux, quelle que soit la fréquence, la surface et la puissance sont toujours minimisées dans ces conditions. Ensuite, comme indiqué précédemment lors du calcul des bornes minimales, les plus petites valeurs de ii permettant la synthèse sont 3 cycles pour la mémoire SP, 2 cycles pour la mémoire DP et 1 cycle pour la version Rot et la version avec 3 mémoires entrelacées.

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	3884	3935	4153	4215	4506	4730	4237
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	4164	4337	4567	4637	4908	5415	4671
$ii = 4$	3923	3995	4270	4312	4717	5237	4409
puissance ($\mu W/point$)							
$auto$	237.4	358.09	523.26	657.76	837.77	1023.98	606.38
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	262.24	424.43	624.28	793.06	1006.43	1300.97	732.24
$ii = 4$	244.99	375.42	576.14	733.19	956.72	1224.55	685.17
énergie ($pJ/point$)							
$auto$	11.884	10.753	12.220	11.520	11.739	11.956	11.679
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	7.888	6.385	6.263	5.967	6.058	6.526	6.515
$ii = 4$	9.816	7.523	7.699	7.348	7.671	8.182	8.040

TABLE 3.1 – FIR3 + mémoire SP : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	4218	4207	4253	4546	4925	5224	4562
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	5146	5376	5818	5868	6456	7076	5957
$ii = 3$	4779	4627	4777	5027	5349	5956	5086
$ii = 4$	4636	4357	4442	4794	5162	5664	4843
puissance ($\mu W/point$)							
$auto$	265.59	397.82	538.83	728.89	941.94	1136.51	668.26
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	326.62	525.52	783.42	992.31	1325.71	1613.48	927.84
$ii = 3$	289.05	473.66	658.65	895.28	1145.86	1488.24	825.12
$ii = 4$	273.97	432.89	591.06	837.08	1079.38	1356.80	761.86
énergie ($pJ/point$)							
$auto$	13.585	12.207	11.023	13.045	13.487	13.560	12.818
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	6.699	5.392	5.361	5.093	5.443	5.521	5.585
$ii = 3$	8.880	7.278	6.749	6.882	7.047	7.627	7.411
$ii = 4$	11.214	8.861	8.068	8.572	8.842	9.262	9.137

TABLE 3.2 – FIR3 + mémoire DP : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	3979	4046	4046	4357	4603	5775	4468
$ii = 1$	4311	4311	4311	5512	5672	6166	5047
$ii = 2$	4487	4487	5299	5406	5636	6492	5301
$ii = 3$	4007	4231	4433	4499	4753	5819	4624
$ii = 4$	3923	4108	4482	4521	4860	5855	4625
puissance ($\mu W/point$)							
$auto$	253.20	384.73	517.90	717.58	892.67	1236.08	667.03
$ii = 1$	223.51	330.49	437.62	850.09	1062.93	1357.68	710.39
$ii = 2$	280.96	434.23	752.30	947.62	1239.38	1413.77	844.71
$ii = 3$	267.08	433.64	639.12	814.48	1027.78	1298.42	746.75
$ii = 4$	256.95	426.05	645.15	815.25	1048.03	1319.56	751.83
énergie ($pJ/point$)							
$auto$	12.969	11.823	10.610	12.863	14.629	12.662	12.592
$ii = 1$	2.248	1.662	1.467	2.142	2.143	2.281	1.990
$ii = 2$	5.641	4.359	5.040	4.761	4.982	4.733	4.919
$ii = 3$	8.039	6.528	6.416	6.133	6.191	6.515	6.637
$ii = 4$	10.306	8.546	8.629	8.178	8.411	8.823	8.816

TABLE 3.3 – FIR3 + *Rot* : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	4656	4704	4916	4725	4904	5790	4949
$ii = 1$	6396	6857	7480	8144	8193	8152	7537
$ii = 2$	5978	6177	6658	6998	7239	7979	6838
$ii = 3$	5421	5549	5761	5867	5977	6987	5927
$ii = 4$	5193	5316	5508	5648	5859	6611	5689
puissance ($\mu W/point$)							
$auto$	288.94	433.84	623.43	748.27	931.70	1249.69	712.65
$ii = 1$	365.36	626.18	987.67	1415.72	1722.35	1927.90	1174.20
$ii = 2$	381.57	599.97	889.76	1160.90	1454.73	1860.73	1057.94
$ii = 3$	346.97	536.85	773.08	991.08	1223.61	1644.35	919.32
$ii = 4$	324.20	507.23	726.98	938.63	1168.13	1523.47	864.77
énergie ($pJ/point$)							
$auto$	14.782	13.315	14.879	13.394	13.342	14.913	14.104
$ii = 1$	3.682	3.158	3.324	3.577	3.482	3.245	3.411
$ii = 2$	7.661	6.026	5.961	5.833	5.847	6.233	6.260
$ii = 3$	10.436	8.076	7.756	7.457	7.366	8.249	8.223
$ii = 4$	12.990	10.164	9.714	9.407	9.366	10.179	10.303

TABLE 3.4 – FIR3 + 3 mémoires SP entrelacées : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

On peut ensuite observer, que pour une valeur de ii donnée, la surface et l'énergie ne varient pas trop avec la fréquence (contrairement, bien sûr, à la puissance). Pour cette raison, nous présenterons par la suite des valeurs moyennes, lorsque les valeurs des différents paramètres à une fréquence donnée n'ont pas d'intérêt remarquable.

Concernant l'analyse de l'énergie, elle est très fortement corrélée à ii. En mode $auto$, l'énergie consommée est très grande – la variation d'énergie est largement supérieure à la variation de surface – et va de 11.679 à 14.104 pJ/point, pour des valeurs ii valant 5 ou 6 cycles. Il est donc important d'obtenir des valeurs de ii les plus petites possibles. Des latences de 1 cycle par point sont possibles, grâce à certaines optimisations logicielles (Rot) ou matérielles (3×SP). Elles ont toutes en commun de limiter le nombre de LOAD à 1 accès par banc mémoire pour une itération de la boucle.

Afin d'avoir une vue plus globale, nous résumons dans le tableau 3.5 l'ensemble de toutes les estimations réalisées. Chaque valeur est une moyenne (lorsque la synthèse a été possible) des valeurs estimées pour des fréquences allant de 100 à 600 MHz par pas de 100 MHz. Deux configurations sont présentées, pour l'ensemble des optimisations logicielles et matérielles :
- *bestS* la configuration associée à la plus petite surface (celle obtenue en mode $auto$),
- *bestE* la configuration associée à la plus petite énergie.
- la valeur de ii associée à *bestE*

mémoire	SP	SR RW	DP	3×SP	SP	SP	SP RW	DP
optimisation	*Reg*	*Reg*	*Reg*	*Reg*	*Rot*	*LU*	*LU*	*LU*
surface(*best S*)	4237	4266	4562	4949	4458	6120	6240	7027
surface(*best E*)	4671	4751	5957	7537	5047	10692	11038	12294
énergie(*best S*)	**11.68**	12.08	12.54	13.80	11.92	28.24	28.49	26.18
énergie(*best E*)	6.52	6.62	5.47	**3.41**	**1.99**	14.07	14.47	10.61
ii(*best E*)	3	3	2	1	1	3	3	2

TABLE 3.5 – FIR3, surface et energie moyenne pour des fréquences de synthèse $\in [100 : 600]$ avec un pas de 100 MHz

Nous pouvons premièrement observer que le déroulage de boucle est inefficace : s'il permet d'atteindre un cpp de 1 (comme SP+Rot ou 3×SP+Reg) sa surface est plus grande, ce qui est normal puisque l'opérateur est dupliqué 3 fois, mais l'énergie associée est elle aussi très élevée.

Concernant l'énergie de *bestE* celle-ci est très corrélée à ii : autour de 6.5 pour $ii = 3$, à 5.42 pour $ii = 2$ et à 3.41 et 1.99 pour $ii = 1$. Ainsi la rotation de registres permet de diviser par $\times\mathbf{5.9}$ l'énergie consommée tandis que la surface n'augmente pas de manière proportionnelle : seulement 19 %.

On peut aussi se poser la question d'un point de vue optimisations matérielles *versus* optimisations logicielles : s'il ne faut faire qu'une optimisation, laquelle est-ce ? Dans le cas de ce filtre, la rotation de registres permet – à la fois – d'obtenir une énergie plus petite mais une surface (5047 μm^2) plus petite que celle de la meilleure optimisation matérielle Reg+3SP (7537 μm^2) et à peine plus grande que celle associé à *bestS*(SP+Reg). Cela milite pour la mixité des équipes de développement *hardware*.

La conclusion de cette première série de mesures est que la rotation de registres a un impact majeur sur l'énergie qui est divisé par $\mathbf{5.9}$. Sur le plan qualitatif, il apparaît que l'énergie minimale est toujours associée à la plus faible valeur de ii. Dans le cadre des systèmes embarqués (ou enfouis) il faut toutefois vérifier que la puissance dissipée reste modérée et compatible avec l'utilisation de l'ASIC synthétisé.

3.2 Optimisation de l'implantation de deux filtres FIR en cascade

3.2.1 Implantation

Dans la précédente partie, nous avons analysé l'impact des optimisations logicielles et matérielles pour l'implantation d'un filtre FIR3. Nous allons maintenant étudier l'implantation de deux filtres FIR en cascade, qui pose comme problème l'interfaçage du cicuit avec la mémoire.

Si l'on reprend le modèle producteur-consommateur (3.1) pour le filtre FIR3, la version de base (Reg) est un opérateur qui consomme 3 entrées pour produire 1 sortie. En dotant cet opérateur d'un état interne (ici un registre à décalage implanté en C sous la forme d'un tableau statique, nous obtenons la version Rot qui consomme 1 entrée pour produire 1 sortie.

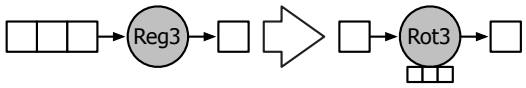

FIGURE 3.1 – modèle *producteur-consommateur*pour le FIR3 : versions Reg et Rot

Le fragment de code 3.1 présente la version Rot du FIR3. Le point important est que la boucle de la ligne 12 qui réalise la rotation des valeurs du registre à décalage est synthétisée de manière optimale, comme le ferait un concepteur VHDL : la sémantique de la boucle est comprise par Catapult-C qui la fait disparaitre pour laisser place en RTL à une copie registre à registre.

Autre point important : dans tous les codes, les boucles sont systématiquement totalement déroulées (*loop unwinding*), ce qui évite de devoir faire des *benchmarks* supplémentaires pour savoir si Catapult-C déroule bien les boucles courtes. L'autre raison est que dans la liste des options d'optimisation de Catapult-C, il faut choisir entre faire du déroulage de boucle (partiel ou total) et du pipeline logiciel. Comme il a été montré dans le chapitre 1, le pipeline logiciel est un facteur essentiel de la performance. Nous avons donc choisi de dérouler manuellement toutes les boucles (sur les coefficients) et de laisser au compilateur le soin de réaliser le pipeline logiciel (l'inverse aurait été très difficile à réaliser).

```
1 uint8 fir3(uint8 x, int input, sint8 b0, sint8 b1, sint8 b2) {
2    static uint8 RD[3]; // registre a decalage
3    uint8 y8;
4    sint16 y16;
5    sint16 round = 1<<7; // 0.5 en virgule fixe Q8
6    int i; int k=3; // taille du FIR
7
```

```
8     if(input<k) {
9        // prologue
10       RD[input] = x;
11       return x;
12    } else {
13       // rotation du registre à decalage
14      for(i=k-1 i>0; i--) { RD[i] = RD[i-1]; }
15      RD[k] = x;
16    }
17
18    // calcul du filtre (version totalement deroulee)
19    y16 = b0 * RD[0] + b1 * RD[1] + b2 * RD[2] + round;
20    y8 = (uint8) (y16>>8);
21    return y8;
22 }
```

Prog 3.1– filtre fir3_rot.c

Dans la suite, nous faisons l'hypothèse que les tableaux X (pour la source), Y (pour la destination) et T (pour le stockage temporaire) ont tous une taille de $N = 1024$ éléments 8 bits (Fig. 3.2). Il y a trois principales façons de cascader deux opérateurs (indépendamment du fait d'utiliser des opérateurs optimisés ou non) :

- cascade d'opérateurs : le premier opérateur consomme toutes les données du tableau X en entrée et produit toutes les données du tableau T en sortie, puis le second opérateur consomme tout le tableau T en entrée pour finalement produire tout le tableau Y (algo. 5).

- pipeline d'opérateurs : le premier opérateur consomme un point du tableau X en entrée et produit 1 point en sortie dans une FIFO, ce point est alors consommé par le second opérateur qui produit un point dans le tableau Y en sortie, puis le premier opérateur consomme un second point et ainsi de suite (algo. 6).

- fusion d'opérateurs : le pipeline d'opérateur est remplacé par un opérateur unique qui est équivalent aux deux précédents (algo. 7).

Algorithme 5 : deux FIR3 filters en cascade avec un tableau temporaire T

for $i = 0$ **to** $n - 1$ **do**
  $x \leftarrow X[i],\ y_1 \leftarrow F_1(x),\ T[i] \leftarrow y_1$
for $i = 0$ **to** $n - 1$ **do**
  $x \leftarrow T[i],\ y_2 \leftarrow F_2(x),\ Y[i] \leftarrow y_2$

Algorithme 6 : deux filtres FIR3 pipelinés

for $i = 0$ **to** $n - 1$ **do**
  $x \leftarrow X[i],\ y_1 \leftarrow F_1(x),\ y_2 \leftarrow F_2(y_1),\ Y[i] \leftarrow y_2$

D'un point de vue compilation le pipeline d'opérateurs correspond à une fusion de boucles (réalisable par un compilateur), tandis que la fusion de filtres correspond véritablement à une

Algorithme 7 : deux filtres FIR3 fusionnés soit un FIR5

for $i = 0$ **to** $n - 1$ **do**
  $x \leftarrow X[i],\ y \leftarrow F_2(F_1(x)),\ Y[i] \leftarrow y$

fusion d'opérateurs (impossible pour un compilateur) telle que nous l'avons décrite précédemment : comme le point clé est d'éviter les accès à des cases de tableaux, il est important de réaliser une *scalarisation* complète et de supprimer l'accès intermédiaire. De plus le tableau interne du nouvel opérateur (implanté sous forme de registre à décalage) doit avoir la bonne taille (ici 5, car ce sont deux filtres identiques de taille 3 qui sont fusionnés). Le pipeline de deux filtres en version Reg aurait été très proche : au lieu d'avoir une FIFO d'une case, il y en aurait eu 3 (la taille du filtre) et il aurait fallu en plus réaliser une rotation des valeurs. Cela aurait été fait avec un *buffer circulaire*.

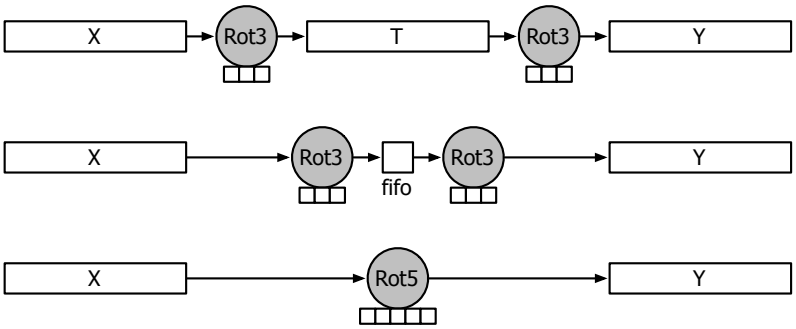

FIGURE 3.2 – trois versions de deux filtres FIR cascadés : deux filtres indépendants avec une mémoire temporaire T (haut), deux filtres avec une FIFO de 1 point (milieu) et deux filtres fusionnés (bas)

3.2.2 Résultats et analyse

La table 3.6 résume l'ensemble des résultats de manière synthétique en faisant la moyenne des surfaces et des énergies pour des fréquences de synthèse allant de 200 à 800 MHz par pas de 200 MHz.

Les chiffres présentés pour la cascade de deux filtres ne prennent pas en compte la surface et la consommation du tableau temporaire T afin d'être cohérents avec les résultats précédents où surfaces et consommations des tableaux en entrée et en sortie n'étaient pas non plus pris en compte. Or une mémoire en 65 nm de 1024 cases 8 bits a une consommation moyenne de 14 $pJ/point$ et une surface d'environ 15000 μm^2, c'est à dire une surface deux fois supérieure à celle du plus grand circuit et une consommation égale au pire cas (mode $auto$). Cela justifie de supprimer les stockages temporaires et de pipeliner les opérateurs via une FIFO, lorsque cela est possible.

Si l'on regarde l'évolution des performances des différentes versions pipelines (configuration *best E*, les optimisations matérielles (mémoire SP, mémoire DP et 3 mémoires SP) permettent d'obtenir des énergies de 9.89 puis 8.11 et enfin 5.55 $pJ/point$ soit un gain d'un facteur

mémoire	SP	SR RW	DP	3×SP	SP	SP	SP RW	DP
optimisation	*Reg*	*Reg*	*Reg*	*Reg*	*Rot*	*LU*	*LU*	*LU*
un filtre FIR3								
surface(*bestS*)	4237	4266	4562	4949	4458	6120	6240	7027
surface(*bestE*)	4671	4751	5957	7537	5047	10692	11038	12294
energie(*bestS*)	**11.68**	12.08	12.54	13.80	11.92	28.24	28.49	26.18
energie(*bestE*)	6.52	6.62	5.47	**3.41**	**1.99**	14.07	14.47	10.61
ii(*bestE*)	3	3	2	1	1	3	3	2
cascade de deux filtres FIR3								
surface(*bestS*)	6393	6482	6542	7498	6574	9913	10155	11155
surface(*bestE*)	8711	9206	10797	10360	7370	15828	16895	16237
energie(*bestS*)	12.88	13.11	13.95	12.86	8.79	24.18	25.05	19.93
energie(*bestE*)	7.56	5.87	4.34	6.69	**2.82**	15.79	16.45	10.89
ii(*best E*)	3	3	2	3	1	3	3	2
pipeline de deux filtres FIR3								
surface(*bestS*)	5888	5943	6207	6543	5715	9625	10039	10400
surface(*bestE*)	7619	7547	8639	12385	9317	18329	19086	20726
energie(*bestS*)	**20.05**	21.78	21.33	23.28	23.11	64.54	65.56	66.98
energie(*bestE*)	9.98	10.03	8.11	**5.55**	**3.65**	21.31	22.08	17.42
ii(*best E*)	3	3	2	1	1	3	3	2
mémoire	SP	SR RW	DP	**5×SP**	SP	SP	SP RW	DP
optimisation	Reg	Reg	Reg	Reg	Rot	LU	LU	LU
fusion de deux filtres FIR3 = un filtre FIR5								
surface(*bestS*)	5563	5619	5683	7121	5990	17968	17563	17330
surface(*bestE*)	6056	6198	7670	12513	8189	26913	28100	30441
energie(*bestS*)	**22.11**	22.72	19.89	27.23	22.01	107.03	107.13	118.48
energie(*bestE*)	13.74	14.05	10.52	**5.59**	**3.19**	48.21	50.98	34.14
ii(*bestE*)	5	5	3	1	1	5	5	3

TABLE 3.6 – FIR moyenne des surfaces et des énergies

$\times 1.8$: le recours à de la mémoire entrelacée est efficace. La rotation de registres fait mieux d'un facteur $\times 1.5$ supplémentaire pour atteindre 3.65 $pJ/point$. Au final, SP+Rot apporte un gain d'un facteur total de $\times$**5.5** pour une augmentation de surface de seulement 58 % par rapport à la version de base.

Si l'on compare la version fusion à la version pipeline, la version SP+Rot gagne sur les deux aspects : une surface plus petite (8189 μm^2 soit seulement $\times 1.47$ fois plus que la plus petite surface possible) et une énergie plus petite (3.19 $pJ/point$ soit $\times 6.9$ fois moins).

On pourra aussi remarquer les performances des Catapult-C quant à la maîtrise de la surface en mode $auto$: 5563 μm^2 pour le FIR5 SP+Reg contre 4237 μm^2 pour le FIR3 SP+Reg. Si l'on analyse maintenant les résultats d'un point de vue de la vitesse (de la valeur de ii), les optimisations matérielles et logicielles permettent toutes les deux d'atteindre la cadence de 1 cycle par point (comme pour le FIR3).

Au final, la configuration SP+Rot est meilleure que $3 \times SP$+Reg. Ainsi les optimisations logicielles l'emportent (encore) sur les optimisations matérielles. Par contre le mieux est l'ennemi du bien : le *Loop Unrolling* n'est jamais efficace : cpp, énergie et surface sont très élevés !

3.3 Evaluation de l'impact des transformations pour les processeurs généralistes

Afin d'évaluer l'impact des transformations logicielles (ici la rotation de registres Rot), en scalaire et en SIMD, plusieurs ensembles de paramètres ont été évalués :

- la spécialisation du filtre : à partir de l'écriture généraliste d'un filtre FIR d'ordre k (eq. 3.1), des versions spécialisés pour chaque taille impaire de filtre ont été écrites : 3, 5 , 7 et 9.

- le jeu d'instructions SIMD : VECx étant plus riche que SSE (il existe des instructions spécialement conçues pour le filtrage), il est possible de soit se limiter aux instructions communes à VECx et à SSE, soit d'utiliser toutes les instructions disponibles (comme la division par une puissance de 2 avec arrondi),

- version Reg et Rot : la version Rot limite à 1 le nombre d'accès à la mémoire contre k pour la version Reg et le code généraliste,

- la latence d'accès à la mémoire : le processeur XP70 n'a pas de cache, mais une mémoire de donnée rapide dite TCDM (*Tightly Coupled Data Memory*), dont la latence peut être fixée à 1 cycle, 2 cycles ou 10 cycles.

La combinaison des différents paramètres donne 21 versions et non 27 car la version généraliste k n'a pas de sens en SIMD. Il est possible d'en écrire une, mais elle nécessiterait des tests (un `switch`) sur la valeur de k pour savoir comment calculer les vecteur non alignés, ce qui la rendrait inefficace et donc contradictoire avec l'objectif intrinsèque de performance du SIMD.

Les versions scalaires sont identiques aux versions C utilisées pour la synthèse automatique, sauf que la rotation de registres (pour la version Rot) se fait dans des registres (*scalarisation* classique) et non dans un tableau local et `static` à la fonction. Par contre les versions SIMD reprennent un *Design Pattern* présenté par Motorola pour Altivec [25] : le calcul de registres vectoriels *non-alignés* pour éviter les rechargements.

Pour que les performances passent à l'échelle en SIMD (une accélération théorique d'un facteur p, avec p le parallélisme d'instructions), il est nécessaire que les instructions SIMD réalisant des opérations arithmétiques (entières ou flottantes) aient une latence proche de leur équivalents scalaires s'il y a des dépendances de données ou qu'elles soient pipelinées (pour en lancer une à chaque cycle).

Le point bloquant à ce stade est que la majorité des algorithmes de TS sont *memory bound* sur la très grande majorité des processeurs RISC généralistes. La raison est que leur intensité arithmétique (le ratio entre le nombre d'opérations de calcul et le nombre d'accès mémoire) est très faible : la latence des accès mémoire ne peut être cachée par la durée des calculs. La plus faible intensité arithmétique est celle du filtre généraliste de taille k : par tour de boucle il est nécessaire de réaliser deux LOAD pour charger le point et le coefficient de filtrage ainsi qu'une multiplication et une accumulation (addition dans un accumulateur), soit un total de $2k+1$ opérations arithmétiques (initialisation de l'accumulateur avec la valeur de l'arrondi et décalage à droite pour la division) et $2k+1$ accès mémoire (1 LOAD unique à la fin) soit une intensité de 1.

Lorsque la taille des filtres est connue, le chargement des coefficients est réalisé en dehors

de la boucle ce qui permet d'obtenir une intensité arithmétique de 2. Cette spécialisation peut néanmoins être contre-efficace car elle nécessite plus de registres (pour stocker la valeur des coefficients) et peut amener le compilateur à générer du *spill code* [1] si le nombre total de registres mise en oeuvre par l'utilisateur dépasse la taille du banc de registres du processeur et que le processeur ne dispose pas de plusieurs jeux de registres. C'est le cas pour le ST XP70 qui ne dispose que de 16 ou 32 registres. Les processeurs Intel et ARM ne sont par contre pas touchés pas cela : ils disposent de plus de registres physiques que de registres qui peuvent être nommés en assembleur et font du *register renaming* lorsque cela est nécessaire.

Le *Design Pattern* utilisé s'attaque donc au problème du nombre d'accès mémoire pour les algorithmes ayant une faible intensité arithmétique. Il consiste à réorganiser les calculs de sorte à toujours avoir un parallélisme maximal (sans instruction de réduction) et à créer des registres vectoriels non-alignés à partir des registres ayant servi pour les LOAD (Fig. 3.3).

Ce schéma correspond au jeu d'instructions VECx : la mémoire est découpée en paquets de 64 bits (interprétés comme 8 blocs 8 bits). Lors de la lecture d'un paquet de 64 bits, l'instruction de chargement convertit à la volée, les 8 blocs de 8 bits en 8 blocs de 16 bits. La phase de chargement est différente en SSE. La mémoire est découpée en paquets de 128 bits (interprétés comme 16 blocs 8 bits), l'instruction de chargement ne fait aucune conversion et le registre contient alors 16 blocs de 8 bits. Il faut ensuite deux instructions de conversion pour obtenir deux registres SIMD contenant chacun 8 blocs de 16 bits ce qui implique que le code de la boucle est dupliqué pour traiter les deux registres. Cela s'apparente à du déroulage de boucle et permet par construction de masquer une partie des latences des instructions, si les deux corps de boucles sont entrelacés (*unroll & jam*) plutôt que positionnés en série. Enfin pour Neon, les deux schémas de chargement sont possibles : 1×64 bits $\rightarrow 1 \times 128$ bits ou 1×128 bits $\rightarrow 2 \times 128$ bits.

A partir de deux chargements mémoire dans des registres vectoriels $v_0 = X[i-0]$ et $v_1 = X[i-1]$, deux registres vectoriels non-alignés u_1 et u_2 sont calculés à partir de v_0 et v_1 grâce à des instructions SIMD. Ce sont `vec_sld` en Altivec, `_mm_alignr_epi8` en SSSE3, `VECx_VSHLUIH` en VECx. A noter que jusqu'à SSE3 inclus, il fallait trois instructions : deux pour faire des décalages sur 128 bits à gauche et à droite (`_mm_slli_si128` et `_mm_srli_si128`) et un OU pour recombiner l'ensemble `_mm_or_si128`).

Ces instructions sont cachées dans un ensemble de macros ($\texttt{left}_1$ et $\texttt{left}_2$) pour rendre le code indépendant de l'extension SIMD utilisée. Après cela, il ne reste plus qu'à réaliser le même calcul qu'en scalaire, en multipliant et accumulant point-à-point les différents registres : $y = h_0 \times v_0 + h_1 \times v_1 + h_2 \times v_2$. Ce *Design Pattern* est particulièrement économe en accès mémoire : il ne faut que deux accès mémoire, tant que la largeur du filtre reste inférieure au cardinal du registre SIMD plus un élément ($k = p + 1$). Avec les instructions VECx : seuls deux LOAD 64 bits suffisent pour les filtres FIR de taille ≤ 9. Avec les instructions SSE et Neon, deux LOAD permettent de traiter en plus les filtres FIR de taille ≤ 17.

Ce *Design Pattern* peut être encore amélioré en le combinant avec la rotation de registres *Rot*. Il n'y a plus qu'un accès mémoire par filtre et – gain supplémentaire par rapport au code scalaire – qu'une seule copie registre-à-registre (contrairement aux versions scalaires qui en nécessite $k-1$). C'est grâce à ces opérations non proportionnelles en nombre (et avec les effets conjugués du fonctionnement du cache) que l'on peut dans certains cas avoir des accélérations

1. sauvegarde en mémoire de la valeur contenue dans un ou des registres

sur-linéaires.

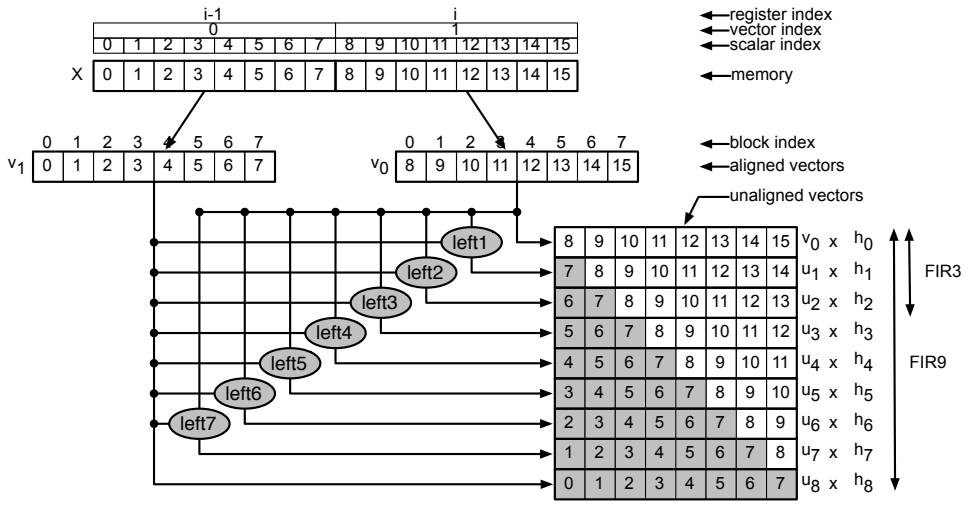

FIGURE 3.3 – calcul de vecteurs non alignés en SIMD pour les filtres FIR3 : en SIMD 128 bits, seuls deux accès mémoire sont nécessaires pour calculer tous les vecteurs non alignés pour des filtres de taille ≤ 9

3.3.1 Résultats et analyse

3.3.1.1 XP70 avec VECx

Le tableau 3.7 regroupe les résultats scalaire et SIMD pour des mémoires TCDM de 1, 2 et 10 cycles. Concernant le code SIMD, soit nous nous limitons à la partie commune avec SSE, soit nous utilisons les instructions supplémentaires disponibles dans VECx

Analyse des versions scalaires : Pouvoir changer la latence de la mémoire de donnée est particulièrement intéressant et pédagogique : lorsque la mémoire est rapide (en 1 cycle) les versions spécialisées (Reg et Rot) l'emportent sur la version généraliste scalaire K. Lorsque la latence double, la version Rot reste toujours plus rapide tandis que la version Reg ne le reste que pour les filtres de grande taille. Puis lorsque la mémoire est lente (10 cycles), la version Reg est toujours plus lente, tandis que la version Rot reste compétitive pour les filtres de petite taille et se trouve elle aussi rattrapée pour les filtres de grande taille.

Pour être efficace *pour l'ensemble des cas* la spécialisation du filtre (qui consiste à faire un déroulage total de la boucle k) n'est efficace que si elle est combinée à la rotation de registres Rot.

		FIR3	FIR5	FIR7	FIR9	2×FIR3/ FIR5
TCDM à 1 cycle						
scalaire	FIR K	17.99	28.95	30.92	32.88	$\times 1.04$
scalaire	FIR Reg	10.99	12.98	15.96	18.94	$\times 1.50$
scalaire	FIR Rot	9.49	9.99	11.98	13.96	$\times 1.80$
SIMD "SSE"	FIR Reg	3.39	4.40	5.40	6.45	$\times 1.54$
SIMD "SSE"	FIR Rot	3.39	4.40	5.40	6.45	$\times 1.54$
SIMD VECx	FIR Reg	2.02	2.77	3.28	3.79	$\times 1.46$
SIMD VECx	FIR Rot	2.14	2.90	3.41	3.92	$\times 1.48$
TCDM à 2 cycles						
scalaire	FIR K	31.98	42.94	44.91	46.88	$\times 1.39$
scalaire	FIR Reg	25.99	39.95	51.89	56.84	$\times 1.25$
scalaire	FIR Rot	22.00	24.00	25.49	26.98	$\times 1.83$
SIMD "SSE"	FIR Reg	6.40	7.41	8.41	9.45	$\times 1.73$
SIMD "SSE"	FIR Rot	6.40	7.41	8.41	9.45	$\times 1.73$
SIMD VECx	FIR Reg	6.52	7.28	7.79	8.30	$\times 1.79$
SIMD VECx	FIR Rot	5.15	5.91	6.42	6.93	$\times 1.74$
TCDM à 10 cycles						
scalaire	FIR K	120.93	133.82	146.67	159.50	$\times 1.81$
scalaire	FIR Reg	97.41	127.81	184.51	239.12	$\times 1.52$
scalaire	FIR Rot	75.47	120.82	146.68	151.09	$\times 1.25$
SIMD "SSE"	FIR Reg	18.74	28.05	37.90	39.91	$\times 1.34$
SIMD "SSE"	FIR Rot	18.19	29.65	37.80	39.91	$\times 1.23$
SIMD VECx	FIR Reg	14.90	21.51	23.27	29.89	$\times 1.39$
SIMD VECx	FIR Rot	13.26	18.27	23.25	28.24	$\times 1.45$

TABLE 3.7 – Evaluation de performance du XP70 en scalaire et en SIMD VECx pour le filtrage non récursif

Analyse des versions SIMD : La Rotation de registre n'a quasiment pas d'impact sur la performance. Elle est même contre-productive (pour TCDM = 1 c). Une hypothèse possible est que le coût des copies registre à registre est trop élevé lorsque le processeur est connecté à une mémoire rapide (une mémoire TCDM de 10 cycles est aussi rapide qu'un cache L2 actuel). Concernant l'impact du jeu d'instructions et la présence d'instructions SIMD spécialisées présentes dans VECx et absentes de SSE, le gain moyen est de l'ordre de $\times 1.6$ pour une mémoire TCDM à 1 cycle et de $\times 1.4$ pour une mémoire TCDM à 10 cycles. Ce qui justifie la présence de ce type d'instructions spécialisées.

Analyse globale : Si l'on compare les versions scalaires aux versions SIMD VECx, les gains vont être très dépendants de la vitesse de la mémoire. Pour une mémoire TCDM de 1 cycle (résultats proche pour une mémoire TCDM à 2 cycles), les gains sont respectivement de $\times 5.4$ et $\times 4.69$ pour les filtres FIR3 et FIR5, tandis que pour une mémoire TCDM de 10 cycles, ils sont respectivement de $\times 7.4$ et $\times 7.0$. La raison est la présence des instructions de mélange (pour le calcul des registres non alignés) qui fait chuter l'accélération. Avec une mémoire à 10 cycles le coût est faible et l'accélération est proche de 8 (le code reste *memory bound*, avec une mémoire à 1 cycle, le coût de ces instructions supplémentaire n'est pas masquable et l'accélération chute.

Le gain maximal (VECx + Rot versus scalaire K) est de $\times 8.1$ pour un FIR3 et de $\times 9.0$ pour un FIR5 avec une mémoire TCDM de 1 cycle et de $\times 9.1$ et $\times 7.3$ pour une mémoire TCDM de 10 cycles. Enfin la fusion d'opérateurs – un FIR5 ou deux FIR3 – est toujours efficace dans tous

les cas : $\times 1.5$.

D'autres tests ont été fait pour évaluer l'impact de la taille du banc de registres qui peut être de 16 ou 32 registres. Il s'avère que pour un banc de 16 registres, il n'y a pas assez de registres disponibles pour les versions *Reg* et *Rot*. Cela se traduit par la présence de beaucoup de *spill code*. De ce fait ces deux versions qui deviennent plus lentes en scalaire que la version généraliste avec la boucle. Par contre il n'y a pas d'impact en SIMD.

3.3.1.2 Cortex-A9 avec Neon

		FIR3	FIR5	FIR7	FIR9	FIR11	2×FIR3/ FIR5
scalaire							
scalaire	FIR *Reg*	15.20	19.21	24.78	31.00	35.85	$\times 1.58$
scalaire	FIR *Rot*	14.16	20.28	26.37	30.09	42.19	$\times 1.40$
accès mémoire SIMD 64 bits							
SIMD "SSE"	FIR *Reg*	2.54	3.27	4.02	4.80	4.81	$\times 1.55$
SIMD "SSE"	FIR *Rot*	2.39	3.14	3.90	4.80	4.56	$\times 1.52$
SIMD Neon	FIR *Reg*	2.41	3.02	3.40	4.02	4.68	$\times 1.60$
SIMD Neon	FIR *Rot*	2.39	3.14	3.39	3.65	4.79	$\times 1.52$
accès mémoire SIMD 128 bits							
SIMD "SSE"	FIR *Reg*	1.83	2.58	3.34	4.16	5.93	$\times 1.42$
SIMD "SSE"	FIR *Rot*	1.70	2.45	3.21	4.10	5.79	$\times 1.39$
SIMD Neon	FIR *Reg*	1.76	2.39	3.01	3.64	4.27	$\times 1.47$
SIMD Neon	FIR *Rot*	1.82	2.45	3.08	3.71	4.27	$\times 1.39$

TABLE 3.8 – Evaluation de performance du Cortex-A9 en scalaire et en SIMD Neon pour le filtrage non récursif

Le tableau 3.8 regroupe les résultats scalaire et SIMD pour des accès mémoire 64 et 128 bits. Concernant le code SIMD, soit nous nous limitons à la partie commune avec SSE, soit nous utilisons les instructions supplémentaires disponibles dans Neon

Analyse des versions scalaires : La rotation de registres n'apporte rien. Elle est même contre-productive. Nous n'avons pas vérifié l'origine du problème.

Analyse des versions SIMD : Là aussi la rotation de registres n'apporte aucun gain. Contrairement à VECx, les instructions Neon spécialisées n'apportent aucun gain, par rapport au "tronc commun" SSE., que ce soit avec des accès mémoire 64 ou 128 bits. Par contre les versions 128 bits sont plus rapides (pour les filtres de petite taille) que les versions 64 bits. Cela est dû à la duplication de code, comme expliqué précédemment (*unroll & jam*). Les gains respectifs pour les filtres FIR3 et FIR5 sont $\times 1.31$ et $\times 1.28$.

Analyse globale : Si l'on compare les versions scalaires aux version SIMD, les gains sont de $\times 6.3$ et $\times 6.4$ avec des accès mémoire 64 bits et de $\times 8.6$ et $\times 8.0$ pour les filtres FIR3 et FIR5 car pour des accès 128 bits, le *unroll & jam* fait qu'il y a deux fois plus d'instructions de calcul qu'il y a de LOAD. Comme précédemment, la fusion d'opérateurs est toujours efficace. En fonction des configurations, le gain oscille entre $\times 1.4$ et $\times 1.6$.

3.3.1.3 Penryn avec SSSE3

		FIR3	FIR5	FIR7	FIR9	FIR11	2×FIR3/ FIR5
scalaire	FIR *Reg*	3.83	6.30	10.91	11.35	14.38	$\times 1.22$
scalaire	FIR *Rot*	3.62	6.10	8.71	9.62	14.55	$\times 1.19$
SIMD SSE	FIR *Reg*	1.06	2.02	2.21	2.76	3.12	$\times 1.05$
SIMD SSE	FIR *Rot*	0.90	1.33	1.85	2.40	2.91	$\times 1.35$

TABLE 3.9 – Evaluation de performance du Penryn en scalaire et en SIMD pour le filtrage non récursif

Analyse des versions scalaires : Comme pour les autres processeurs la rotation de registres n'apporte pas de gain significatif.

Analyse des versions SIMD : Cette fois la rotation de registres apporte un gain pour les filtres de petites tailles : $\times 1.2$ pour le FIR3 et $\times 1.5$ pour le FIR5.

Analyse globale : Les versions SIMD apportent une accélération plus faible que pour les autres processeurs : $\times 4.3$ pour le filtre FIR3 et $\times 4.9$ pour le FIR5. La seule différence entre ces trois architectures SIMD est – à notre connaissance – que sur le Cortex-A9 il n'est possible de lancer une instruction Neon que tous les deux cycles, contrairement au Penryn où une instruction SSE peut être lancé tous les cycles. Ainsi le *unroll & jam* aurait plus d'impact en Neon qu'en SSE ou en VECx.

La fusion d'opérateurs est là aussi toujours efficace, mais avec des gains plus faibles : $\times 1.2$ en scalaire et au mieux $\times 1.35$ en SSE.

3.3.1.4 Analyse globale des trois processeurs

Si on analyse globalement les résultats communs aux des trois architectures SIMD :

- la fusion d'opérateurs – un filtre FIR5 à la place de deux filtre FIR3 – est toujours efficace,
- la rotation de registres *Rot* n'a pas beaucoup d'impact, que ce soit en scalaire ou en SIMD.

La raison est que sur un processeur généraliste, seule une instruction (plus, si le processeur est super-scalaire) peut être exécutée à chaque cycle, contrairement à l'ASIC où en fonction de la longueur des chemins critiques il sera possible de faire plusieurs opérations *simples* du même type en parallèle, grâce à la logique combinaitoire. Ainsi, la version *Rot* qui doit éviter des accès mémoire devient plus lente que la version *Reg* qui recharge les registres, lorsque la latence de la mémoire est faible (XP70 avec TCDM = 1 cycle) car l'ensemble des copies prennent autant de temps voir plus, que les chargements mémoire.

	FIR3	FIR5	FIR7	FIR9	FIR11
gain SIMD					
XP70 TCDM=1	$\times 8.9$	$\times 9.0$	$\times 9.4$	$\times 8.7$	-
XP70 TCDM=10	$\times 9.1$	$\times 7.3$	$\times 6.3$	$\times 5.6$	-
ARM Cortex-A9	$\times 8.6$	$\times 8.0$	$\times 8.2$	$\times 8.3$	$\times 8.4$
Intel Penryn	$\times 4.3$	$\times 4.9$	$\times 5.9$	$\times 4.7$	$\times 4.9$

TABLE 3.10 – Evaluation de performance du Penryn en scalaire et en SIMD pour le filtrage non récursif

3.3.2 ASIC vs GPP : comparaison en temps et en énergie

	cpp		temps (ns/p)		énergie (pJ/p)		ratio temps		ratio énergie	
	FIR3	FIR5	FIR3	FIR5	FIR3	FIR5	FIR3	FIR5	FIR3	FIR5
ASIC (*bestE*)	1	1	1.667	1.667	1.99	3.19	$\times 1$	$\times 1$	$\times 1$	$\times 1$
ASIC ($auto$)	3	5	5.000	8.333	11.68	22.11	$\times 3.00$	$\times 5.0$	$\times 5.9$	$\times 6.9$
XP 70 + VECx	2.21	2.90	4.911	6.444	123	161	$\times 2.95$	$\times 3.87$	$\times 62$	$\times 51$
Cortex A9 + Neon	1.70	2.45	0.708	1.021	580	1225	$\times 0.43$	$\times 0.61$	$\times 427$	$\times 384$
Penryn ULV + SSE	0.90	1.33	0.375	0.554	3750	5542	$\times 0.23$	$\times 0.33$	$\times 1884$	$\times 1737$

TABLE 3.11 – ASIC, XP70, ARM Cortex-A9, Intel SU9300 Penryn ULV

Afin d'avoir des comparaisons équitables il faut corriger les chiffres découlant du cpp de deux manières, car les processeurs Cortex-A9 et Penryn diffèrent du XP70 sur deux points : il sont bi-coeurs et gravé en 45 nm. Ainsi concernant ces deux processeurs :

- le temps de calcul est divisé par deux : $t' = cpp \times Freq/2n$,
- puisque la puissance moyenne dissipée (TDP) concerne le processeur dans la globalité, l'énergie consommée se base sur ce temps divisé par 2 : $E' = t \times P$ auquel il faut appliquer un facteur correctif lié à leur différentes technologies de gravure (65nm contre 45nm). L'ITRS conseille le facteur d'échelle $(65/45)^{1.5} = \times 1.74$.

Par contre, le cpp reste inchangé car il caractérise l'efficacité d'un cœur. Le tableau 3.11 prend en compte ces ajustements.

On peut faire plusieurs observations. La première est que le ratio des vitesses n'est pas proportionnelle à la fréquence (ASIC versus GPP). l'ASIC à 600 MHz est trois plus rapide que le XP70 qui a une fréquence d'horloge proche. Il est deux fois plus lent que le Cortex-A9 qui a une fréquence deux fois plus élevée. Puis la proportionnalité n'est à nouveau plus respectée car le Penryn est deux fois plus rapide que le Cortex-A9.

La seconde est que le processeur XP70 est très efficace énergétiquement, puisqu'il n'est qu'à un facteur $\times 62$ et $\times 51$ de l'ASIC. Si ce dernier n'avait pas bénéficié d'optimisation, les ratios énergétiques seraient tombés à 10.5 et 7.3. Idem d'un point de vue vitesse : grâce à l'extension VECx, il n'est qu'à un facteur 3.87 de l'ASIC.

Enfin si l'on cherche à positionner le Cortex-A9 entre le XP70 et le Penryn, il apparaît que le Cortex-A9 est plus proche du Penryn que du XP70 : il y a un ratio de 10 entre l'énergie

consommé par le Cortex-A9 et celle consommée par le XP70 tandis qu'entre le Cortex-A9 et le Penryn, ce ratio est de 4.5. Alors que d'un point de vue vitesse, il y a environ un ratio d'environ 2 entre ces différents processeurs

3.4 Conclusion du chapitre

	FIR3 SP+Rot		*FIR5 SP+Rot*	
	S (μm^2)	E(pJ/p)	S (μm^2)	E(pJ/p)
$ii = 1$	5047	1.990	8506	3.254
$ii = 2$	5301	4.919	7958	7.017
$ii = 3$	4624	6.637	7426	9.832
$ii = 4$	4625	8.816	7382	12.514
...	...	...	...	...
$auto(ii \geq 6)$	4468	12.592	6544	22.779
gain (max/min)	$\times$1.13	$\times$6.33	$\times$1.30	$\times$7.00

TABLE 3.12 – Surfaces et énergies moyennes des FIR3 et FIR5 pour la configuration SP+*Rot*

Dans ce chapitre, nous avons appliqué plusieurs transformations algorithmiques (fusion d'opérateurs), logicielles (rotation de registres) et matérielle (mémoire multi bancs). Ces optimisations ont aussi été appliquées à des processeurs RISC généralistes, en scalaire et en SIMD afin de comparer leur impact sur les performances (uniquement en *cpp* dans ce cas). Il apparait que la fusion d'opérateurs est toujours efficace, quelle que soit l'architecture cible. Par contre, la rotation de registres n'est efficace que pour l'ASIC. La raison est que Catapult-C est capable d'adapter la surface et la latence d'un opérateur, afin qu'il se termine bien avant la fin d'un cycle d'horloge. Cela permet d'avoir en logique combinatoire, plusieurs opérateurs pouvant être exécutés au même cycle sur l'ASIC (le nombre dépend de la complexité des opérateurs), alors que sur un processeur généraliste, cela est figé et dépend du nombre d'ALU et de FPU du processeur (super-scalaire).

Concernant les performances de Catapult-C, nous avons montré qu'il est possible de proposer d'appliquer des optimisations logicielles au code C et des optimisations matérielles à la gestion mémoire (mémoire multi-bancs et automate de sélection), qui soient "comprises" (à part le déroulage de boucle qui est à éviter) par Catapult-C.

Ce premier chapitre d'évaluation laisse aussi entrevoir un début de stratégie pour l'optimisation sous contrainte : la meilleure configuration énergétique est toujours liée à l'obtention du plus petit ii. Grâce aux optimisations, l'augmentation en surface reste contenue (30%) alors que l'énergie consommée peut être divisée par un facteur 6 ou 7 (table 3.12). Ainsi, en fonction des objectifs antagonistes de surface et d'énergie, l'utilisateur peut choisir toute une gamme de configurations entre les deux configurations extrêmes données par $ii = 1$ et $ii = auto$.

Enfin si l'on compare les performances de l'ASIC aux processeurs généralistes, il obtient des vitesses de traitement proches du plus rapide des trois, tout en ayant une consommation énergétique plus petite de trois ordres de grandeur (Cortex et Penryn). Par contre comparé à celle du XP70 avec extension SIMD VECx, il n'y a plus qu'un facteur 50. Ce micro-contrôleur/microprocesseur est donc un concurrent majeur pour le monde de l'embarqué.

Une partie de ces résultats ainsi que ceux du chapitre suivant ont été publiés à la conférence Patmos en 2013 [71].

Maintenant que les performances de Catapult-C ont été évaluées pour la synthèse automatique de filtres non récursif, qui représente un cas d'optimisation relativement facile, le prochain chapitre va traiter du cas bien plus complexe des filtres récursifs.

CHAPITRE 4

FILTRAGE RÉCURSIF

Ce chapitre présente l'implantation optimisée de filtres récursifs. A l'instar du chapitre précédent, il est aussi découpé en trois parties. La première partie présente les transformations algorithmiques de haut niveau possibles pour un filtre récursif, ainsi que les problèmes classiques de codage et de stabilité numérique en virgule fixe. La seconde partie présente l'optimisation de l'enchainement de deux filtres. Et la troisième compare l'impact des transformations et optimisations proposées sur des codes s'exécutant sur des processeurs généralistes.

4.1 Introduction

$$y(n) = \sum_{k=0}^{k=N-1} b_k x(n-k) + \sum_{k=1}^{k=M-1} a_k y(n-k) \tag{4.1}$$

Les filtres récursifs IIR (*Infinite Impulse Response*) sont particulièrement difficiles à implanter et à optimiser que ce soit en matériel ou en logiciel. La raison est la dépendance de données entre la sortie courante $y(n)$ et les sorties précédentes $y(n-1)$, ..., $y(n-k)$.

Dans ce chapitre, nous allons étudier en détail le filtre récursif IIR12 (eq. 4.2, une entrée en x, deux entrées en y) qui est un filtre très utilisé en traitement du signal et des images car c'est la version simplifiée du célèbre filtre de Canny-Deriche [9, 21, 22]. Ce filtre est dû à Fédérico Garci-Lorca, le doctorant de Didier Demigny [43]. Il a été particulièrement étudié en France, lors de l'action Ardoise du GdR ISIS. Le livre [18] éditée par Didier Demigny et co-écrit par une dizaine d'auteurs présente à la fois la problématique "signal" (stabilité du filtre, nombre de bits pour coder les valeurs et les coefficients) et la problématique "architecture". Les transformations algorithmiques que nous reprenons ici sont celles proposées dans le chapitre 8 sur les RISC [36] et le chapitre 10 sur le DSP VLIW C6x [37].

Le paramètre de lissage du filtre de FGL est α. On lui préfère $\gamma = e^{-\alpha}$ pour avoir une notation plus compacte. L'intervalle de valeurs utiles pour le lissage est $\alpha = 0.5$ pour un lissage fort (équivalent à un filtrage gaussien très large), $\alpha = 0.8$ lissage moyen lorsqu'il y a peu de bruit

dans l'image et $\alpha = 1.0$ lissage faible lorsqu'il y a très peu de bruit dans l'image.

Afin d'avoir des multiplieurs sur un petit nombre de bits, Didier Demigny propose une astuce : coder γ sous forme de puissance de 2 sur trois bits ($\gamma = 2^{-1}\gamma_1 + 2^{-2}\gamma_2 + 2^{-3}\gamma_3$) afin que les multiplications soient remplacées par des décalages. Cette simplification a été utilisée pour porter les filtres de FGL sur FPGA. Nous ne l'avons pas évaluée afin de ne pas trop spécialiser l'étude, pour que les propositions et analyses faites dans ce chapitre puissent servir à d'autres filtres récursifs.

4.2 Transformations algorithmiques : les différentes formes du filtre de FGL

Nous présentons dans cette section trois expressions du filtre de FGL : la forme *Normal* et les formes *Factor* et *Delay*.

La forme standard du filtre est la formée développée dite *Normal* avec 3 multiplications et deux additions (eq. 4.2). Comme pour le filtre FIR du chapitre précédent, il est nécessaire d'ajouter une addition pour le calcul se fasse avec arrondi et un *shift* pour normaliser le résultat du calcul en virgule fixe Q_8 (algo. 8).

Plus encore que pour les filtres non récursifs, il est important de faire une *scalarisation* afin de limiter les accès mémoires pour les versions avec rotation de registres *Rot* et déroulage de boucle LU, car c'est une transformation complexe à réaliser – et les compilateurs n'ont pas forcément d'heuristiques pour les guider dans le choix de l'ordre de déroulage. A l'époque où KAP (KAI Software racheté par Intel) existait, des tests de transformations source à source avaient été réalisés. KAP était capable de trouver l'ordre de déroulage permettant une *scalarisation* totale du code. Une mise en registre quasi parfaite été réalisée. Seul un accès à une case mémoire persistait dans le code.

L'outil PIPS de l'ENSMP/CRI [31] maîtrise parfaitement ces différentes transformations – entièrement *scriptable* – mais il ne possède pas d'heuristique pour trouver l'ordre de déroulage permettant une scalarisation complète.

$$\text{forme } \textit{Normal} : \quad y(n) = (1-\gamma)^2 x(n) + 2\gamma y(n-1) - \gamma^2 y(n-2) \tag{4.2}$$

Algorithme 8 : filtre IIR12 – forme *Normal* version tableau

```
1 r ← 128
2 Y[0] ← X[0]
3 Y[1] ← X[1]
4 for i = 2 to n − 1 do
5     Y[i] ← (b0 × X[i] + a1 × Y[i − 1] + a2 × Y[i − 2] + r)/256
```

En fonction de la latence des opérateurs de multiplications et d'addition, ainsi que de leur nombre (pour les processeurs superscalaires et les processeurs VLIW), il peut être avantageux de développer l'expression et de la factoriser par rapport au paramètre de lissage γ. C'est le cas pour le DSP C6x de Texas Instrument. Comme il est possible via Catapult-C d'avoir plusieurs opérateurs par cycle, il intéressant de voir comment Catapult-C va réagir. On obtient ainsi la forme *Factor* (eq. 4.3, algo. 11).

Algorithme 9 : filtre IIR12 – forme *Normal* version Reg

$r \leftarrow 128$
$Y[0] \leftarrow X[0]$
$Y[1] \leftarrow X[1]$
for $i = 2$ **to** $n - 1$ **do**
  $x_0 \leftarrow X[i - 0]$
  $y_1 \leftarrow Y[i - 1]$
  $y_2 \leftarrow Y[i - 2]$
  $y_0 \leftarrow (b_0 \times x_0 + a_1 \times y_1 + a_2 \times y_2 + r)/256$
  $Y[i] \leftarrow y_0$

Algorithme 10 : filtre IIR12 – forme *Normal* version LU

$r \leftarrow 128$
$y_2 \leftarrow Y[0] \leftarrow X[0]$
$y_1 \leftarrow Y[1] \leftarrow X[1]$
for $i = 2$ **to** $n - 1$ ***step*** 3 **do**
  $x_0 \leftarrow X[i + 0]$
  $y_0 \leftarrow (b_0 \times x_0 + a_1 \times y_1 + a_2 \times y_2 + r)/256$
  $Y[i + 0] \leftarrow y_0$
  $x_1 \leftarrow X[i + 1]$
  $y_2 \leftarrow (b_0 \times x_1 + a_1 \times y_0 + a_2 \times y_1 + r)/256$
  $Y[i + 1] \leftarrow y_2$
  $x_2 \leftarrow X[i + 2]$
  $y_1 \leftarrow (b_0 \times x_2 + a_1 \times y_2 + a_2 \times y_0 + r)/256$
  $Y[i + 2] \leftarrow y_1$

forme *Factor :* $$y(n) = x(n) + 2\gamma\,[y(n-1) - x(n)] - \gamma^2\,[y(n-2) - x(n)] \tag{4.3}$$

Algorithme 11 : filtre IIR12 – forme *Factor*

$r \leftarrow 128$
for $i = 2$ **to** $n - 1$ **do**
  $x_0 \leftarrow X[i - 0],\ y_1 \leftarrow Y[i - 1],\ y_2 \leftarrow Y[i - 2]$
  $y_0 \leftarrow \big(256 \times x_0 + a_1(y_1 - x_0) + a_2(y_2 - x_0) + r\big)/256$
  $Y[i] \leftarrow y_0$

Comme le nombre de multiplications diminue, passant de 3 à 2, cela peut avoir un impact positif sur la surface du circuit, en particulier, si Catapult-C décide de n'implanter qu'un multiplieur et de le réutiliser via des multiplexeurs. La complexité totale du filtre augmente : il y a 4 additions pour l'expression "signal" du filtre (eq. 4.3) tandis que le codage en virgule fixe Q_8 nécessite une addition supplémentaire et un *shift* (pour l'arrondi) ainsi encore qu'un second *shift* pour adapter la dynamique de $x(n)$ (algo. 11). Au lieu d'être multiplié par 1 en virgule flottante, il sera multiplié par 2^8 en virgule fixe, soit un décalage de 8 à gauche.

Il est possible aussi de s'attaquer au problème de la dépendance de données entre $y(n)$ et $y(n-1)$. A défaut de pouvoir la faire disparaitre, on peut relâcher la contrainte entre $y(n)$ et $y(n-1)$. Pour cela on exprime $y(n-1)$ en fonction de $y(n-2)$ et $y(n-3)$ et on injecte cette

expression dans $y(n)$, ce qui a pour effet de faire disparaitre $y(n-1)$ (eq. 4.4). Cette forme est appelée forme *Delay* car elle retarde d'un la dépendance de donnée entre les sorties précédentes $y(n-2)$, $y(n-3)$ et la sortie courante $y(n)$ (algo. 12).

$$\text{forme } \textit{Delay} : \quad y(n) = (1-\gamma)^2 x(n) + 2\gamma(1-\gamma)^2 x(n-1) + 3\gamma^2 y(n-2) - 2\gamma^3 y(n-3) \tag{4.4}$$

Algorithme 12 : filtre IIR12 – forme *Delay*

```
1 r ← 128
2 for i = 2 to n − 1 do
3 |   x0 ← X[i − 0], x1 ← X[i − 1]
4 |   y2 ← Y[i − 2], y3 ← Y[i − 3]
5 |   y0 ← (b0 × x0 + b1 × x1 + a2 × y2 + a3 × y3 + r)/256
6 |   Y[i] ← y0
```

Il a été montré que cette forme est la plus efficace sur le DSP C6x, lors du filtrage de signaux 1D, où il n'est pas possible de traiter en parallèle deux lignes (comme dans une image). Grâce au pipeline logiciel, cette forme s'exécute avec un ii de 2 contre 4 pour la forme *Normal*. Nous verrons dans la troisième partie que cette forme est aussi intéressante pour certains processeurs généralistes.

La complexité des différentes formes du filtre IIR12 est donnée dans la table 4.1. Comme pour les filtres non récursifs du chapitre précédent, nous pouvons appliquer une rotation de registres pour faire diminuer le nombre d'accès mémoire. Ces versions seront notées Reg version avec *scalarisation* et Rot version avec rotation de registres.

version	MUL	ADD	SHIFT	LOAD	STORE	MOVE	IA
version Reg							
forme *Normal*	3	2+1	0+1	3	1	0	$7/4 = 1,75$
forme *Factor*	2	4+1	0+2	3	1	0	$9/4 = 2,25$
forme *Delay*	4	3+1	0+1	4	1	0	$9/5 = 1,80$
version Rot							
forme *Normal*	3	2+1	0+1	1	1	2	$7/2 = 3,5$
forme *Factor*	2	4+1	0+2	1	1	2	$9/2 = 4,5$
forme *Delay*	4	3+1	0+1	1	1	5	$9/2 = 4,5$

TABLE 4.1 – complexité des trois formes du filtre IIR12, avec et sans rotation de registre

4.2.1 Stabilité numérique du filtre

La stabilité numérique des filtres récursifs est problématique, et plus particulièrement encore pour le filtre de FGL. Elle a été étudiée par ses créateurs [19]. L'analyse conclut que si les coefficients peuvent être stockés sur quelques bits, 23 bits sont nécessaires pour stocker les résultats intermédiaires. Or dans tous les calculs, nous supposons que les entrées/sorties sont sur 8 bits et les calculs intermédiaires sont sur 16 bits.

D'un point de vue matériel, il est totalement possible de faire cela pour un ASIC ou un FPGA : le langage VHDL est fait pour cela. Catapult-C propose la même fonctionalité en C, grâce aux classes `ac_int` et `ac_fixed`. Mais si l'on veut avoir un code rapide en SIMD sur un processeur généraliste, il n'est pas possible d'aller au delà de 16 bits pour la multiplication, car il n'y a pas de multiplieur 32 bits en SIMD adaptés au traitement du signal.

Sur les architectures Intel, il existe cependant des multiplieurs plus larges. Il y en a un en SSE4.2 AVX2 et AVX-512 mais ils sont conçus pour des applications de type cryptographie où le produit de deux nombre 32 bits donne un nombre 64 bits. L'utilisation d'une telle instruction entrainerait une perte de parallélisme importante. Il est donc nécessaire de se limiter au cas 16 bits $\times$ 16 bits $\rightarrow$ 16 bits, et mettre en place du stratégie de correction des calculs.

indice i	-1	-2	0	1	2	3	4	5	6	7	8	9	10	11	12
filtre IIR12 : $y(n) = 2\gamma y(n-1) - \gamma^2 y(n-2)$															
somme fausse	255	255	190	125	77	45	25	13	6	2	0	255	254	189	124
somme fausse + arrondi	255	255	190	126	78	46	26	14	7	3	1	0	0	0	0
somme corrigée	255	255	192	129	81	49	29	17	9	4	1	0	255	254	191
somme corrigée + arrondi	255	255	192	129	82	50	30	18	11	7	4	2	1	1	1
filtre IIR11 : $y(n) = \gamma y(n-1)$															
somme fausse	255	255	126	62	30	14	6	2	0	0	0	0	0	0	0
somme fausse + arrondi	255	255	127	63	31	15	7	3	1	0	0	0	0	0	0
somme corrigée	255	255	127	63	31	15	7	3	1	0	0	0	0	0	0
somme corrigée + arrondi	255	255	128	64	32	16	8	4	2	1	0	0	0	0	0

TABLE 4.2 – stabilité des filtres IIR12 et IIR11 pour différentes stratégies de calcul (correction de la somme, arrondi) : valeurs de sortie du filtre pour un signal qui vaut 255 pour $i < 0$ et qui passe à 0 pour $i \geq 0$

Le second problème, propre au filtre de FGL est que le coefficient a_2 est négatif. Soit le cas d'un signal qui vaut 255 pour $i < 0$ et qui passe à 0 pour $i \geq 0$ (tab. 4.2). Comme la somme des coefficients a_1 et a_2 est strictement inférieure à 1, le filtre va produire une suite de nombres décroissants qui finira par atteindre zéro et – en fonction de la manière de faire les calculs – le dépasser. Dans ce cas, il y aura alors un dépassement de capacité par le bas (`underflow`), le codage en complément à 2 donnera un nombre positif.

Il y a plusieurs manière de faire les calculs. Tout d'abord, la conversion par troncature des coefficients flottants en coefficients entiers peut mener à une somme des coefficients fausse (< 255). La somme peut être corrigée en appliquant la correction sur le dernier coefficient ou sur tous les coefficients. Ensuite les calculs peuvent être fait par défaut (troncature) ou par arrondi. Enfin les valeurs peuvent être *clampées* (en ajoutant un test pour empêcher les *underflow* ou *overflow*) sur [0,255].

La table 4.2 présente les résultats de la combinaison des deux premières actions (sans `clamping`). Les valeurs de sortie des filtres sont obtenus pour $\alpha = -ln(2)$ soit $\gamma = 1/2$ qui est la valeur souvent utilisée pour appliqué un lissage important lorsque les images sont très bruitées.

Il apparaît que le calcul par arrondi est suffisant pour ne pas avoir d'instabilité. Sans cela la sortie suivante du filtre aurait été 255 et un cycle de valeurs décroissantes se serait répété alors à l'infini. On peut noter que pour le filtre d'ordre 1 (voir la deuxième partie du chapitre

traitant de la cascade de deux filtres IIR11), le problème n'apparaît pas, car ce filtre n'a que des coefficients positifs.

4.3 Benchmarks

4.3.1 Benchmarks des trois formes sans optimisation

Les résultats des versions Reg des trois formes sont présentés dans la table 4.4. Concernant la forme *Delay*, le nombre d'opérateurs arithmétiques mis en jeu la rend non compétitive sur les trois critères de surface, puissance et énergie. Elle possède toutefois un avantage qui peut être majeur dans certains cas : elle permet d'atteindre des fréquences de synthèse pour des valeurs de ii hors de portée des autres formes. Si l'on compare aux filtres FIR3 (tab. 3.1), il faut dans les deux cas aux moins trois cycles, mémoire SP oblige. Mais à cause de la dépendance de donnée pour les filtres IIR, 4 cycles peuvent être nécessaires pour atteindre des fréquences de 600 MHz. Si l'on vise $ii = 3$, la forme *Normal* est synthétisable jusqu'à 500 MHz, la forme *facteur* atteint seulement 400 MHz, tandis que la forme *retard* atteint les 600 MHz. La forme *retard* atteint même des fréquences encore plus élevées dans d'autres cas qui seront analysés dans une autre section.

La table 4.3 présente la durée d'exécution (en pico-seconde ps) d'un filtre pour les différentes fréquences de synthèse et valeurs de ii. Pour simplifier les calculs, on fera l'hypothèse que la durée est exactement égale au nombre de cycles (la contrainte ii). Si l'on oublie les contraintes de surface et de consommation, la forme *Delay* s'exécute en 5.0 ps, contre 6.0 ps pour la forme *Normal* et 6.7 ps pour la forme *Factor*. En permettant des synthèses à fréquences plus élevées et des ii plus bas que les autres formes, la forme *Delay* est donc la plus rapide.

Si l'on compare les formes *Normal* et *Factor*, la factorisation apporte systématiquement un gain en surface et en énergie. Enfin si l'on compare la forme *Normal* aux FIR3 Reg, on peut observer que la dépendance de données a un effet négatif sur les trois critères : pour un nombre équivalent d'opérateurs d'addition et de multiplication, le filtre récursif nécessite une surface plus grande et consomme plus d'énergie.

Nous allons maintenant appliquer des optimisations logicielles (rotation de registres, déroulage de boucle) et matérielles (mémoire DP). Comme le filtre IIR12 ne réalise que deux accès mémoire au tableau en sortie, il est inutile d'utiliser une mémoire entrelacée à deux bancs, puis que la mémoire double port est disponible.

fréquence (MHz)	400	500	600
cycle (ps)	2.5	2.0	1.67
$ii = 3$	7.5	6.0	5.0
$ii = 4$	10.0	8.0	6.7

TABLE 4.3 – durée d'exécution d'un filtre IIR12 en fonction de la fréquence et de ii

forme	fréq (MHz)	100	200	300	400	500	600
surface (μm^2)							
Normal	$auto$	3976	3987	4034	4321	4683	4933
Normal	$ii = 1$	-	-	-	-	-	-
Normal	$ii = 2$	-	-	-	-	-	-
Normal	$ii = 3$	5385	4612	4697	5192	5842	-
Normal	$ii = 4$	4142	5683	5854	5217	5601	5610
Factor	$auto$	3570	3775	3987	4216	5111	4308
Factor	$ii = 1$	-	-	-	-	-	-
Factor	$ii = 2$	-	-	-	-	-	-
Factor	$ii = 3$	3611	3717	4808	4293	-	-
Factor	$ii = 4$	3590	3830	3931	4911	4635	5003
Delay	$auto$	4642	4642	4819	4886	5335	5759
Delay	$ii = 1$	-	-	-	-	-	-
Delay	$ii = 2$	-	-	-	-	-	-
Delay	$ii = 3$	5045	5045	5123	5691	6277	7117
Delay	$ii = 4$	5322	5273	5562	5736	7182	6526
puissance ($\mu W/point$)							
Normal	$auto$	251.70	383.53	515.51	701.90	907.37	1102.48
Normal	$ii = 1$	-	-	-	-	-	-
Normal	$ii = 2$	-	-	-	-	-	-
Normal	$ii = 3$	361.48	431.54	594.17	884.19	1246.32	-
Normal	$ii = 4$	267.95	590.10	801.96	848.04	1112.23	1352.07
Factor	$auto$	229 ;66	370.16	509.94	685.82	1002.21	957.42
Factor	$ii = 1$	-	-	-	-	-	-
Factor	$ii = 2$	-	-	-	-	-	-
Factor	$ii = 3$	232.11	366.47	617.18	760.19	-	-
Factor	$ii = 4$	230.87	384.18	536.42	813.57	1035.75	1186.13
Delay	$auto$	284.46	433.72	613.92	771.39	978.03	1235.65
Delay	$ii = 1$	-	-	-	-	-	-
Delay	$ii = 2$	-	-	-	-	-	-
Delay	$ii = 3$	306.90	470.41	647.09	958.18	1290.30	1701.92
Delay	$ii = 4$	340.21	523.37	747.49	965.38	1322.12	1501.42
énergie ($pJ/point$)							
Normal	$auto$	12.600	11.517	10.320	12.294	12.714	12.873
Normal	$ii = 1$	-	-	-	-	-	-
Normal	$ii = 2$	-	-	-	-	-	-
Normal	$ii = 3$	10.883	6.488	5.955	6.647	7.505	-
Normal	$ii = 4$	10.736	11.840	10.727	8.495	8.913	9.036
Factor	$auto$	9.200	9.265	10.209	10.297	14.043	9.584
Factor	$ii = 1$	-	-	-	-	-	-
Factor	$ii = 2$	-	-	-	-	-	-
Factor	$ii = 3$	6.979	5.510	6.192	5.716	-	-
Factor	$ii = 4$	9.248	7.697	7.165	8.156	8.300	7.923
Delay	$auto$	17.070	13.014	14.325	15.426	17.601	16.473
Delay	$ii = 1$	-	-	-	-	-	-
Delay	$ii = 2$	-	-	-	-	-	-
Delay	$ii = 3$	9.222	7.068	6.481	7.198	7.762	8.532
Delay	$ii = 4$	13.632	10.485	9.984	9.670	10.595	10.034
surface : ratio *bestE/bestS*							
Normal		$\times 1.04$	$\times 1.16$	$\times 1.16$	$\times 1.20$	$\times 1.25$	$\times 1.14$
Factor		$\times 1.01$	$\times 1.00$	$\times 1.22$	$\times 1.02$	$\times 1.00$	$\times 1.16$
Delay		$\times 1.09$	$\times 1.09$	$\times 1.06$	$\times 1.16$	$\times 1.18$	$\times 1.24$
énergie : ratio *bestS/bestE*							
Normal		$\times 1.17$	$\times 1.78$	$\times 1.73$	$\times 1.85$	$\times 1.69$	$\times 1.42$
Factor		$\times 1.32$	$\times 1.00$	$\times 1.65$	$\times 1.80$	$\times 1.69$	$\times 1.21$
Delay		$\times 1.85$	$\times 1.84$	$\times 2.21$	$\times 2.14$	$\times 2.27$	$\times 1.93$

TABLE 4.4 – IIR12 + mémoire SP + Reg : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

4.3.2 Benchmarks des trois formes : optimisations logicielles ou matérielles

Optimisation logicielle (*Rot*) versus optimisation matérielle (mémoire DP). La première différence est que la mémoire DP ne permet pas d'obtenir des synthèses en 1 cycle, contrairement à l'optimisation équivalente pour les filtres FIR3 ($3\times$ SP). En analysant en détail les résultats produits par Catapult-C, il s'avère que ce n'est pas un problème de vitesse de composant, sans quoi des synthèses auraient pu être possibles à basse fréquence (au moins à 100 ou 200 MHz), mais un problème de non respect de timing dans Catapult-C qui réalise les LOAD au premier cycle et le STORE au second. Du coup la rotation de registres est bien plus efficace car elle permet de réaliser des synthèses avec un $ii = 1$, sauf à haute fréquence (600 MHz) pour la forme *Normal*. Les gains moyens liés à la mémoire DP sont respectivement de $\times 2.14$, $\times 1.86$ et $\times 2.35$ pour les formes *Normal*, *Factor* et *Delay* tandis qu'ils sont de $\times 4.42$, $\times 4.11$ et $\times 4.57$ tout en ayant des augmentations de surface très proches de celles de la configuration avec de la mémoire DP.

Lorsque l'on analyse les différentes configurations minimisant l'énergie (*bestE* obtenue pour la plus petite valeur possible de ii) ou la surface (*bestS* obtenue pour des valeurs de ii oscillant autour de 3, 4 ou 5 cycles), il apparait que *bestE* est plus proche de *bestS* que dans le cas des filtres non récursifs : la surface augmente en moyenne d'un facteur $\times 1.10$ et l'énergie diminue d'un facteur $\times 1.50$ pour les configurations *SP+Reg* et *DP+Reg*, et ce pour les trois formes de filtres. Avec la rotation de registres l'écart se creuse : autour de $\times 1.5$ pour la surface (toutes formes confondues) et de $\times 2.8$ (forme *Normal*) à $\times 3.8$ (formes *Factor* et *Delay*).

Formes *Normal vs Factor vs Delay*. Si l'on compare les trois formes entre elles pour la meilleure optimisation (*Rot*), il apparaît que la forme *Factor* l'emporte sur la forme *Normal* pour la minimisation de l'énergie alors que c'est l'inverse pour la minimisation de la surface : la surface moyenne de la configuration *bestS(Normal + Rot)* est de $3814\mu m^2$ contre $4029\mu m^2$ pour *bestS(Factor + Rot)*. Par contre, la forme *Delay* amène toujours – complexité oblige – à des surfaces plus grandes ou des énergies consommées plus élevées. Son principal intérêt réside dans la possibilité de réaliser des synthèses à des fréquences plus élevées et sera traitée dans une autre section. Il apparait ainsi que le choix de la meilleure version est moins simple que pour les filtres non récursifs.

forme	fréq (MHz)	100	200	300	400	500	600
surface (μm^2)							
Normal	$auto$	3677	3677	3677	3839	4004	4363
Normal	$ii = 1$	5102	5342	5342	5616	6926	-
Normal	$ii = 2$	4265	4265	4630	4943	5295	6214
Normal	$ii = 3$	3720	3720	3720	4026	4278	4542
Normal	$ii = 4$	3559	3559	3559	3939	4127	4590
Factor	$auto$	3666	3842	4007	4092	4243	4660
Factor	$ii = 1$	4559	4559	5070	5745	6041	6968
Factor	$ii = 2$	4066	4066	4401	4784	5051	5064
Factor	$ii = 3$	3701	3701	4534	4467	4607	5353
Factor	$ii = 4$	3586	3586	4303	4154	4399	4852
Delay	$auto$	5176	4494	4568	4760	4999	5701
Delay	$ii = 1$	6658	7336	7749	7823	9915	9625
Delay	$ii = 2$	4824	5510	5499	6270	7139	8931
Delay	$ii = 3$	4798	5322	5505	5501	6261	7856
Delay	$ii = 4$	4792	4658	5072	5421	5349	5963
puissance (μW/point)							
Normal	$auto$	227.51	349.87	471.46	635.31	804.82	999.42
Normal	$ii = 1$	272.57	433.72	578.41	770.37	1255.54	-
Normal	$ii = 2$	245.48	373.95	581.45	770.10	993.49	1430.70
Normal	$ii = 3$	229.35	353.89	475.12	693.11	890.64	1106.94
Normal	$ii = 4$	216.77	333.27	449.06	662.52	838.73	1065.93
Factor	$auto$	233.02	372.22	509.10	638.97	806.43	1074.72
Factor	$ii = 1$	259.77	392.96	631.10	945.72	1220.33	1610.27
Factor	$ii = 2$	274.71	430.43	631.77	819.47	1042.57	1213.17
Factor	$ii = 3$	235.58	363.48	625.87	758.61	952.66	1198.49
Factor	$ii = 4$	223.30	342.73	590.28	698.39	898.37	1180.36
Delay	$auto$	311.17	444.37	593.06	800.54	980.70	1285.46
Delay	$ii = 1$	356.84	632.78	943.61	1185.92	1973.44	2156.45
Delay	$ii = 2$	295.32	557.57	777.97	1120.82	1507.97	1994.47
Delay	$ii = 3$	286.21	485.14	721.72	918.29	1248.69	1638.18
Delay	$ii = 4$	281.24	470.54	719.62	858.41	1133.18	1424.36
énergie (pJ/point)							
Normal	$auto$	9.114	7.008	6.295	7.951	8.058	8.338
Normal	$ii = 1$	2.747	2.186	7.603	1.941	2.533	-
Normal	$ii = 2$	4.926	3.752	3.892	3.866	3.990	4.788
Normal	$ii = 3$	6.896	5.320	4.762	5.212	5.358	5.549
Normal	$ii = 4$	8.684	6.675	5.996	6.637	6.721	7.118
Factor	$auto$	9.334	9.316	8.495	9.594	9.687	10.758
Factor	$ii = 1$	2.618	1.980	2.122	2.390	2.465	2.710
Factor	$ii = 2$	5.516	4.321	4.230	4.117	4.191	4.064
Factor	$ii = 3$	7.084	5.465	6.277	5.706	5.733	6.012
Factor	$ii = 4$	8.945	6.865	7.886	6.998	7.201	7.884
Delay	$auto$	12.456	11.114	11.863	14.009	15.689	12.857
Delay	$ii = 1$	3.596	3.192	3.173	2.991	3.989	3.629
Delay	$ii = 2$	5.924	5.595	5.204	5.626	6.061	6.678
Delay	$ii = 3$	8.598	7.289	7.231	6.901	7.512	8.207
Delay	$ii = 4$	11.258	9.420	9.609	8.592	9.081	9.507
surface : ratio *bestE/bestS*							
Normal		$\times 1.43$	$\times 1.50$	$\times 1.50$	$\times 1.46$	$\times 1.73$	$\times 1.42$
Factor		$\times 1.27$	$\times 1.27$	$\times 1.27$	$\times 1.40$	$\times 1.42$	$\times 1.50$
Delay		$\times 1.39$	$\times 1.63$	$\times 1.70$	$\times 1.64$	$\times 1.98$	$\times 1.69$
énergie : ratio *bestS/bestE*							
Normal		$\times 3.16$	$\times 3.05$	$\times 1.54$	$\times 4.10$	$\times 3.18$	$\times 1.74$
Factor		$\times 3.42$	$\times 3.47$	$\times 4.00$	$\times 4.01$	$\times 3.93$	$\times 3.97$
Delay		$\times 3.13$	$\times 3.48$	$\times 3.74$	$\times 4.68$	$\times 3.93$	$\times 3.54$

TABLE 4.5 – IIR12 + mémoire SP + *Rot* : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

forme	fréq (MHz)	100	200	300	400	500	600
surface (μm^2)							
Normal	$auto$	4517	4551	4802	4608	4806	5622
Normal	$ii = 1$	-	-	-	-	-	-
Normal	$ii = 2$	5410	5496	5496	6029	-	-
Normal	$ii = 3$	5149	5989	6111	5577	6949	-
Normal	$ii = 4$	4647	5175	5175	5537	6559	7117
Factor	$auto$	4189	4394	4583	4659	4779	5063
Factor	$ii = 1$	-	-	-	-	-	-
Factor	$ii = 2$	4731	4731	5070	4992	-	-
Factor	$ii = 3$	4228	4228	4323	4621	5309	5618
Factor	$ii = 4$	4173	4173	5155	4666	5377	5711
Delay	$auto$	5114	5138	5256	5480	5666	6236
Delay	$ii = 1$	-	-	-	-	-	-
Delay	$ii = 2$	7095	7028	7591	7726	9434	9747
Delay	$ii = 3$	5478	6511	6744	6777	7957	7627
Delay	$ii = 4$	5339	5737	5649	7050	7437	8243
puissance (μW/point)							
Normal	$auto$	283.94	431.33	623.68	748.75	931.03	1234.58
Normal	$ii = 1$	-	-	-	-	-	-
Normal	$ii = 2$	295.10	446.89	593.90	864.27	-	-
Normal	$ii = 3$	318.18	575.05	805.94	905.86	1392.10	-
Normal	$ii = 4$	299.47	527.15	712.26	880.82	1285.79	1646.17
Factor	$auto$	275.30	440.56	646.58	757.66	944.75	1165.39
Factor	$ii = 1$	-	-	-	-	-	-
Factor	$ii = 2$	280.54	426.64	650.11	967.85	-	-
Factor	$ii = 3$	281.60	437.18	627.99	894.67	982.61	1396.75
Factor	$ii = 4$	274.72	424.48	705.50	851.96	1010.68	1288.02
Delay	$auto$	316.94	484.79	679.54	859.72	1073.49	1288.40
Delay	$ii = 1$	-	-	-	-	-	-
Delay	$ii = 2$	430.47	647.52	892.19	1252.79	1630.15	1927.25
Delay	$ii = 3$	335.59	624.62	881.27	1119.22	1475.36	1680.12
Delay	$ii = 4$	323.58	551.08	757.58	1141.76	1367.53	1831.84
énergie (pJ/point)							
Normal	$auto$	14.211	12.950	14.563	13.112	13.044	14.414
Normal	$ii = 1$	-	-	-	-	-	-
Normal	$ii = 2$	5.919	4.482	3.971	4.334	-	-
Normal	$ii = 3$	9.564	8.648	8.080	6.807	8.374	-
Normal	$ii = 4$	11.996	10.564	9.515	8.821	10.306	10.996
Factor	$auto$	11.025	11.025	10.787	11.374	11.346	11.663
Factor	$ii = 1$	-	-	-	-	-	-
Factor	$ii = 2$	5.627	4.279	4.347	4.856	-	-
Factor	$ii = 3$	8.465	6.571	6.292	6.723	5.907	7.000
Factor	$ii = 4$	11.002	8.500	9.425	8.532	8.097	8.599
Delay	$auto$	19.016	16.965	18.117	19.337	19.317	17.174
Delay	$ii = 1$	-	-	-	-	-	-
Delay	$ii = 2$	8.639	6.497	5.968	6.285	6.549	6.452
Delay	$ii = 3$	10.081	9.388	8.830	8.411	8.869	8.417
Delay	$ii = 4$	12.950	11.030	10.113	11.432	10.954	12.227
surface : ratio *bestE*/*bestS*							
Normal		$\times 1.20$	$\times 1.21$	$\times 1.14$	$\times 1.31$	$\times 1.45$	$\times 1.27$
Factor		$\times 1.13$	$\times 1.13$	$\times 1.17$	$\times 1.08$	$\times 1.11$	$\times 1.11$
Delay		$\times 1.39$	$\times 1.37$	$\times 1.44$	$\times 1.41$	$\times 1.67$	$\times 1.56$
énergie : ratio *bestS*/*bestE*							
Normal		$\times 2.40$	$\times 2.89$	$\times 3.67$	$\times 3.03$	$\times 1.56$	$\times 1.31$
Factor		$\times 1.96$	$\times 1.99$	$\times 1.45$	$\times 1.00$	$\times 1.00$	$\times 1.23$
Delay		$\times 2.20$	$\times 2.61$	$\times 3.04$	$\times 3.08$	$\times 2.95$	$\times 2.66$

TABLE 4.6 – IIR12 + mémoire DP + Reg : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

Gain total et minimisation de l'énergie. Les meilleures solutions dépendent donc des objectifs recherchés. Leur point commun est que la rotation de registres apporte systématiquement un gain, que ce soit en terme de surface ou d'énergie. Ainsi si c'est la surface qu'il faut minimiser, alors la meilleure solution est la forme *Normal* tandis que si c'est l'énergie qu'il faut minimiser c'est la forme *Factor*. Dans ce cas, le gain total en énergie (comparé par rapport à la version *Normal+Reg* dépasse – en moyenne – $\times 5$ (Tab. 4.7).

fréq (MHz)	100	200	300	400	500	600	moyenne
forme *Normal* + SP+*Reg* versus forme *Factor*+SP+*Rot*							
augmentation de surface	$\times 1.15$	$\times 1.14$	$\times 1.26$	$\times 1.33$	$\times 1.29$	$\times 1.41$	$\times \mathbf{1.26}$
diminution énergie	$\times 4.81$	$\times 5.82$	$\times 4.86$	$\times 5.14$	$\times 5.16$	$\times 4.75$	$\times \mathbf{5.09}$

TABLE 4.7 – IIR12 ratio global entre la version de base et la version la plus optimisée en énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

4.3.2.1 Utilité de la forme *Delay*

Lorsque les premiers tests ont été faits en début de thèse, une version a rapidement été codée pour analyser les performances de Catapult-C (prog. 4.1). Elle était presque identique aux versions développées ensuite (prog. 4.2) : calcul en Q_8, entrée et sortie sur 8 bits, résultats intermédiaires sur 16 bits et coefficients passé en paramètres. L'arrondi était absent (non gênant en dehors du cas particulier présenté avant) et les tableaux étaient globaux au code C (qui pouvait aider Catapult-C à plus optimiser le code, de la même manière qu'un compilateur F77 peut être plus optimisant qu'un compilateur C89 à cause de l'*aliasing de pointeur* présent en C89 et absent en Fortran 77). Cette expérience n'a pu être refaite, car depuis une nouvelle version plus performante de Captapult-C a été installée.

Les résultats des premiers tests avec troncature sont présentés dans la table 4.8 et ceux avec arrondi dans la table 4.9.

```
1 uint8 ii12(sint8 b0, sint8 a1, sint8 a2) {
2    int i;
3
4    Y[0] = Y[1] = X[0]; // prologue
5
6    for(i=2; i>n; i++) {
7        Y[i] = (uint8) ((b0*X[i]+b1*X[i-1]+a2*Y[i-2]+a3*Y[i-3]) >> 8);
8    }
9 }
```

Prog 4.1– filtre iir12_delay_trunc.c

```
1 #define b0 h[0]
2 #define a1 h[1]
3 #define a2 h[2]
4 uint8 ii12(uint8 X[N], uint8 Y[N], sint8 H[3]) {
5    int i;
6    sint16 round = 1<<7;
7    sint8 h[3]; h[0]=H[0]; h[1]=H[1]; h[2]=H[2]; // copie locale
8
9    Y[0] = Y[1] = X[0]; // prologue
```

```
10
11      for(i=2; i>n; i++) {
12          Y[i] = (uint8) ((b0*X[i]+b1*X[i-1]+a2*Y[i-2]+a3*Y[i-3]+r) >> 8);
13      }
14  }
```

Prog 4.2– filtre iir12_delay_round.c

A noter que dans la version avec arrondi (prog. 4.2), les coefficients du filtre sont stockés dans un tableau local à la fonction plutôt que dans trois registres séparés. Cela simplifie la partie génération automatique de code, sans que cela ait d'impact sur performance (vérifié pour l'ensemble des résultats présentés dans cette section).

Pour la version avec troncature (prog. 4.1), l'avantage de la forme *Delay* est manifeste : c'est la seule permettant d'obtenir des synthèse à 800 MHz lorsque les autres doivent s'arrêter à 400 MHz. Les synthèses ont des surfaces plus grandes et consomment plus d'énergie, mais elles sont deux fois plus rapides (puisque toute ont un débit de 1 point/cycle) que les autres. Quand on voit l'emballement que provoque la sortie d'un processeur 10% plus rapide que les précédent (idem pour les smartphones), ce n'est pas négligeable.

Une autre façon de voir les choses est qu'il est possible d'atteindre des fréquences élevées avec une techno ancienne (par rapport à du 32 nm ou du 25) et donc moins chère. Avec la nouvelle version de Catapult-C, l'avantage n'est plus aussi significatif car les formes *Normal* et *Factor* sont synthétisables jusqu'à 600 MHz, mais l'avantage demeure : les synthèses avec la forme *Delay* sont 50% plus rapides que les autres formes.

forme	*Normal*		*Factor*		*Delay*			
Freq (MHz)	200	400	200	400	200	400	**600**	**800**
Area (μm^2)								
auto ii	3780	3762	3635	3931	4469	4830	5517	9963
$ii = 1$	5274	5227	3984	4789	6769	7817	8239	9872
$ii = 2$	4746	4739	3555	4048	5425	6012	6285	8024
$ii = 3$	4163	4557	3492	3824	5204	5585	6496	10354
$ii = 4$	4019	3944	3475	3924	4925	5211	5648	10750
bestE / bestA	×1.40	×1.39	×1.10	×1.22	×1.51	×1.62	×1.49	×1.00
Energy ($pJ/point$)								
auto ii	5.88	9.20	5.66	7.34	9.73	16.02	14.63	20.43
$ii = 1$	2.12	1.65	1.40	1.77	2.02	2.97	2.73	3.02
$ii = 2$	3.15	3.98	3.18	3.53	3.54	5.17	5.28	6.29
$ii = 3$	4.88	5.23	4.50	4.49	5.41	6.97	7.55	8.53
$ii = 4$	5.82	6.86	5.74	6.20	7.41	9.27	9.19	11.87
bestS / bestE	×2.78	×5.58	×4.05	×4.14	×4.81	×5.40	×5.35	×6.77

TABLE 4.8 – résultat pour le filtre IIR12 version *trunc*, pour des fréquences de synthèse allant de 200 à 800 MHz par pas de 200 MHz, avec rotation de registres *Rot*

	Normal form			*Factor* form			*Delay* form		
Freq (MHz)	200	400	600	200	400	600	200	400	600
Area (μm^2)									
auto ii	3677	3839	4363	3842	4092	4660	4494	4760	5701
ii = 1	5342	5616	-	4559	5745	6968	7336	7823	9625
ii = 2	4265	4943	6214	4066	4784	5064	5510	6270	8931
ii = 3	3720	4026	4542	3701	4467	5353	5322	5501	7856
ii = 4	3559	3939	4590	3586	4154	4852	4658	5421	5963
bestE / bestS	$\times$1.50	$\times$1.46	$\times$1.42	$\times$1.27	$\times$1.40	$\times$1.50	$\times$1.63	$\times$1.64	$\times$1.69
Energy ($pJ/point$)									
auto ii	7.01	7.951	8.338	9.316	9.594	10.758	11.114	14.009	12.857
ii = 1	2.19	1.941	-	1.980	2.390	2.710	3.192	2.991	3.629
ii = 2	3.75	3.866	4.788	4.321	4.117	4.064	5.595	5.626	6.678
ii = 3	5.32	5.212	5.549	5.465	5.706	6.012	7.289	6.901	8.207
ii = 4	6.68	6.637	7.118	6.865	6.998	7.884	9.420	8.592	9.507
bestS / bestE	$\times$3.05	$\times$4.10	$\times$1.74	$\times$3.47	$\times$4.01	$\times$3.97	$\times$3.48	$\times$4.68	$\times$3.54

TABLE 4.9 – résultat pour le filtre IIR12 version *round*, pour des fréquences de synthèse allant de 200 à 600 MHz par pas de 200 MHz, avec rotation de registres Rot

4.3.2.2 Tentative de *rescheduling* des calculs

Lorsque l'on observe l'ordre d'exécution des calculs pour la forme *Normal*, mais cela est vrai aussi pour les deux autres formes, Catapult-C réalise des *schedulings* contre-intuitifs. Prenons la version avec *scalarisation* et sans la partie de calcul en virgule fixe pour simplifier les expressions : $y_0 = b_0x_0 + a_1y_1 + a_2y_2$. On s'attend à ce que l'ordre des calculs soit du plus ancien résultat disponible au plus courant, soit pour une expression évaluée de gauche à droite : $y_0 = b_0x_0 + a_2y_2 + a_1y_1$ où la multiplication avec $y(n-1)$ soit la dernière des trois multiplications. Il n'en est rien : elle est exécuté en deuxième. Nous avons ajouté des parenthèses pour contraindre l'ordre d'exécution : $y_0 = (b_0x_0 + a_2y_2) + a_1y_1$, cela n'a servi à rien. En regardant les multiplieurs utilisés, il apparaît que le multiplieur de a_1y_1 a une surface bien plus grande et qu'il est bien plus rapide que les deux autres, ce qui est logique, vu les contraintes plus strictes que cet ordre d'exécution implique.

La conclusion de cette expérience est que, comme tous les compilateurs optimisants "classiques", le nombre de passes, le nombre d'optimisations réalisées et les heuristiques de minimisation globale font que le "raisonnement" et les choix de Catapult-C sont assez difficiles à suivre et parfois contre-intuitifs. Catapult-C n'en demeure pas moins très efficace.

4.4 Optimisation de l'implantation de deux filtres IIR en cascade

4.4.1 Implantations

Comme pour le chapitre précédent sur les filtres FIR, nous nous intéressons maintenant à l'implantation optimisée de deux filtres IIR. Deux points sont toutefois différents. Premièrement, la consommation énergétique du tableau intermédiaire est prise en compte. Deuxièmement combiner deux filtres IIR12 donnerait un filtre IIR14 qui n'a pas grand intérêt car ce filtre est peu courant en traitement d'images. On ne peut donc pas procéder comme pour les filtres FIR en doublant le nombre de coefficients du filtre IIR.

Par contre, le filtre IIR12 étudié est en fait l'application en série de deux fois le même filtre IIR11 (eq. 4.5). Ses coefficients se retrouvent en calculant le carré de la fonction de transfert de ce filtre IIR11 (eq. 4.7). C'est donc ce filtre IIR11 qui sera utilisé pour les synthèses. Comme ce filtre n'a que des coefficients positifs, son implantation *approximative* pose beaucoup de problème, comme cela a été montré en début de chapitre dans la table 4.2.

S'il fallait ajouter un opérateur en sortie de ce filtre, ce sera l'opérateur de gradient de Roberts, comme proposé par Garcia-Lorca, mais cela aurait empêché une comparaison simple car comme cet opérateur nécessite 4 points (2×2) en entrée, il aurait fallu dupliquer deux fois le filtre IIR12 et ajouter un système de registre à décalage pour simuler le voisinage 2×2. Cela aurait beaucoup augmenté la complexité des codes C et la *glue* logicielle à ajouter pour faire fonctionner l'ensemble. Pour toutes ces raisons, nous considérons donc l'enchaînement de deux filtres IIR11 qui, lorsqu'il sont fusionnés donne le filtre IIR12 étudié précédemment.

Comme pour le filtre IIR12, il est possible d'écrire des formes *Factor* (eq. 4.8 et algo. 14) et *Delay* (eq. 4.9 algo 15). Comme pour le filtre II12 la forme *Factor* du filtre IIR11 nécessite une multiplication supplémentaire pour le terme $x(n)$.

$$\text{Forme } \textit{Normal} : y(n) = (1-\gamma)x(n) + \gamma y(n-1) \tag{4.5}$$

Algorithme 13 : filtre IIR11 – forme *Normal*

```
1 r ← 128
2 for i = 2 to n − 1 do
3 |   x_0 ← X[i − 0]
4 |   y_1 ← Y[i − 1]
5 |   y_0 ← (b_0 × x_0 + a_1 × y_1 + r)/256
6 |   Y[i] ← y_0
```

$$H(z) = \frac{1-\gamma}{1-\gamma z^{-1}} \tag{4.6}$$

$$H^2(z) = \frac{(1-\gamma)^2}{1-2\gamma z^{-1}+\gamma^2 z^{-2}} \tag{4.7}$$

Algorithme 14 : filtre IIR11 – forme *Factor*

```
1 r ← 128
2 for i = 2 to n − 1 do
3 |   x0 ← X[i − 0]
4 |   y1 ← Y[i − 1]
5 |   y0 ← (256 × x0 + a1(y1 − x0) + r)/256
6 |   Y[i] ← y0
```

$$\text{Forme } \textit{Factor} : y(n) = x(n) + \gamma\big(y(n-1) - x(n)\big) \tag{4.8}$$

$$\text{Forme } \textit{Delay} : y(n) = (1-\gamma)x(n) + (1-\gamma)\gamma x(n-1) + \gamma^2 y(n-2) \tag{4.9}$$

Algorithme 15 : filtre IIR11 – forme *Delay*

```
1 r ← 128
2 for i = 2 to n − 1 do
3 |   x0 ← X[i − 0]
4 |   x1 ← X[i − 1]
5 |   y2 ← Y[i − 2]
6 |   y0 ← (b0 × x0 + b1 × x1 + a2 × y2 + r)/256
7 |   Y[i] ← y0
```

Le tableau 4.10 donne la complexité des trois formes dans la configuration de base manipulant des registres (*Reg*) et celles optimisées, faisant de la rotation de registres (*Rot*). Comme précédemment le + indique dans le décompte du nombre d'opérations, les opérations supplémentaires liées au calcul en virgule fixe : le *shift* à droite pour la division par 256 et le *shift* à gauche pour la multiplication par 256. Dans tous les cas, l'intensité arithmétique [1] reste faible, sauf pour la dernière configuration *Delay+Rot* qui atteint 3.5

version	MUL	ADD	SHIFT	LOAD	STORE	MOVE	IA
version Reg sans rotation de registres							
forme *Normal*	2	1+1	0+1	2	1	0	$5/3 = 1.67$
forme *Factor*	1	2+1	0+2	2	1	0	$6/3 = 2.00$
forme *Delay*	3	2+1	0+1	3	1	0	$7/4 = 1.75$
version Rot avec rotation de registres							
forme *Normal*	2	1+1	0+1	1	1	1	$5/2 = 2.5$
forme *Factor*	1	2+1	0+2	1	1	1	$6/2 = 3.0$
forme *Delay*	3	2+1	0+1	1	1	3	$7/2 = 3.5$

TABLE 4.10 – complexité des trois formes du filtre IIR12, avec et sans rotation de registres

La table 4.11 présente les surfaces et énergies obtenues pour les formes *Normal* et *Factor*. Les résultats pour la forme *Delay* sont présentés en annexe, car comme il a été dit précédemment

1. le ratio entre le nombre d'opérations arithmétiques et le nombre d'accès mémoire

pour le filtre II12 l'intérêt de la forme *Delay* est d'être synthétisable à haute fréquence (plus hautes que celles des formes *Normal* et *Factor*), au prix d'une plus grande complexité, ce qui se traduit par de mauvaises performances en surface et énergie.

Cette table regroupe l'ensemble des optimisations logicielles et matérielles pour l'application en cascade de deux filtres IIR. Le premier bloc décrit les performance d'un filtre IIR11. Le second bloc (duplication) est simplement le double du premier bloc. Il n'a pas de réalité physique et n'a d'autre but, que de servir de référence à la cascade (troisième bloc) qui prend en compte la consommation énergétique du tableau intermédiaire. Le troisième bloc concerne le pipeline de deux filtres connectés par une FIFO d'une case. Et enfin le dernier bloc (fusion) décrit les performances d'un filtre IIR12.

Pour tous ces blocs, les chiffres indiqués sont les moyennes des surfaces et des énergies qui sont indiquées pour la configuration minimisant la surface (*bestS*) et celle minimisant l'énergie (*bestE*). Pour chaque configuration *bestE*, la valeur de ii est indiquée. Les valeurs en italique indiquent le meilleur résultat (parmi l'ensemble des optimisations logicielles et matérielles) de *bestS* sur une ligne et les valeurs en gras le meilleur résultat pour *bestE*.

La version cascade utilisant un tableau temporaire (1024 cases 8 bits) est bien sûr d'une conception naïve qui est la façon de faire classique en C. Elle est présentée à titre pédagogique (et parce que la consommation et la surface de la mémoire étaient disponibles) pour souligner l'importance de concevoir des opérateurs *flot de données* évitant au maximum les stockages en mémoire. La consommation de la mémoire est plus de deux fois supérieure à celle de la partie calcul du circuit et la surface associée est supérieure à celle du circuit.

Optimisation matérielle *versus* optimisation logicielle : Comme il a été montré, l'utilisation de la mémoire double port DP ne permet pas de synthèse avec un $ii = 1$. L'optimisation Rot l'emporte donc dans tous les cas (pour toutes les combinaisons d'optimisations) : les meilleurs résultats pour *bestS* et *bestE* sont obtenus pour la configuration SP+Rot. A noter que là encore le déroulage de boucle est inefficace, que ce soit avec de la mémoire simple port ou double port.

Cascade, pipeline et fusion : Si l'on compare les résultats pour les colonnes Rot, il apparait que Catapult-C est capable d'optimiser la version pipeline : les surfaces diminuent d'un facteur 1.12 à 1.58 tandis que l'énergie dimimue d'un facteur 1.12 à 1.25. Il n'y a que l'énergie de *bestS* qui augmente. Si l'on compare les résultats des versions pipeline et fusion, la surface de *bestE* diminue pour la forme *Normal* et augmente pour la forme *Factor* et c'est l'inverse pour l'énergie consommée, ce qui est cohérent avec l'ensemble des résultats antérieurs qui ont toujours mis en évidence une évolution inverse de ces deux grandeurs.

Forme *Normal versus* forme *Factor* ? Il apparaît qu'il n'est pas possible d'analyser indépendamment l'enchainement des opérateurs (pipeline ou fusion) de la forme de calcul (*Normal* ou *Factor*). Ce que l'on peut observer globalement, c'est que la meilleure de toutes les combinaisons possibles est le pipeline de la forme *Factor*. Si l'on compare la version de base (bloc duplication, *bestS* + SP + Reg) à la meilleure version sans changer de forme (fusion, *bestE* + SP + Rot), le gain moyen en surface est de $6900/5757 = \times 1.20$ et le gain moyen en énergie est $9.36/3.01 = \times 3.11$.

Si l'on compare maintenant à la meilleure forme *Factor* forme (pipeline, *bestE* + SP + *Rot*), les gains augmentent encore : $6900/4886 = \times\mathbf{1.41}$ pour la surface et $9.36/2.07 = \times\mathbf{4.52}$ pour l'énergie. Ainsi on gagne à la fois en énergie et en surface pour ces deux configurations.

forme	*Normal*					*Factor*				
mémoire	SP	DP	SP	SP	DP	SP	DP	SP	SP	DP
optimisation	*Reg*	*Reg*	*Rot*	*LU*	*LU*	*Reg*	*Reg*	*Rot*	*LU*	*LU*
	un filtre IIR11									
surface(*bestS*)	3450	3258	*3252*	3664	3918	2891	2966	*2836*	3526	3560
surface(*bestE*)	4201	3334	**4353**	5176	6471	2966	3103	**2998**	3885	4620
energie(*bestS*)	7.99	4.80	*5,21*	8.98	9.60	3.88	4.43	*2.85*	8.34	8.30
energie(*bestE*)	4.68	3.66	**1,75**	4.30	3.97	2.97	3.09	**1.30**	3.59	3.14
ii(*bestE*)	2	2	1	2	1	2	2	1	2	1
	duplication d'un filtre IIR11 = deux filtres IIR11 (pour référence)									
surface(*bestS*)	6900	6516	*6504*	7328	7836	5782	5932	*5672*	7052	7120
surface(*bestE*)	8402	6668	**8706**	10352	12942	5932	6206	**5996**	7770	9240
energie(*bestS*)	15.98	9.60	*10.42*	17.95	19.19	7.76	8.86	5.70	16.68	16.61
energie(*bestE*)	9.36	7.32	**3.51**	8.61	7.93	5.94	6.19	2.59	7.17	6.29
ii(*bestE*)	2	2	1	2	1	2	2	1	2	1
	cascade de deux filtres IIR11 avec mémoire intermédiaire									
surface(*bestS*)	19560	31081	*19180*	19063	30942	18682	30215	*18254*	19001	30525
surface(*bestE*)	20595	32360	**22480**	20862	34037	19218	30658	**19536**	19433	32110
energie(*bestS*)	58.70	65.05	*46.51*	94.02	122.30	45.58	53.19	*39.63*	94.21	106.47
energie(*bestE*)	28.24	31.80	**20.26**	50.80	48.52	26.99	30.84	**18.66**	48.93	46.53
ii(*bestE*)	2	2	1	2	1	2	2	1	2	1
	pipeline de deux filtres IIR11									
surface(*bestS*)	4441	4049	*4399*	5105	5350	3741	3657	*3587*	5667	5703
surface(*bestE*)	6343	5112	**7781**	9018	12016	4444	3831	**4886**	6575	7724
energie(*bestS*)	11.84	10.28	*12,18*	22.48	22.26	5.62	6.84	*7.39*	14.72	14.04
energie(*bestE*)	6.47	5.23	**3,13**	7.27	7.14	4.62	4.14	**2.07**	6.05	5.80
ii(*bestE*)	2	2	1	2	1	2	2	1	2	1
	fusion de deux filtres IIR11 = un filtre IIR12									
surface(*bestS*)	4322	4818	*3814*	5917	6358	4063	4522	*4029*	6345	7001
surface(*bestE*)	5016	6083	**5757**	11007	12435	4345	5075	**5490**	12146	11984
energie(*bestS*)	12.05	13.72	*7.62*	28.53	29.04	9.81	7,53	*9.06*	27.78	27.07
energie(*bestE*)	7.73	6.35	**3.01**	13.04	12.11	6.77	5,34	**2.38**	12.85	12.66
ii(*bestE*)	3	2	1	3	2	3	2	1	3	2

TABLE 4.11 – IIR11 moyenne des surfaces et des énergies pour des fréquences de synthèse allant de 100 à 600 MHz par pas de 100 MHz

4.5 Portage sur processeurs généralistes

4.5.1 Comparaison en *cpp*

A cause de la dépendance de données, l'implementation SIMD est complexe. Il existe deux façon de faire. La première, en générale inefficace avec les jeux d'instructions SIMD existants, consiste à réaliser des opérations *horizontales*, c'est à dire, des opérations entre blocs d'un même registre et non des opérations entre registres SIMD. Cela fait perdre beaucoup de parallélisme et conduit à des accélérations peu importantes. Les instructions SIMD ne sont pas faites pour cela.

La seconde consiste à allouer une bande mémoire pour y faire des transpositions par blocs, ce qui est très efficace car le nombre d'instructions est logarithmique (par rapport à la taille des registres SIMD) pour réaliser la transposition et que la bande allouée tient généralement dans le cache L1 ou L2 du processeur. Cette solution a été mise en oeuvre dans [34]. Mais comme indiqué dans l'introduction, nous nous limitons aux HLT ne nécessitant pas d'allocation mémoire. Donc les comparaisons réalisées dans cette section n'utiliseront pas les instructions SIMD disponibles.

Forme	*Normal*		*Factor*		*Delay*	
optimisation	*Reg*	*Rot*	*Reg*	*Rot*	*Reg*	*Rot*
XP70 TCDM=1	9.00	9.50	11.00	10.99	10.99	9.50
XP70 TCDM=10	75.48	81.97	87.47	94.96	107.92	88.49
ARM Cortex A9	16.78	13.17	17.09	18.20	15.58	17.70
Intel Penryn ULV	8.73	6.93	9.13	6.74	7.35	5.35

TABLE 4.12 – Evaluation de performance des processeurs généralistes pour le filtre récursif IIR12 en scalaire, résultats en cycle par point *cpp*

ST XP70 : Sur XP70, la forme *Factor* est toujours plus lente que la forme *Normal*. Avec une mémoire rapide (TCDM=1) L'optimisation *Rot* dégrade les performances ce qui est normale puisqu'un LOAD est aussi rapide qu'une copie registre à registre. Par contre, avec une mémoire lente (TCDM=10) *Rot* apporte un gain à la forme *Delay* qui devient plus rapide. Mais c'est la forme *Normal* qui reste la plus rapide des trois. Comme sur l'architecture XP70, les instructions ont des latences très proches, le fait d'en ajouter (des ADD supplémentaires pour Factor et des ADD et des MUL pour *Delay*) ralentit le code, car ces instructions ne sont plus *masquées* par la durée des accès mémoire. La version la plus rapide est celle avec le moins d'instructions et d'opérations : c'est donc la forme *Normal*. Par contre, le fait que la configuration *Normal+Rot* soit plus lente pour TCDM=10 (que la même version avec *Reg*) est difficilement explicable. En regardant le code assembleur généré, nous n'avons pas trouvé plus de *spill code* que pour les autres versions.

Cortex-A9 : La rotation de registres apporte un gain pour la forme *Normal* ($\times 1.27$) mais est contre-performante pour les deux autres formes. Par contre, c'est la forme *Delay* qui est la plus rapide des trois, et la forme *Factor* la plus lente.

Intel Penryn ULV : La rotation de registres apporte un gain aux trois formes ($\times 1.26$, $\times 1.35$ et $\times 1.37$). Grâce à son architecture plus évoluée (exécution *out-order*, présence de *by-passes*) et bien qu'ayant une fréquence de fonctionnement identique et des latences de caches très proches, le Penryn nécessite moins de cycles que le Cortex-A9 pour exécuter le filtre récursif.

Analyse globale : La première observation que l'on peut faire est que les temps d'exécution des versions sans optimisation ne sont pas proportionnels à la complexité algorithmique, ni au nombre d'accès mémoire, bien que le filtre considéré soit simple. La seconde, qui est une lapalissade est l'importance d'avoir de la mémoire rapide : ainsi avec une mémoire en 1 cycle, le XP70 est environ deux fois plus rapide que le Cortex-A9 (4 cycles de latence du cache L1). Seul le Penryn avec son architecture plus évoluée (3 cycles de latence du cache L1) arrive à faire aussi bien en terme de *cpp*. La même conclusion s'applique aux GPP qu'à la synthèse ASIC : les transformations logicielles (*Rot* et *LU*) et les transformations de haut niveau (forme *Factor* et forme *Delay*) ont un impact important. Mais la meilleure configuration n'est pas toujours celle dictée par le bon sens et certains résultats sont contre-intuitifs. Il reste nécessaire de faire des benchmarks exhaustifs pour bien cerner le problème.

4.5.2 Comparaison en temps et en énergie

Comme expliqué dans le chapitre sur les filtres non récursifs, une comparaison équitable des résultats nécessite de corriger d'une part les temps d'exécution pour les processeurs bi-coeurs et l'énergie consommée pour les processeurs gravées en 45 nm. Comme il n'est pas possible d'avoir un parallélisme de threads (via OpenMP) pour accélérer l'exécution des filtres récursifs, on fait l'hypothèse que le recours à des processeurs bi-coeurs est liés au traitement d'au moins deux flux de données : soit deux signaux indépendants, soit deux lignes d'une image.

	cpp	temps (ns/p)	énergie (pJ/p)	ratio temps	ratio énergie
ASIC (*bestE*)	1	1.667	2	$\times 1$	$\times 1$
ASIC (auto)	5	8.333	12	$\times 5.0$	$\times 5.8$
XP 70	9.00	20.000	500	$\times 12.0$	$\times 242$
Cortex A9 (bi-coeur)	13.17	5.488	11458	$\times 6.6$	$\times 5535$
Penryn ULV (bi-coeur)	5.35	2.229	38788	$\times 2.7$	$\times 18738$

TABLE 4.13 – ASIC, XP70, ARM Cortex-A9, Intel SU9300 Penryn ULV

Globalement, le fait de ne pas avoir codé les filtres en SIMD sur les processeurs généralistes permet à l'ASIC d'avoir des ratios de performances plus élevés que pour les filtres non récursifs. Ainsi le XP70 est 242 fois plus consommateur d'énergie que l'ASIC, mais reste 22 fois plus efficace que le Cortex-A9, alors que le ratio des vitesses est d'environ 4. Quant au Cortex-A9, il maintient un ratio d'environ trois avec les performances énergétiques du Penryn.

4.6 Conclusion du chapitre

Dans ce chapitre, nous avons appliqué des techniques d'optimisations logicielles, matérielles ainsi que des transformations algorithmiques de haut niveau sur des filtres récursifs représentatifs de ceux utilisés en traitement du signal et des images. Nous avons aussi dans un second temps mis en évidence les capacités de Catapult-C à optimiser l'exécution d'une cascade de deux filtres récursifs. et enfin nous avons comparé les performances estimées de l'ASIC à des processeurs généralistes.

L'ensemble des résultats montre clairement que l'implantation optimisée de filtres récursifs reste complexe et que de nombreuses combinaisons doivent être évaluées avant de pouvoir se prononcer sur la *meilleure* configuration (en vitesse ou en énergie). Le *benchmarking* n'est pas une option mais une nécessité.

Mais contrairement au cas des filtres non récursifs où la configuration avec la plus faible consommation énergétique impliquait une augmentation de surface, les optimisations et transformations réalisées sur ces filtres récursifs permettent de gagner sur les deux critères : la surface est réduite d'un facteur $\times\mathbf{1.4}$ tandis que l'énergie consommée est divisée par un facteur $\times\mathbf{4.5}$.

Grâce à ces optimisations, un petit processeur généraliste très optimisé comme le XP70 est plus efficace énergétiquement d'un facteur $\times 22$ et $\times 78$ qu'un processeur embarqué Cortex-A9 et un processeur Penryn ULV, alors que les ratios de temps d'exécution sont dix fois plus petits.

Le prochain chapitre présentera d'autres transformations algorithmiques de haut niveau rendues possibles par la nature bi-dimensionnelle des opérateurs utilisées pour la détection de mouvement.

CHAPITRE 5

DÉTECTION DE MOUVEMENT

Ce chapitre présente l'implantation optimisée d'un algorithme de détection de mouvement et d'un post-traitement morphologique. La première partie présente les transformations faites sur l'algorithme. Plusieurs ré-écritures de l'algorithme de détection qui comporte deux structures de tests sont proposées. L'opérateur de post-traitement morphologique étant une ouverture 3×3 permet de présenter une autre transformation de haut niveau : la réduction par colonne.

5.1 Introduction

Il existe un grand nombre d'algorithmes de détection de mouvement. L'algorithme choisi est Sigma Delta (noté $\Sigma\Delta$ par la suite) car il se prête bien à des implémentations matérielles, qu'il a été porté sur un grand nombre de plateformes (logicielles et matérielles) et que sa complexité reste raisonnable. L'algorithme de détection de mouvement est composé de deux parties : l'algorithme $\Sigma\Delta$ pour réaliser la détection et un post-traitement composé d'opérateurs de morphologie mathématique (ouverture) pour débruiter l'image produite.

5.1.1 Filtrage Sigma-Delta

Le principe de l'algorithme $\Sigma\Delta$ d'Antoine Manzanera [45,46] est d'estimer la moyenne M_t et la variance V_t de la séquence I_t en utilisant des modulations $\Sigma\Delta$. Le seul paramètre de l'algorithme (Algo. 16) est N, le facteur d'amplification de la différence d'images qui s'apparente à un nombre d'écarts types. Si l'écart type V_t est supérieur à N fois Ot (la différence entre l'image courante $I_t$et l'image de référence M_t), alors il y a mouvement en ce pixel.

La force de cet algorithme tient dans sa robustesse et dans sa simplicité : il est moins sensible aux données et au choix des paramètres que les algorithmes à base de moyennes récursives (paramètre d'oubli) tout en ne nécessitant aucun calcul flottant, contrairement à ces dernières. Il est ainsi en rupture avec toute une série d'algorithmes de plus en plus complexes (utilisation de la couleur et de filtrage non linéaire [27,50]), certes plus robustes, mais de moins en moins

Algorithme 16 : algorithme $\Sigma\Delta$ initial

foreach *pixel* x **do** [step #1 : estimation de M_t]
  if $M_{t-1}(x) < I_t(x)$ **then** $M_t(x) \leftarrow M_{t-1}(x) + 1$
  if $M_{t-1}(x) > I_t(x)$ **then** $M_t(x) \leftarrow M_{t-1}(x) - 1$
  otherwise $M_t(x) \leftarrow M_{t-1}(x)$
foreach *pixel* x **do** [step #2 : calcul de O_t]
  $O_t(x) = |M_t(x) - I_t(x)|$
foreach *pixel* x **do** [step #3 : mise à jour de V_t]
  if $V_{t-1}(x) < N \times O_t(x)$ **then** $V_t(x) \leftarrow V_{t-1}(x) + 1$
  if $V_{t-1}(x) > N \times O_t(x)$ **then** $V_t(x) \leftarrow V_{t-1}(x) - 1$
  otherwise $V_t(x) \leftarrow V_{t-1}(x)$
  $V_t(x) \leftarrow max(min(V_t(x), V_{max}), V_{min})$
foreach *pixel* x **do** [step #4 : estimation de $\hat{E}_t$]
  if $O_t(x) < V_t(x)$ **then** $\hat{E}_t(x) \leftarrow 0$ **else** $\hat{E}_t(x) \leftarrow 1$

rapides.

En 2007-2008, une campagne de *benchmarking* a été menée entre l'ENSTA, l'IEF et le LISIF pour faire une comparaison croisée de trois algorithmes de détection de mouvement (différence d'images, $\Sigma\Delta$, relaxation markovienne) et des architectures adaptées à leur implantation, (rétine *Pvlsar34*, PowerPC G4 Altivec et Maille Associative). Les benchmarks ont entre autres mis en évidence que sur le plan architectural, la simplicité de $\Sigma\Delta$ est aussi son point faible. Avec des accès mémoire dans plusieurs tableaux et peu de calculs par point, cet algorithme est principalement limité par la bande passante de l'architecture.

Depuis l'algorithme initial, différentes évolutions ont été réalisées. On peut citer les variantes suivantes qui visent à améliorer le temps de calcul ou la robustesse :

- avec mise à jour conditionnelle, fonction de l'état précédent [20, 65],
- avec une fréquence de mise à jour liée à une loi de Zipf [44],
- une version multi-modes sur FPGA [1],
- une implantation temps réel sur processeur ARM 9 [63],
- une version hiérarchique (bi-niveaux) [40].

La version hiérarchique est la synthèse des différentes versions issues de la coopération entre l'ENSTA et l'IEF : Antoine Manzanera a fourni la loi de Zipf pour la fréquence de mise à jour et Lionel Lacassagne a apporté la mise à jour conditionnelle et le modèle hiérarchique à deux niveaux.

En plus de ces algorithmes visant spécifiquement les architectures numériques classiques, un algorithme à base de double moyennes récursives (Algo. 17) spécialement conçu pour les architectures analogiques a été développé par Antoine Dupret à l'IEF et Arnaud Verdant au CEA Leti [67]. Cet algorithme (RA2) vise les très faibles consommations, via un système d'adaptation de la résolution et des calculs au plus près du capteur [68]. Le système de codage analogique de l'information – à base de capacités commutées – est très proche d'un codage "continu" avec des nombres réels et permet l'utilisation efficace de division sans risque de perte de précision, contrairement aux architectures numériques.

Algorithme 17 : RA2

foreach *pixel* x **do** [step #1 : estimation de $M_{1,t}$]
  $M_{1,t}(x) \leftarrow M_{1,t-1}(x) + \alpha_1[I_t(x) - M_{1,t-1}(x)]$ avec $\alpha_1 = 1/2^2$
foreach *pixel* x **do** [step #1' : $estimationde M_{2,t}$]
  $M_{2,t}(x) \leftarrow M_{2,t-1}(x) + \alpha_2[I_t(x) - M_{2,t-1}(x)]$ avec $\alpha_2 = 1/2^4$
foreach *pixel* x **do** [step #2 : calcul de O_t]
  $O_t(x) \leftarrow |M_{1,t}(x) - M_{2,t}(x)|$
foreach *pixel* x **do** [step #3 : calcul de T_t]
  $T_t(x) \leftarrow T_{t-1}(x) + \alpha_T[O_t(x) - T_{t-1}(x)]$ avec $\alpha_T = 1/2^6$
  if $T_t(x) < T_{min}$ **then** $T_t(x) \leftarrow T_{min}$
foreach *pixel* x **do** [step #4 : estimation de $\hat{E}_t$]
  if $O_t(x) > kT_t(x)$ **then** $\hat{E}_t(x) \leftarrow 1$ **else** $\hat{E}_t(x) \leftarrow 0$ avec $k \in [1.5, 2]$

Les résultats des variantes de $\Sigma\Delta$ ont été comparés du point de vue de la robustesse à RA2 et à un algorithme d'estimation gaussienne [57] *mono mode*. Ces algorithmes ont été choisis car ils sont représentatifs des méthodes *mono-modes* les plus sophistiquées (ce qui exclut les méthodes multi-modes comme les *mixtures de gaussiennes* ou les *estimations de densité de noyaux*). Une analyse comparative de ces méthodes est disponible dans [53].

La robustesse de ces différents algorithmes a été estimée grâce à une analyse ROC, en utilisant le coefficient de corrélation MCC de Mathheus (Eq. 5.1). Soient VP, VN, FP et FN respectivement le nombre de Vrais Positifs, de Vrais Négatifs, de Faux Positifs et de Faux Négatifs. Un MCC de 0 signifie une mauvaise segmentation tandis qu'un MCC de $+1$ signifie une segmentation parfaite.

$$MCC = \frac{VP \times VN - FP \times FN}{\sqrt{(VP + FP)(VP + VN)(VN + FP)(VN + FN)}} \tag{5.1}$$

algorithmes	#38	#91	#170	#251	moyenne
*MCC **sans** post-traitement morphologique*					
estimation gaussienne	0.275	0.310	0.262	0.263	0.277
RA2	0.325	0.572	0.577	0.390	0.466
$\Sigma\Delta$	0.495	0.347	0.169	0.141	0.288
$\Sigma\Delta$ + loi de Zipf	0.676	0.600	0.366	0.308	0.487
$\Sigma\Delta$ conditionnel	0.424	0.533	0.555	0.590	0.526
$\Sigma\Delta$ hiérarchique conditionnel + loi de Zipf	0.644	0.663	0.468	0.415	0.548
*MCC **avec** post-traitement morphologique*					
estimation gaussienne	0.536	0.706	0.584	0.518	0.586
RA2	0.359	0.611	0.647	0.422	0.510
$\Sigma\Delta$	0.811	0.657	0.372	0.298	0.554
$\Sigma\Delta$ + loi de Zipf	0.830	0.728	0.547	0.449	0.640
$\Sigma\Delta$ conditionnel	0.511	0.666	0.769	0.756	0.633
$\Sigma\Delta$ hiérarchique conditionnel + loi de Zipf	0.816	0.827	0.686	0.582	0.699

TABLE 5.1 – résultats : MCC pour l'estimation gaussienne, RA2 et 4 variantes de $\Sigma\Delta$, *sans* et *avec* post-traitement morphologique, séquence *hall*, images 38, 91, 170 et 251

En mono-résolution, les meilleurs résultats sont obtenus pour $\Sigma\Delta$ avec une loi de Zipf. En multi-résolution, les meilleurs résultats sont obtenus en combinant une estimation conditionnelle et une loi de Zipf. Que ce soit avec ou sans post-traitement morphologique (ouverture 3×3), les deux dernières évolutions de $\Sigma\Delta$ sont plus robustes que l'estimation gaussienne et la double moyenne récursive RA2. A noter que ces deux algorithmes nécessitent des calculs en nombres flottants (la précision en virgule fixe étant insuffisante dans certains cas) contrairement à l'ensemble des algorithmes $\Sigma\Delta$ qui ne requièrent qu'un codage et des calculs sur 8 bits.

Comme on peut le constater, visuellement (Fig. 5.1 et 5.2) et numériquement (Tab. 5.1), les résultats de tous les algorithmes – *avec* post-traitement – sont en progression, ce qui met en évidence le rôle très important des opérateurs morphologiques utilisés. Actuellement c'est une ouverture 3×3 qui est utilisée pour enlever des pixels de bruit plutôt qu'une fermeture qui permettrait d'ajouter de pixels et de remplir les composantes connexes, mais qui risquerait d'augmenter le bruit.

5.1.2 Post-traitement morphologique

L'opérateur de post-traitement est une ouverture morphologique binaire avec un élément structurant 3×3. C'est la combinaison d'une érosion binaire puis d'une dilatation binaire. L'érosion et la dilatation binaires s'implantent efficacement sur toutes les architectures contrairement aux versions en niveaux de gris qui peuvent poser problème. En niveaux de gris les opérateurs mathématiques utilisés sont min et max qui nécessitent des tests (via des `if-then-else`) et donc créent des ruptures d'exécution dans le pipeline. En binaire, ce sont les opérateurs booléens AND et OR qui sont utilisés. Il ne créent pas de rupture (en fonction de l'architecture – ARM Cortex et DSP C6x – les opérateurs min et max n'en créent pas non plus) et on une latence très faible (souvent 1 cycle) de par leur simplicité.

L'élément structurant ES a été choisi carré, car ainsi il est décomposable en deux éléments structurants 1D (Eq. 5.2), ce qui permettra d'appliquer des transformations de haut niveau. Cela n'aurait pas été possible, ou beaucoup plus compliqué si l'élément n'avait pas été décomposable.

$$ES_{3\times3} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \tag{5.2}$$

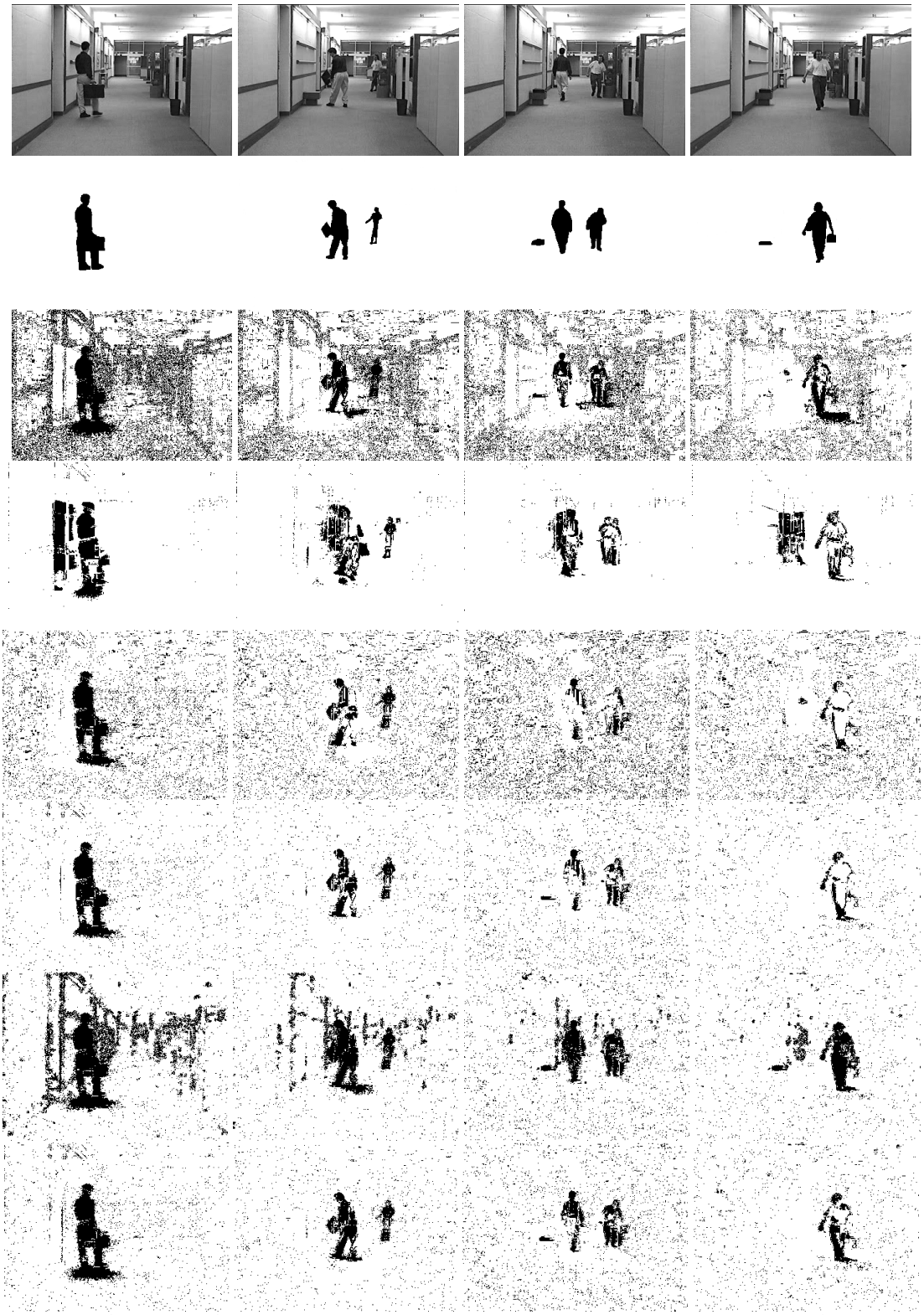

FIGURE 5.1 – détection de mouvement **sans** post-traitement : 1 séquence originale 2 vérité terrain, 3 estimation gaussienne, 4 RA2, 5 $\Sigma\Delta$ basique, 6 $\Sigma\Delta$+Zipf, 7 *Conditional* $\Sigma\Delta$, 8 *Hierarchical* $\Sigma\Delta$. Résultats issus de [39].

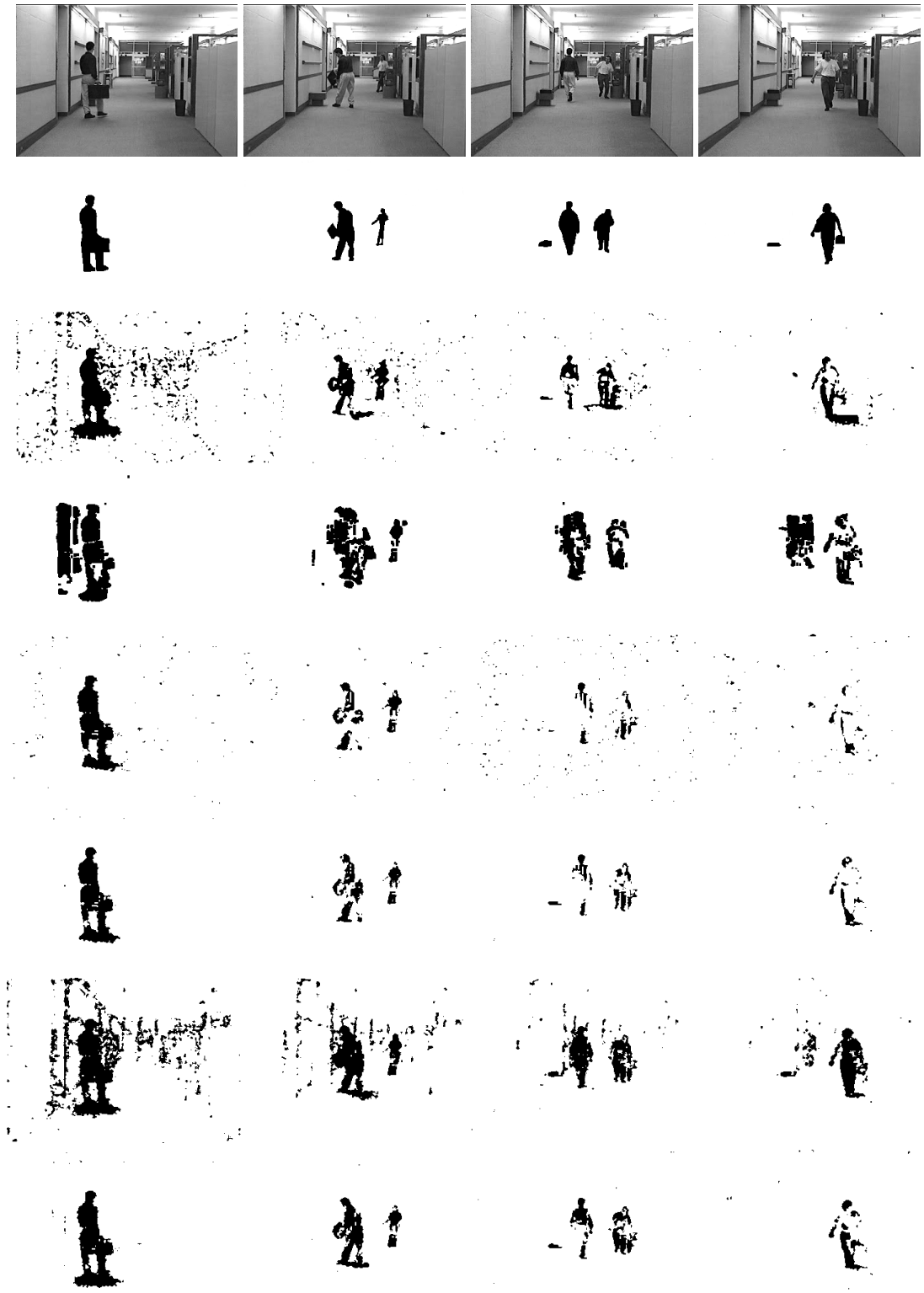

FIGURE 5.2 – détection de mouvement **avec** post-traitement : 1 séquence originale 2 vérité terrain, 3 estimation gaussienne, 4 RA2, 5 $\Sigma\Delta$ basique, 6 $\Sigma\Delta$+Zipf, 7 *Conditional* $\Sigma\Delta$ 8 *Hierarchical* $\Sigma\Delta$. Résultats issus de [39].

5.2 Optimisations et transformations de haut niveau

5.2.1 Opérateur morphologique

$$\text{voisinage } 3 \times 3 \text{ de points : } \begin{bmatrix} a_0 & b_0 & c_0 \\ a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{bmatrix} \tag{5.3}$$

Comme les opérateurs mathématiques sont associatifs et ont la même complexité, nous allons nous concentrer sur l'implantation d'un seul opérateur. Soit OP cet opérateur (noté $\oplus$ dans les différents algorithmes) et soient a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1 et c_2 les points de l'image coïncidant avec l'élément structurant à la position courante (Eq. 5.3). L'implantation naïve de l'opérateur d'érosion ou de la dilation est la version Reg donnée par l'algorithme 18.

Algorithme 18 : version Reg : opérateur morphologique 2D avec un balayage d'image avec utilisation explicite des registres (*scalarisation*)

```
1 for i = 1 to n − 1 do
2     for j = 1 to n − 1 do
3         a0 ← X(i − 1, j − 1), b0 ← X(i − 1, j), c0 ← X(i − 1, j + 1)
4         a1 ← X(i, j − 1), b1 ← X(i, j), c1 ← X(i, j + 1)
5         a2 ← X(i + 1, j − 1), b2 ← X(i + 1, j), c2 ← X(i + 1, j + 1)
6         r ← a0 ⊕ b0 ⊕ c0 ⊕ a1 ⊕ b1 ⊕ c1 ⊕ a2 ⊕ b2 ⊕ c2
7         Y(i, j) ← r
```

Comme pour les filtres 1D, il y a recouvrement des points chargés entre deux itérations de l'opérateur morphologique. Nous pouvons réaliser une rotation de registres Rot_1 (Algo. 19) pour limiter les accès mémoire qui passent ainsi de 9 à 3. Il faut par contre réaliser 6 copies registre à registre pour pour la rotation, ce qui peut être limitant. A noter que la complexité algorithmique ne change pas : il y a toujours 8 opérateurs OP.

En prenant en compte les propriétés mathématiques de l'opérateur morphologique, il est possible de décomposer l'élément structurant 2D en deux éléments structurants 1D (eq. 5.2) et ainsi, de diminuer la complexité algorithmique à 4 OP, contre 8 OP dans la version basique, c'est la version Reg_2 (Algo. 20). Par contre, le nombre d'accès mémoire ré-augmente à 6. Cette version est à la fois problématique pour un ASIC et pour un processeur généraliste. Pour un ASIC, c'est l'augmentation du nombre total de LOAD qui va ralentir la vitesse de traitement, tandis que pour un processeur généraliste, ce sont les deux balayages en mémoire et les défauts de caches qui vont doubler le temps de calcul, car pour ce type d'algorithme le temps d'exécution est fortement proportionnel au nombre d'accès mémoire.

Algorithme 19 : version Rot_1 : opérateur morphologique 2D avec un balayage d'image avec avec rotation de registres

for $i = 1$ **to** $n - 1$ **do**
  [chargement des points des deux premières colonnes de chaque ligne]
  $j \leftarrow 1$
  $a_0 \leftarrow X(i-1, j-1), b_0 \leftarrow X(i-1, j)$
  $a_1 \leftarrow X(i, j-1), b_1 \leftarrow X(i, j)$
  $a_2 \leftarrow X(i+1, j-1), b_2 \leftarrow X(i+1, j)$
  for $j = 1$ **to** $n - 1$ **do**
    [chargement des points de la dernière colonne]
    $c_0 \leftarrow X(i-1, j+1)$
    $c_1 \leftarrow X(i, j+1)$
    $c_2 \leftarrow X(i+1, j+1)$
    $r \leftarrow a_0 \oplus b_0 \oplus c_0 \oplus a_1 \oplus b_1 \oplus c_1 \oplus a_2 \oplus b_2 \oplus c_2$
    $Y(i, j) \leftarrow r$
    $a_0 \leftarrow b_0, b_0 \leftarrow c_0$ [rotation de registres]
    $a_1 \leftarrow b_1, b_1 \leftarrow c_1$ [rotation de registres]
    $a_2 \leftarrow b_2, b_2 \leftarrow c_2$ [rotation de registres]

Algorithme 20 : version Reg_2 : deux opérateurs morphologiques 1D (correspondant à la décomposition de l'élément structurant 3×3 (eq. 5.2) avec deux balayages en mémoire

for $i = 1$ **to** $n - 1$ **do**
  for $j = 1$ **to** $n - 1$ **do**
    $x_0 \leftarrow X(i-1, j), x_1 \leftarrow X(i, j), x_2 \leftarrow X(i+1, j)$
    $r \leftarrow x_0 \oplus x_1 \oplus x_2$
    $T(i, j) \leftarrow r$
for $i = 1$ **to** $n - 1$ **do**
  for $j = 1$ **to** $n - 1$ **do**
    $a \leftarrow T(i, j-1), b \leftarrow T(i, j), c \leftarrow T(i, j+1)$
    $r \leftarrow a \oplus b \oplus c$
    $Y(i, j) \leftarrow r$

En introduisant une nouvelle optimisation, nous allons pouvoir à la fois factoriser les calculs et réduire le nombre d'accès mémoire. Les deux passes nécessaires pour les opérateurs 1D peuvent être combinées ensemble. Pour cela, chaque résultat du premier opérateur 1D est stocké dans un registre. Cette transformation est appelée une *réduction*. Dans notre cas, c'est une réduction par colonne : au lieu de mémoriser 6 pixels (a_0, a_1, a_2, b_0, b_1, b_2 dans l'algorithme 19), d'une itération à l'autre de la boucle, c'est seulement deux valeurs *réduites* : r_a et r_b (Algo. 21, lignes 5 et 6) qui seront copiées. Dans la boucle interne qui ne contient que le chargement des 3 nouveaux points, la variable réduite r_c sera calculée de la même manière. Puis le second opérateur sera appliqué à ces trois variables réduites (ligne 12) pour donner le résultat final. Il ne reste plus qu'à effectuer une rotation de registres sur les registres réduits, soit deux copies au lieu de six. La version Red combine donc les avantage de Reg_2 (4 OP au lieu de 8) et de Rot (4 accès mémoire au lieu de 8 ou 10).

Algorithme 21 : version Red : implémentation en 1 passe de deux opérateurs morphologiques 1D avec *réduction* par colonne

```
1  for i = 1 to n − 1 do
2      a0 ← X(i − 1, j − 1), b0 ← X(i − 1, j)
3      a1 ← X(i, j − 1), b1 ← X(i, j)
4      a2 ← X(i + 1, j − 1), b2 ← X(i + 1, j)
5      ra ← a0 ⊕ a1 ⊕ a2    [réduction de la première colonne : opérateur vertical]
6      rb ← b0 ⊕ b1 ⊕ b2    [réduction de la seconde colonne : opérateur vertical]
7      for j = 1 to n − 1 do
8          c0 ← X(i − 1, j + 1)
9          c1 ← X(i, j + 1)
10         c2 ← X(i + 1, j + 1)
11         rc ← c0 ⊕ c1 ⊕ c2    [réduction de la troisième colonne : opérateur vertical]
12         r ← ra ⊕ rb ⊕ rc    [opérateur horizontal]
13         Y(i, j + 0) ← r
14         ra ← rb, rb ← rc    [rotation des registres réduits]
```

La complexité algorithmique, le nombre d'accès mémoire et l'intensité arithmétique de ces quatre versions sont récapitulés dans le tableau 5.2.

version	OP	LOAD + STORE	MOVE	IA
Reg (un opérateur 2D en 1 passe)	8	9+1=10	0	0.8
Rot_1 (un opérateur 2D en 1 passe)	8	3+1=4	6	2.0
Reg_2 (deux opérateurs 1D en 2 passes)	4	2(3+1)=8	0	0.5
Red (deux opérateurs 1D en 1 passes)	4	3+1=4	2	1.0

TABLE 5.2 – complexité des opérateurs morphologiques et intensité arithmétique

5.2.2 Optimisations logicielles

Comme nous l'avons dit, la simplicité de l'algorithme $\Sigma\Delta$ est à la fois son point fort et son point faible. Si un algorithme est composé de peu d'instructions, alors il est très difficile d'en diminuer le nombre ou d'en accélérer l'exécution. Comme $\Sigma\Delta$ est majoritairement composé de structures de contrôle `if-then-else`, nous allons nous focaliser sur l'optimisation du double test permettant d'incrémenter ou de décrémenter une variable. L'algorithme 22 reproduit les lignes 2 et 3 de l'algorithme 16. Comme après l'estimation de $M_t(x)$, $M_{t-1}(x)$ n'est plus utilisé (idem pour Vt et V_{t-1}), l'estimation peut se faire *in situ* dans le même tableau. Cela permet de faire disparaitre la clause **otherwise**, puisqu'il n'est plus nécessaire de faire de copie entre les tableaux associés aux instants t et $t-1$.

Soit r la variable contenant la valeur de $M(x)$ et x la valeur de l'image courante $I(x)$. La traduction naïve (version *basic*) de ces deux tests consiste donc à faire le premier test, puis le second. Bien qu'ils soient indépendants, les deux tests doivent être fait en série car c'est la même variable r qui est modifiée.

Algorithme 22 : version *basic* du double *if-then-else*

```
1 if r < x then r ← r + 1
2 if r > x then r ← r − 1
```

Comme les cas testés sont en exclusion mutuelle (si le premier test est vrai, le second est faux et inversement), la façon classique de diminuer en moyenne le nombre de tests réalisés est d'imbriquer les deux tests. C'est la version *nested* (Algo. 23) : le premier test est toujours réalisé, tandis que le second ne l'est que dans 50 % des cas (si les distributions de r et x sont uniformes).

Algorithme 23 : version *nested* du double *if-then-else*

```
1 if r < x then
2 |   r ← r + 1
3 else
4 |   if r > x then
5 |   |   r ← r − 1
```

Une alternative est de séparer les tests des calculs. Au lieu de faire l'incrémentation ou la décrémentation dans les structures de contrôle, ces dernières ne servent qu'à sélectionner la quantité δ – positive ou négative – qui sera ajoutée à r en dehors de ces structures.

Algorithme 24 : version *delta* du double *if-then-else*

```
1 δ ← 0
2 if r < x then δ ← +1
3 if r > x then δ ← −1
4 r ← r + δ
```

La dernière reformulation est plus un *hack* qu'une optimisation logicielle. Elle combine comparaison et addition et une astuce liée à l'arithmétique en complément à 2. Elle se base sur deux

points. Le premier est que le résultat (dans le cas de nombres 8 bits) d'une comparaison vraie se code `0xFF` et celui d'une comparaison fausse `0x00`. Le second, est qu'en complément à 2, `0xFF` vaut `-1`. Ainsi en logique positive, le cas vrai est codé par `-1`.

On peut donc directement utiliser le résultat des comparaisons pour faire des calculs d'incrément et de décrement, il suffit juste de permuter les opérations d'addition et de soustraction. Par exemple, l'expression $r \leftarrow r+1$ devient $r \leftarrow r-(-1)$ et finalement $r \leftarrow r-lt$ (Algo. 25 line 3).

La complexité de ces quatre versions est récapitulée dans la table 5.3 qui indique le nombre de comparaisons (opérations $<$ et $>$), le nombre d'additions, leur somme (puisque les comparaisons sont en fait des soustractions dont on teste le bit de signe) et le nombre de branchements conditionnels liés aux `if-then-else`.

Algorithme 25 : version *hack* du double *if-then-else* - arithmétique en complément à 2

1 $lt \leftarrow (r < x)$
2 $gt \leftarrow (r > x)$
3 $r \leftarrow r - lt$
4 $r \leftarrow r + gt$

Si `0xFF` n'était pas égal à `-1`, il aurait fallu une étape supplémentaire : le résultat des comparaisons aurait servi de masque binaire pour maintenir ou forcer à zéro les valeurs -1 et $+1$ de l'algorithme 26.

Algorithme 26 : version *mask* du double *if-then-else*

1 $lt \leftarrow (r < x)$
2 $gt \leftarrow (r > x)$
3 $inc \leftarrow (+1)$ `AND` lt
4 $dec \leftarrow (-1)$ `AND` gt
5 $r \leftarrow r + inc + dec$

Si l'on fait l'hypothèse que chaque comparaison est totalement aléatoire et non prédictible, le nombre de mauvaises prédictions (du prédicteur de branchement, pour un processeur généraliste) est alors proportionnel au nombre de branchements. Il peut donc être judicieux d'éviter ainsi les *stalls* du pipeline associés aux erreurs de prédiction. Comme nous pouvons le voir, seule la version *hack* ne possède aucun branchement conditionnel. C'est ainsi que sont codées les versions SIMD les plus efficaces de $\Sigma\Delta$. L'impact de ce *hack* sur Catapult-C sera analysé dans la section résultat.

version	CMP	ADD	total	BRANCH conditionnel
basic	1 + 1	0.5 + 0.5	3	1+1=2
nested	1 + 0.5	0.5 + 0.25	2.25	1+0.5=1.5
delta	1 + 1	0.5 + 0.5	3	1+1=2
hack	1 + 1	1 + 1	4	**0**

TABLE 5.3 – Complexité des différentes versions de *if-then-else*

5.2.3 Optimisations matérielles

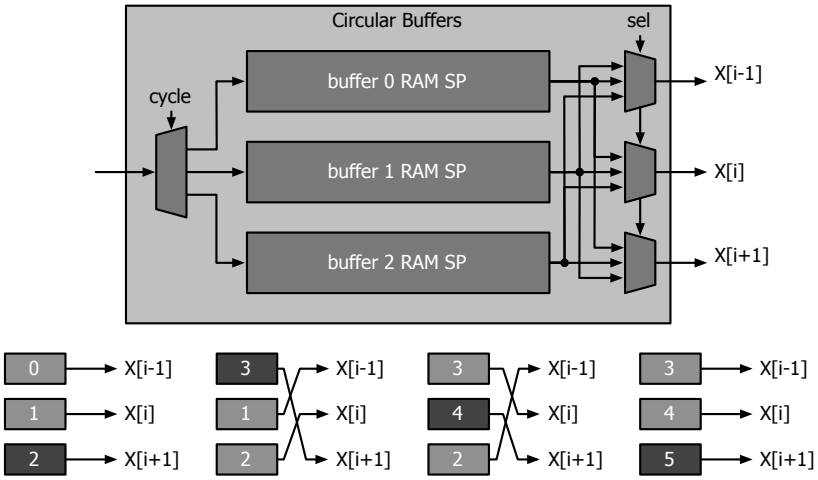

FIGURE 5.3 – Buffers circulaires : structure et fonctionnement

Les optimisations précédentes avaient pour but de permettre deux ou trois lectures de pixel à chaque cycle d'horloge grâce à des mémoires double port ou à des mémoires simple port entrelacées par bancs ($3 \times SP$). Mais plutôt que de faire 9 lectures rapides, il peut être intéressant de n'en faire que trois, s' il est possible de mémoriser et de ré-utiliser les pixels précédents. C'est l'objectif des buffers circulaires (CB).

Le fonctionnement des buffers circulaires est proche du fonctionement des registres à décalage utilisés pour l'optimisation Rot. Pour cela trois buffers de la largeur d'une ligne de l'image sont créés. Ils sont successivement remplis de manière à ce qu'en permanence la ligne i, ainsi que la précédente $(i-1)$ et la suivante $(i+1)$ s'y trouvent. Ils sont mis à jour de manière circulaire – d'où leur nom – car lorsque l'on passe à la ligne suivante $(i \rightarrow i+1)$, le buffer contenant la ligne la plus ancienne $(i-1)$ qui ne sert plus est recyclé en contenant la nouvelle ligne $(i+2)$.

5.2.4 Résultats et analyse

5.2.4.1 Optimisation des structures de test

freq (MHz)	200	400	600	800	moyenne
version *basic*	1.75	1.52	1.47	1.87	1.65
version *nested*	1.63	1.87	2.10	2.48	2.02
version *hack*	1.40	1.41	1.59	1.50	1.48
version *delta*	1.40	1.40	1.49	1.50	1.45
gain	×1.25	×1.09	×0.99	×1.25	**×1.14**

TABLE 5.4 – énergie (pJ/pixel) de l'algorithme$\Sigma\Delta$ pour $ii = 1$ en fonction de l'implémentation du `if-then-else`

Le tableau 5.4 présente l'impact de la ré-écriture des doubles `if-then-else` de l'algorithme $\Sigma\Delta$ d'un point de vue énergétique, pour une cadence de traitement de 1 cycle par pixel. Le gain moyen est de 14% et monte à 25% pour les fréquences 200 et 800 MHz. Si les versions *hack* et *delta* sont très proches, il est intéressant de constater que *nested* qui est la version qui fait le moins de comparaisons et le moins de branchements conditionnels (à part *hack*) est la moins efficace de toutes. Il faut donc faire attention en HLS à la manière de coder des structures de contrôle et ne pas forcément reproduire ce qu'on a appris.

5.2.4.2 Optimisation des opérateurs morphologiques

Nous allons maintenant analyser les résultats des différentes implantations de l'opérateur de morphologie mathématique qui sont regroupés dans le tableau 28.

Minimisation de ii et impact des HLT. Comme nous l'avons mis en évidence dans les deux précédents chapitres, c'est en minimisant la valeur de ii que l'on minimise l'énergie consommée. Or dans le cas d'un opérateur morphologique 3×3, il faut 9 accès mémoire avec une mémoire simple port. Ce qui se traduit par une consommation très élevée (que ce soit en mode $auto$ ou pour la synthèse minimisant ii. Le simple fait de passer à de la mémoire DP et en gardant la version sans optimisation Reg permet un gain moyen de $\times 1.38$ en mode $auto$ et $\times 1.68$ en mode *bestE*, car ainsi ii passe à 5 cyles. Grâce à l'optimisation Rot (avec de la mémoire SP), ii va descendre à 3 cycles et les gains respectifs des versions *bestS* et *bestE* vont monter à $\times 2.22$ et $\times 2.76$. Enfin l'optimisation Red va permettre d'atteindre des gains plus importants car, pour une même valeur de ii, elle diminue le nombre de calculs : $\times 2.99$ pour *bestS* et $\times \mathbf{4.31}$ pour *bestE*. L'impact des transformations de haut niveau sur la consommation énergétique est donc majeur. De plus, ce gain s'accompagne d'une diminution de la surface du circuit d'environ 14%. Il n'est plus nécessaire de choisir entre minimiser la surface ou l'énergie : les HLT permettent d'avoir les deux.

Optimisations matérielles et HLT. Afin d'aller plus loin dans la minimisation de l'énergie, il est nécessaire de combiner HLT et mémoire rapide. Quatre configurations permettre d'atteindre $ii = 1$, ce sont les HLT Rot et Red combinées à une mémoire entrelacée $3 \times SP$ ou à des buffers circulaires. Les coûts surfaciques et énergétiques liés aux buffers circulaires rendent leur utilisation non compétitive par rapport à de la mémoire entrelacée par bancs. La meilleure configuration est $Red + 3 \times SP$ et permet une réduction énergétique supérieure à **un ordre de grandeur** par rapport à la version initiale. Le gain moyen est de $\times \mathbf{12.1}$ tandis que la surface a diminué de 8%.

config	fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)								
Reg + SP	*auto*	2570	2570	2570	2570	2570	2570	2570
Reg + SP	$ii = 9$	3091	3091	3091	3091	3091	3094	3092
Reg + DP	*auto*	2893	2893	2893	2893	2893	2893	2893
Reg + DP	$ii = 5$	3206	3206	3208	3208	3206	3206	3207
Rot + SP	*auto*	2650	2650	2650	2650	2655	2681	2656
Rot + SP	$ii = 3$	3289	3289	3289	3289	3289	3311	3293
Rot + DP	*auto*	2910	2910	2910	2910	2910	2932	2914
Rot + DP	$ii = 2$	3531	3531	3531	3531	3536	3556	3536
Rot + 3×SP	*auto*	3013	3013	3013	3013	3013	3013	3013
Rot + 3×SP	$ii = 1$	3035	3035	3035	3035	3035	3037	3035
Rot + CB	*auto*	6605	6605	6605	6605	6605	6612	6606
Rot + CB	$ii = 1$	7945	7945	7945	7945	7945	7962	7948
Red + SP	*auto*	2013	2013	2013	2013	2018	2045	2019
Red + SP	$ii = 3$	2253	2253	2253	2253	2253	2268	2256
Red + DP	*auto*	2388	2388	2388	2388	2389	2407	2391
Red + DP	$ii = 2$	2685	2685	2685	2685	2687	2714	2690
Red + 3×SP	*auto*	2393	2393	2393	2393	2393	2393	2393
Red + 3×SP	$ii = 1$	2389	2389	2389	2389	2389	2391	2389
Rot + CB	*auto*	5939	5939	5939	5939	5939	5956	5942
Rot + CB	$ii = 1$	6904	6904	6904	6904	6904	6928	6908
énergie ($pJ/point$)								
Reg + SP	*auto*	19.660	14.950	13.448	12.672	12.206	11.894	14.138
Reg + SP	$ii = 9$	20.234	15.630	14.154	13.395	12.938	12.349	14.783
Reg + DP	*auto*	14.160	10.873	9.777	9.229	8.901	8.682	10.271
Reg + DP	$ii = 5$	11.881	9.263	8.395	7.959	7.693	7.519	8.785
Rot + SP	*auto*	8.837	6.745	6.074	5.729	5.474	5.355	6.369
Rot + SP	$ii = 3$	7.276	5.667	5.131	4.862	4.666	4.563	5.361
Rot + DP	*auto*	8.005	6.184	5.576	5.273	5.089	4.927	5.842
Rot + DP	$ii = 2$	5.296	4.176	3.802	3.616	3.477	3.415	3.964
Rot + 3×SP	*auto*	4.571	3.549	3.208	3.038	2.935	2.867	3.361
Rot + 3×SP	$ii = 1$	2.153	1.633	1.460	1.373	1.321	1.286	1.537
Rot + CB	*auto*	10.804	9.421	8.959	8.729	8.590	8.445	9.158
Rot + CB	$ii = 1$	6.539	5.612	5.303	5.148	5.055	4.987	5.440
Red + SP	*auto*	6.626	5.017	4.505	4.240	4.033	3.949	4.728
Red + SP	$ii = 3$	4.729	3.639	3.276	3.094	2.968	2.868	3.429
Red + DP	*auto*	6.430	4.912	4.405	4.152	4.001	3.863	4.627
Red + DP	$ii = 2$	3.910	3.033	2.741	2.595	2.495	2.437	2.868
Red + 3×SP	*auto*	3.503	2.696	2.426	2.292	2.211	2.157	2.548
Red + 3×SP	$ii = 1$	1.660	1.247	1.110	1.041	1.000	0.973	1.172
Rot + CB	*auto*	9.743	8.611	8.234	8.045	7.932	7.838	8.400
Rot + CB	$ii = 1$	5.693	4.949	4.701	4.577	4.502	4.454	4.813

TABLE 5.5 – Morphologie : surface puissance et énergie pour des fréquences de synthèse $\in$ [100 : 600] par pas de 100 MHz

5.3 Evaluation de l'impact des transformations pour les processeurs généralistes

Nous comparons maintenant l'impact de ces transformations sur les trois processeurs généralistes

5.3.1 Résultats et analyse

5.3.1.1 Structures de contrôle

Evaluer l'impact de la ré-écriture des doubles `if-then-else` sur Cortex-A9 et Penryn n'est pas une chose facile car ces processeurs sont capables de faire de l'exécution dans le désordre afin de masquer la latence des instructions et les problèmes de dépendance de données (*stalls* du pipeline). L'évaluation ne se fait que sur le processeur XP70 qui n'a pas de capacité d'exécution dans le désordre. Le benchmark est réalisé sur le simulateur du XP70 avec de la mémoire TCDM à 1 cycle ce qui permet d'avoir des résultats plus précis car non pollués par l'OS et où la part des accès mémoire dans le temps de calcul n'est pas majoritaire.

version	*cpp*
basic	31.04
nested	26.04
delta	30.04
hack	15.94
gain	$\times$1.95

TABLE 5.6 – *cpp* des différentes versions du double *if-then-else* sur XP70 avec une mémoire TCDM à 1 cycle

Nous pouvons voir (table 5.6) deux choses. La première est que la version *nested* est un peu plus rapide que la version *basic*. Imbriquer les structures de contrôle pour les processeurs simples afin de diminuer le nombre moyen d'aléas reste donc utile. La seconde, c'est l'impact majeur du *hack* proposé : le temps d'exécution est quasiment divisé par deux. Il est important, pour ce genre de processeur de bien faire la différence entre un test et une comparaison. Lorsque cela est possible, il faut re-coder un algorithme pour que les comparaisons soient faites sans structure de contrôle.

5.3.1.2 Opérateur morphologique

Le tableau 5.7 donne les *cpp* versions scalaires et SIMD de l'algorithme $\Sigma\Delta$ et l'opérateur morphologique. Pour ce dernier, les versions Reg, Rot et Red sont présentées, ainsi que l'impact des HLT (ici Reg/Red).

Concernant le XP70, l'impact de la version Red augmente avec la latence de la mémoire TCDM, ce qui était attendu.

Concernant le Cortex-A9, le gain en scalaire est proche de celui du XP70 et du Penryn, mais il est beaucoup plus élevé en SIMD. Cela vient de la latence des instructions SIMD Neon du Cortex-A9 et du fait qu'il n'est possible d'en lancer une que tous les deux cycles (contre un

	versions scalaires					versions SIMD				
		op. morphologique			gain		op. morphologique			gain
	$\Sigma\Delta$	*Reg*	*Rot*	*Red*	HLT	$\Sigma\Delta$	*Reg*	*Rot*	*Red*	HLT
XP70 TCDM=1c	31.04	17.17	12.04	12.08	$\times 1.42$	5.57	2.95	2.50	2.43	$\times 1.21$
XP70 TCDM=2c	84.04	67.56	43.36	47.23	$\times 1.43$	17.32	16.11	9.57	9.43	$\times 1.71$
XP70 TCDM=10c	303.60	241.67	182.88	139.95	$\times 1.73$	35.42	41.08	36.18	21.77	$\times 1.89$
Cortex-A9	95.10	86.70	58.90	53.40	$\times 1.62$	18.0	43.60	10.70	7.0	$\times 6.21$
Cortex-A15	53.10	11.50	9.70	5.30	$\times 2.18$	3.2	1.20	1.00	0.8	$\times 1.60$
Penryn	28.55	16.34	18.20	8.32	$\times 1.96$	1.60	1.15	1.33	1.02	$\times 1.13$

TABLE 5.7 – *cpp* des versions scalaires et SIMD de $\Sigma\Delta$ et de l'opérateur morphologique pour les trois processeurs généralistes ; gain des transformations HLT pour l'opérateur morphologique

cycle pour les autres processeurs). De plus, avec la version *Red*, les chaines de dépendance de données sont plus courtes que pour les versions *Reg* et *Rot* : le calcul des différentes variables r peuvent être entrelacés dans le pipeline (Algo.21, lignes 11 et 12) tandis que pour *Reg* (Algo 18, ligne 6) et *Rot* (Algo 19, ligne 11), il n'y a qu'un long calcul en série, si l'on fait l'hypothèse que le compilateur n'utilise pas l'associativité de l'opérateur morphologique pour implémenter le calcul sous forme d'arbre. Pour mettre cela en évidence, un benchmark sous un Cortex-A15 a été réalisé : les *cpp* mesurés sont bien plus bas que ceux du Cortex-A9 et proches de ceux du Penryn.

Enfin concernant le Penryn, les *cpp* sont plus bas que ceux du XP70 et du Cortex-A9 et proches de ceux du Cortex-A15. Les gains sont là aussi du même ordre de grandeur.

		op. morphologique			gain
	$\Sigma\Delta$	*Reg*	*Rot*	*Red*	HLT + SIMD
XP70 TCDM=1c	$\times\mathbf{5.57}$	$\times 5.82$	$\times 4.82$	$\times 4.97$	$\times\mathbf{6.03}$
XP70 TCDM=2c	$\times\mathbf{4.85}$	$\times 4.19$	$\times 4.53$	$\times 5.01$	$\times\mathbf{5.67}$
XP70 TCDM=10c	$\times\mathbf{8.57}$	$\times 5.88$	$\times 5.05$	$\times 6.43$	$\times\mathbf{9.53}$
Cortex-A9	$\times\mathbf{5.28}$	$\times 1.99$	$\times 5.51$	$\times 7.61$	$\times\mathbf{12.36}$
Cortex-A15	$\times\mathbf{16.74}$	$\times 9.36$	$\times 9.28$	$\times 6.86$	$\times\mathbf{14.96}$
Penryn	$\times\mathbf{17.84}$	$\times 14.21$	$\times 13.68$	$\times 8.16$	$\times\mathbf{16.02}$

TABLE 5.8 – gain SIMD, gain des HLT et gain total pour les trois processeurs généralistes

Le tableau 5.8 présente les gains SIMD pour des tailles de données telles que toutes les images tiennent dans le cache. Dans ces conditions les gains du SIMD sont proches de l'optimal. L'origine des gains surlinéaires pour l'algorithme $\Sigma\Delta$ est qu'en scalaire, c'est la version *basic* des doubles `if-then-else` qui est utilisée, alors qu'en SIMD, c'est la version *hack*.

5.3.1.3 ASIC vs GPP : comparaison en temps et en énergie

Le tableau 5.9 récapitule le *cpp*, le temps d'exécution et l'énergie consommée par point. Comme précédemment, le temps d'exécution correspond au temps d'exécution multi-threadé pour les processeurs bi-coeurs et l'estimation de l'énergie consommé est corrigée par un facteur lié à la différence de techno. Pour le *cpp* et le temps d'exécution, c'est la somme du temps de $\Sigma\Delta$ et de deux fois le temps d'un opérateur morphologique puisque l'on souhaite réaliser une ouverture 3×3 qui est la combinaison d'une érosion 3×3 et d'une dilatation 3×3.

	cpp	temps (ns/p)	énergie (pJ/p)	ratio temps	ratio énergie
ASIC (*bestE*)	3	5.000	4.80	$\times 1.00$	$\times 1.0$
ASIC (auto)	19	31.667	40.97	$\times 6.33$	$\times 8.5$
XP 70	10.43	23.178	579	$\times 4.64$	$\times 120.7$
Cortex A9 (bi-coeur)	32.03	13.347	27869	$\times 2.67$	$\times 5806.0$
Cortex A15 (bi-coeur)	4.72	1.387	6815	$\times 0.28$	$\times 1419.8$
Penryn ULV (bi-coeur)	3.64	1.525	26535	$\times 0.31$	$\times 5528.1$

TABLE 5.9 – ASIC, XP70, ARM Cortex-A9, Intel SU9300 Penryn ULV (énergie en 65 nm)

Concernant les GPP, le Cortex-A9 est cette fois dépassé par le Penryn qui se montre à la fois plus rapide et plus économe en énergie. Il faut ajouter le Cortex-A15 pour qu'un processeur ARM repasse devant un processeur Intel. En terme de vitesse, le XP70 est dépassé par les autres processeurs, par contre sa consommation énergétique reste encore très basse. Cette consommation est tellement basse que si on prenait un cluster de 16 processeurs XP70 (en faisant l'hypothèse que les autres paramètres passent à l'échelle), ce cluster irait aussi vite qu'un Penryn, mais avec une consommation énergétique qui resterait 2.8 plus faible que celle du Peryn ou du Cortex-A9.

Concernant la comparaison entre ASIC et GPP, on peut observer que le gap a augmenté par rapport aux résultats des filtres non récursifs et ce, bien que ces filtres rendent possible l'utilisation d'instructions SIMD permettant ainsi d'accélérer l'exécution de ces opérateurs sur GPP. Il y a deux raisons à cela. Tout d'abord la conception d'un système mémoire adapté aux opérateurs morphologiques et couplé à des transformations de haut niveau qui permettent d'atteindre un ii de 1 cycle. Ensuite, le fait qu'il soit possible pour un ASIC, lorsque les opérations arithmétiques sont très simples (`OR` et `AND` binaires) d'en exécuter 9 par cycle, contre seulement 2 ou 3 multiplications pour les FIR. L'ASIC profite naturellement du parallélisme de la logique combinatoire.

5.4 Conclusion du chapitre : *higher, faster and greener*

Dans ce chapitre nous avons présenté une nouvelle transformation algorithmique de haut niveau : la réduction par colonne, qui permet – en scalaire comme en SIMD – de réduire à la fois le nombre d'accès mémoire et le nombre d'opérations arithmétiques pour les opérateurs de morphologie mathématique. Cette transformation algorithmique combinée à un système mémoire efficace (mémoire entrelacée par bancs) permet d'atteindre une cadence de 1 cycle par point pour chaque algorithme (détection de mouvement $\Sigma\Delta$, opérateurs morphologiques binaires) et ainsi d'atteindre un niveau de consommation énergétique très bas. Cette consommation est douze fois plus faible que celle obtenue en mode automatique. Point remarquable, la réduction de la complexité des opérateurs morphologiques permet de plus d'obtenir une surface 8% plus petite que celle en mode automatique. Il n'est plus nécessaire de faire de compromis entre surface et consommation, les optimisations et transformations présentées permettent les deux.

Considérons maintenant des systèmes embarqués *extrêmes* comme les rétines électroniques. Leur raison d'être initiale était la très grande vitesse de calcul (par rapport aux processeurs généralistes de l'époque) et la faible consommation énergétique. Puis sont arrivés des processeurs plus efficaces, qu'ils soient généralistes comme le PowerPC G4 (10 W en 130 nm) d'une puissance comparable au supercalculateur Cray 1 ou spécialisé comme le DSP C64x (1 W en

90 nm). Ils ont permis d'atteindre des cadences de traitement réservées jusque là aux seules rétines [34] [38]. A partir de ce moment, les avantages des rétines – avantages qui perdurent encore aujourd'hui – sont le niveau d'intégration et le couplage entre le capteur et les processeurs élémentaires.

Si l'on compare les performances de la rétine *pvlsar34* de l'Ensta (développé par Thierry Bernard et programmée par Antoine Manzanera) à celles de l'ASIC implémentant l'algorithme $\Sigma\Delta$, les résultats sont très proches : 3.75 pJ/p pour la rétine et $5pJ/p$ pour l'ASIC. Cette comparaison n'est pas *fairplay* : la rétine intègre un capteur et des convertisseurs CAN en chaque point. La consommation du capteur APS qu'il faudrait ajouter à l'ASIC n'est pas prise en compte, ni le fait que si la rétine était gravée en 65 nm (contre 350), les processeurs élémentaires consommeraient $(350/65)^{1.5} = 12.5$ fois moins (les photosites consommant *a priori* toujours autant).

Ce qui est plus *fairplay* c'est d'observer qu'avec un décalage de 6 ans, un ASIC fait aussi bien (en terme de vitesse et de consommation) qu'une rétine, mais avec des temps de conception et de mise-au-point bien plus faible.

Les nouveaux outils de synthèse d'architectures forcent donc à repenser les approches classiques, y compris celles concernant des niches comme les rétines électroniques.

Conclusion et perspectives de recherche

Dans cette thèse, nous avons évalué les performances de Catapult-C, un logiciel de synthèse HLS utilisé en production à STMicroelectronics, sur trois exemples représentatifs du traitement du signal et des images à savoir les filtres non récursifs, les filtres récursifs et un algorithme complet de détection de mouvement avec post-traitement morphologique. Les synthèses ont été réalisées en techno 65 nm GP de ST. Pour les systèmes embarqués, les optimisations multi-critères (vitesse, surface, puissance et énergie) sont particulièrement importantes. Comme Catapult-C est très performant pour optimiser la surface et la puissance consommée, nous avons décidé d'explorer différentes stratégies pour optimiser la vitesse et l'énergie.

Nous avons mis en oeuvre différentes optimisations logicielles et matérielles auxquelles nous avons ajouté des transformations algorithmiques de haut niveau (HLT). L'impact des ces optimisations et transformations a été évalué pour ces trois opérateurs de TSI et mis en perspective avec des implantations sur trois processeurs généralistes : ST XP70, ARM Cortex-A9 et Intel Penryn ULV. Nous avons mis en évidence que ces transformations permettent des baisses de l'énergie consommée d'un facteur 5 à 12 permettant à l'ASIC de conserver un avantage en terme de consommation

Nous avons exploré de manière exhaustive l'espace des configurations pour comprendre où se trouvaient les configurations minimisant les critères de surface et d'énergie et nous avons relié ces configurations à la valeur de l'*initiation interval* ii qui est la cadence de lancement des calculs.

Afin d'augmenter l'automatisation du flux de synthèse, nous avons développé un axe de métaprogrammation par macros pour générer automatiquement les optimisations de code et les transformations d'algorithmes. Cela a permis au final de diminuer les temps de développement et de mise au point (car ne nécessite plus de re-codage ni d'expert applicatifs) tout en améliorant les performances des circuits.

La contribution de cette thèse est donc à la croisée des domaines de la conception d'architectures, de la compilation et des applications. C'est une proposition de méthodologie de synthèse très automatisée utilisant un code C unique (avec des types et des objets C++ synthétisables), servant à la fois pour la conception de l'algorithme – et sa vérification fonctionnelle – et pour la synthèse de l'architecture. En fonction des compromis de surface et de consommation, des transformations algorithmiques sont activables pour privilégier la vitesse d'exécution et la consommation énergétique ou pour minimiser la surface.

L'automatisation poussée ne signifie pas que les ingénieurs ne sont plus nécessaires en conception, au contraire. Cela signifie qu'il n'est plus nécessaire de consacrer beaucoup de temps sur des tâches réalisables par ces outils et que l'expertise des ingénieurs doit être mobi-

lisée sur des tâches à plus grande valeur ajoutée comme les transformations automatiques de haut niveau et la recherche de compromis surface/consommation.

Il reste encore beaucoup de recherches à faire dans le domaine de la synthèse HLS.

Concernant les algorithmes et l'évolution des besoins dans l'embarqué, il serait intéressant d'évaluer les capacités des outils de synthèse HLS lorsqu'ils seront capables de gérer des calculs flottants, et en particulier le format *binary16* défini en 2008 par l'IEEE (754-2008) qui spécifie un format de calcul 16 bits, car certains algorithmes passent difficilement en virgule fixe. Il serait tout aussi intéressant d'évaluer les capacités de ces outils pour des algorithmes où la partie contrôle est bien plus importante.

Concernant les architectures et l'impact des transformations HLT, il serait intéressant d'évaluer les dernières générations de DSP C66x de Texas Instrument (architecture VLIW SIMD multi-coeurs entière et flottante) ainsi que des architectures parallèles composées de processeurs XP70 dotés d'extensions SIMD VECx car ces processeurs sont nettement plus performants que les Cortex A9 Neon.

ANNEXE FIR

Cette annexe présente les résultats détaillés du filtre FIR5 comme étant la fusion de deux filtres FIR3 :

- FIR5 avec mémoire SP et optimisation Reg (tab. 1),
- FIR5 avec mémoire SP et optimisation Rot (tab. 2),
- FIR5 avec mémoire entrelacées par bancs et optimisation Reg (tab. 3).

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	5183	5244	5314	5390	5803	6433	5563
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	-	-	-	-	-	-	-
$ii = 4$	-	-	-	-	-	-	-
$ii = 5$	5565	5734	5926	5977	6239	6896	6056
puissance ($\mu W/point$)							
$auto$	316.95	482.17	679.63	860.85	1086.57	1378.68	800.81
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	-	-	-	-	-	-	-
$ii = 4$	-	-	-	-	-	-	-
$ii = 5$	345.20	550.39	784.65	987.37	1232.07	1614.28	918.99
énergie ($pJ/point$)							
$auto$	22.656	19.692	23.126	21.969	24.400	23.456	22.550
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	-	-	-	-	-	-	-
$ii = 4$	-	-	-	-	-	-	-
$ii = 5$	17.640	14.065	13.373	12.621	12.602	13.756	14.010

TABLE 1 – FIR5 + mémoire SP + Reg : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	5511	5579	5423	5467	5899	8060	5990
$ii = 1$	6686	6686	7748	8979	9399	9638	8189
$ii = 2$	6159	6622	7434	7751	8293	9900	7693
$ii = 3$	6086	6387	6862	6964	7364	9324	7165
$ii = 4$	6030	6298	6573	6826	7155	9439	7054
$ii = 5$	5996	5828	5526	5580	5882	8109	6072
puissance ($\mu W/point$)							
$auto$	353.70	541.87	720.66	909.93	1146.22	1544.46	869.47
$ii = 1$	327.74	479.55	864.70	1403.59	1783.93	2040.66	1150.03
$ii = 2$	374.03	631.63	1022.57	1350.01	1737.82	2176.49	1215.43
$ii = 3$	389.51	621.71	946.51	1210.45	1550.45	1950.70	1111.56
$ii = 4$	370.78	600.46	885.00	1150.01	1468.58	1891.99	1061.14
$ii = 5$	360.56	581.04	773.07	979.54	1235.56	1670.59	933.39
énergie ($pJ/point$)							
$auto$	25.360	22.200	22.143	20.969	23.497	21.092	22.541
$ii = 1$	3.369	2.465	2.966	3.618	3.678	3.503	3.267
$ii = 2$	7.675	6.484	7.001	6.936	7.142	7.451	7.115
$ii = 3$	11.981	9.565	9.714	9.317	9.548	10.007	10.022
$ii = 4$	15.198	12.309	12.098	11.793	12.048	12.932	12.730
$ii = 5$	18.475	14.889	13.209	12.553	12.667	14.270	14.344

TABLE 2 – FIR5 + mémoire SP + Rot : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	6524	6446	6576	6840	7153	9189	7121
$ii = 1$	10429	11229	12492	13410	13705	13815	12513
$ii = 2$	9209	9574	10330	11306	12265	13071	10959
$ii = 3$	8526	8626	9167	9657	10308	11012	9549
$ii = 4$	8245	8410	8800	8782	9345	10574	9026
$ii = 5$	7677	7668	7800	8378	8824	10039	8398
puissance ($\mu W/point$)							
$auto$	406.15	612.26	849.34	1097.32	1365.11	1823.41	1025.60
$ii = 1$	585.13	1009.64	1648.23	2281.10	2854.22	3188.85	1927.86
$ii = 2$	568.53	911.20	1387.09	1841.99	2403.73	2864.41	1662.83
$ii = 3$	529.88	511.74	1198.37	1570.72	2003.38	2437.15	1375.21
$ii = 4$	495.06	777.99	1082.35	1374.80	1775.42	2262.96	1294.76
$ii = 5$	482.97	739.03	1024.87	1369.98	1721.15	2147.06	1247.51
énergie ($pJ/point$)							
$auto$	29.032	25.005	26.012	28.004	30.655	27.922	27.772
$ii = 1$	6.027	5.205	5.675	5.897	5.908	5.490	5.700
$ii = 2$	11.655	9.344	9.492	9.454	9.875	9.801	9.937
$ii = 3$	16.267	7.858	12.275	12.067	12.317	12.482	12.211
$ii = 4$	20.243	15.910	14.763	14.064	14.534	15.433	15.825
$ii = 5$	24.680	18.886	17.464	17.512	17.604	18.293	19.073

TABLE 3 – FIR5 + 5 mémoires SP entrelacées : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

ANNEXE IIR

Cette annexe présente les résultats détaillés du filtre IIR12 et du filtre IIR11 :
– IIR12 forme *Normal* + mémoire DP + Reg, (tab. 4),
– IIR12 forme *Factor* + mémoire DP + Reg, (tab. 5),

– IIR12 forme *Normal* + mémoire SP + LU (tab. 6),
– IIR12 forme *Factor* + mémoire SP + LU (tab. 7),

– IIR11 forme *Normal* + mémoire SP + Reg (tab. 8),
– IIR11 forme *Factor* + mémoire SP + Reg (tab. 9),

– IIR11 forme *Normal* + mémoire SP + Rot (tab. 10),
– IIR11 forme *Factor* + mémoire SP + Rot (tab. 11),

– IIR11 forme *Normal* + mémoire DP + Reg (tab. 12),
– IIR11 forme *Factor* + mémoire DP + Reg (tab. 13),

– IIR11 forme *Normal* + mémoire SP + LU (tab. 14),
– IIR11 forme *Factor* + mémoire SP + LU (tab.15),

– IIR forme *Factor*, moyenne des surfaces et des énergies (tab. 16),
– IIR forme *Delay*, moyenne des surfaces et des énergies (tab. 17).

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	4517	4551	4802	4608	4806	5622	4818
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	5410	5496	5496	6029	-	-	5608
$ii = 3$	5149	5989	6111	5577	6949	-	5955
$ii = 4$	4647	5175	5175	5537	6559	7117	5702
puissance ($\mu W/point$)							
$auto$	283.94	431.33	623.68	748.75	931.03	1234.58	708.89
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	295.10	446.89	593.90	864.27	-	-	550.04
$ii = 3$	318.18	575.05	805.94	905.86	1392.10	-	799.43
$ii = 4$	299.47	527.15	712.26	880.82	1285.79	1646.17	891.94
énergie ($pJ/point$)							
$auto$	14.211	12.950	14.563	13.112	13.044	14.414	13.716
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	5.919	4.482	3.971	4.334	-	-	4.677
$ii = 3$	9.564	8.648	8.080	6.807	8.374	-	8.295
$ii = 4$	11.996	10.564	9.515	8.821	10.306	10.996	10.366

TABLE 4 – IIR12 forme *Normal* + mémoire DP + Reg : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	4189	4394	4583	4659	4779	5063	4611
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	4731	4731	5070	4992	-	-	4881
$ii = 3$	4228	4228	4323	4621	5309	5618	4721
$ii = 4$	4173	4173	5155	4666	5377	5711	4876
puissance ($\mu W/point$)							
$auto$	275.30	440.56	646.58	757.66	944.75	1165.39	705.04
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	280.54	426.64	650.11	967.85	-	-	581.29
$ii = 3$	281.60	437.18	627.99	894.67	982.61	1396.75	770.13
$ii = 4$	274.72	424.48	705.50	851.96	1010.68	1288.02	759.23
énergie ($pJ/point$)							
$auto$	11.025	11.025	10.787	11.374	11.346	11.663	11.203
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	5.627	4.279	4.347	4.856	-	-	4.777
$ii = 3$	8.465	6.571	6.292	6.723	5.907	7.000	6.826
$ii = 4$	11.002	8.500	9.425	8.532	8.097	8.599	9.026

TABLE 5 – IIR12 forme *Factor* + mémoire DP + Reg : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	5084	5211	6450	5268	5687	7804	5917
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	8763	10306	11101	8692	12470	-	10266
$ii = 4$	7362	7898	9380	11172	13470	14708	10665
puissance ($\mu W/point$)							
$auto$	309.51	468.02	685.31	827.27	1034.10	1498.28	803.75
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	464.25	886.09	1152.93	1242.92	2081.22	-	1165.48
$ii = 4$	405.26	625.33	1075.58	1469.17	1984.95	2597.48	1359.63
énergie ($pJ/point$)							
$auto$	31.124	23.532	25.269	27.026	29.107	35.129	28.531
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	14.132	13.487	11.710	9.468	12.708	-	12.301
$ii = 4$	16.389	12.645	14.520	14.875	16.090	17.546	15.344

TABLE 6 – IIR12 forme *Normal* + mémoire SP + LU : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	5689	5826	7079	6203	6507	6765	6345
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	8007	9417	10531	11147	16063	17710	12146
$ii = 4$	7187	7712	9183	10462	12663	-	9441
puissance ($\mu W/point$)							
$auto$	327.15	495.31	742.54	882.44	1082.05	1301.96	805.24
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	390.53	745.13	1163.90	1545.30	2405.56	3059.87	1551.72
$ii = 4$	388.78	599.90	986.49	1403.17	1948.67	-	1065.40
énergie ($pJ/point$)							
$auto$	29.607	22.413	24.890	26.616	32.627	30.532	27.781
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	11.877	11.330	11.810	11.794	14.716	15.569	12.850
$ii = 4$	15.711	12.122	13.308	14.217	15.807	-	14.233

TABLE 7 – IIR12 forme *Factor* + mémoire SP + LU : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	3135	3125	3389	3537	3692	3853	3455
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	4000	4000	4097	4282	-	-	4095
$ii = 3$	3119	3119	3870	3914	4211	4613	3808
$ii = 4$	3115	3115	3390	3953	4141	4338	3675
puissance ($\mu W/point$)							
$auto$	197.85	304.36	448.16	581.14	738.13	884.02	525.61
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	266.87	412.35	568.10	762.25	-	-	502.39
$ii = 3$	199.87	308.01	541.03	691.05	883.41	1122.80	624.36
$ii = 4$	197.92	304.39	457.18	687.45	879.10	1018.20	590.71
énergie ($pJ/point$)							
$auto$	7.924	6.095	7.477	8.724	8.865	8.847	7.989
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	5.358	4.140	3.802	3.826	-	-	4.282
$ii = 3$	6.008	4.629	5.424	5.196	5.314	5.629	5.367
$ii = 4$	7.926	6.095	6.105	6.888	7.047	6.801	6.810

TABLE 8 – IIR11 forme *Normal* + mémoire SP + Reg : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2688	2688	2903	3046	3065	3231	2937
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	2681	2681	2816	3151	-	-	2832
$ii = 3$	2657	2657	2918	2966	3177	3292	2945
$ii = 4$	2702	2702	2915	3042	3062	3190	2936
puissance ($\mu W/point$)							
$auto$	158.86	239.06	344.43	456.83	554.50	690.68	407.39
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	159.65	240.81	366.13	477.24	-	-	310.96
$ii = 3$	157.33	237.00	368.61	459.02	625.38	714.36	426.95
$ii = 4$	158.44	237.94	367.02	461.28	560.17	690.84	412.62
énergie ($pJ/point$)							
$auto$	4.774	3.592	4.598	5.716	5.550	5.761	4.998
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	3.202	2.415	2.448	2.393	-	-	2.615
$ii = 3$	4.728	3.561	3.693	3.449	3.760	3.579	3.795
$ii = 4$	6.345	4.765	4.900	4.620	4.488	4.612	4.955

TABLE 9 – IIR11 forme *Factor* + mémoire SP + Reg : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	3042	3042	3309	3359	3714	3636	3350
$ii = 1$	3842	3861	3861	4522	4589	5443	4353
$ii = 2$	2977	2977	3443	3415	3785	4430	3505
$ii = 3$	2943	2943	3230	3266	3675	3795	3309
$ii = 4$	2968	2968	3208	3265	3519	3751	3280
puissance ($\mu W/point$)							
$auto$	195.24	301.34	431.34	551.31	720.32	812.95	502.08
$ii = 1$	212.27	314.02	418.44	680.68	823.39	1202.05	608.48
$ii = 2$	187.69	289.98	490.43	625.03	836.76	1085.28	585.86
$ii = 3$	180.08	276.42	423.87	538.93	720.69	941.96	513.66
$ii = 4$	178.60	272.02	410.57	522.78	677.79	859.62	486.90
énergie ($pJ/point$)							
$auto$	5.867	4.527	5.758	5.520	5.770	6.781	5.704
$ii = 1$	2.137	1.581	1.404	1.715	1.660	2.021	1.753
$ii = 2$	3.765	2.908	3.281	3.136	3.358	3.630	3.346
$ii = 3$	5.411	4.153	4.247	4.050	4.333	4.721	4.486
$ii = 4$	7.153	5.447	5.481	5.234	5.429	5.739	5.747

TABLE 10 – IIR11 forme *Normal* + mémoire SP + *Rot* : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2694	2694	2827	2983	3033	3208	2907
$ii = 1$	2567	2567	2917	2965	3228	3742	2998
$ii = 2$	2647	2647	2955	3154	3174	3281	2976
$ii = 3$	2610	2610	2842	3073	3093	3255	2914
$ii = 4$	2636	2636	2758	2980	2999	3161	2862
puissance ($\mu W/point$)							
$auto$	157.28	237.69	335.44	445.92	550.01	686.41	402.13
$ii = 1$	149.61	225.99	369.00	466.98	634.34	860.32	451.04
$ii = 2$	155.53	235.90	381.43	511.78	621.66	763.93	445.04
$ii = 3$	150.07	225.89	346.25	483.01	587.28	728.52	420.17
$ii = 4$	149.68	224.48	329.32	448.27	544.32	677.58	395.61
énergie ($pJ/point$)							
$auto$	4.726	3.571	4.478	5.579	5.505	5.726	4.931
$ii = 1$	1.506	1.138	1.240	1.177	1.279	1.445	1.297
$ii = 2$	3.120	2.366	2.552	2.569	2.496	2.556	2.610
$ii = 3$	4.509	3.394	3.469	3.631	3.532	3.651	3.698
$ii = 4$	5.995	4.495	4.396	4.489	4.361	4.524	4.710

TABLE 11 – IIR11 forme *Factor* + mémoire SP + *Rot* : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	3004	3004	3280	3309	3674	3667	3323
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	3011	3011	3011	3222	-	-	3064
$ii = 3$	2996	2996	3294	3336	3681	4065	3395
$ii = 4$	3021	3021	3272	3297	3657	3781	3342
puissance ($\mu W/point$)							
$auto$	190.85	294.22	433.49	552.42	742.25	848.18	510.24
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	195.69	303.11	410.51	572.24	-	-	370.39
$ii = 3$	190.96	294.13	448.17	575.45	770.38	947.82	537.82
$ii = 4$	189.28	290.50	433.61	551.27	744.35	898.02	517.84
énergie ($pJ/point$)							
$auto$	5.735	4.420	5.787	5.531	5.945	7.075	5.749
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	3.925	3.040	2.745	2.870	-	-	3.145
$ii = 3$	5.738	4.419	4.490	4.324	4.631	4.748	4.725
$ii = 4$	7.580	5.817	5.789	5.519	5.962	5.996	6.111

TABLE 12 – IIR11 forme *Normal* + mémoire DP + Reg : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2783	2783	2940	3048	3068	3258	2980
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	2797	2797	3017	3224	-	-	2959
$ii = 3$	2777	2777	2971	3011	3272	3510	3053
$ii = 4$	2804	2804	2952	3061	3081	3220	2987
puissance ($\mu W/point$)							
$auto$	162.99	245.30	337.17	445.65	539.74	671.16	400.34
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	165.32	248.83	370.75	516.74	-	-	325.41
$ii = 3$	163.21	245.35	359.28	463.50	625.02	783.93	440.05
$ii = 4$	163.48	245.51	344.35	460.06	558.08	690.52	410.33
énergie ($pJ/point$)							
$auto$	4.898	3.685	4.501	5.576	5.403	5.598	4.944
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	3.316	2.496	2.479	2.591	-	-	2.720
$ii = 3$	4.904	3.686	3.600	3.483	3.757	3.927	3.893
$ii = 4$	6.547	4.916	4.597	4.607	4.471	4.610	4.958

TABLE 13 – IIR11 forme *Factor* + mémoire DP + Reg : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	3331	3331	3331	3697	4162	4134	3664
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	4300	4306	4739	4767	5383	7561	5176
$ii = 3$	4117	4202	5348	4937	5548	-	4830
$ii = 4$	3476	3444	3875	5106	4287	5006	4199
puissance ($\mu W/point$)							
$auto$	207.78	317.97	428.15	579.43	764.56	892.61	531.75
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	244.00	362.85	560.43	737.29	1037.85	1666.57	768.17
$ii = 3$	234.04	363.59	581.49	741.89	1010.07	-	586.22
$ii = 4$	220.98	329.82	509.87	846.69	836.23	1131.32	645.82
énergie ($pJ/point$)							
$auto$	10.409	7.965	7.150	8.706	9.190	10.428	8.975
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	4.913	3.653	3.762	3.715	4.184	5.599	4.304
$ii = 3$	7.049	5.475	5.841	5.590	6.088	-	6.009
$ii = 4$	8.865	6.616	6.822	8.504	6.713	7.572	7.515

TABLE 14 – IIR11 forme *Normal* + mémoire SP + LU : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	3347	3336	3481	4390	3590	3722	3644
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	3559	3549	3657	3802	4148	4595	3885
$ii = 3$	3371	3431	3515	3680	4922	5317	4039
$ii = 4$	3373	3439	3523	4607	3883	5359	4031
puissance ($\mu W/point$)							
$auto$	182.16	290.06	418.98	564.85	639.60	778.57	479.04
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	224.56	339.54	483.14	636.43	857.74	1083.17	604.10
$ii = 3$	192.35	318.21	447.74	610.96	794.07	1005.76	561.52
$ii = 4$	199.22	312.08	414.30	587.66	705.14	932.20	525.10
énergie ($pJ/point$)							
$auto$	9.126	7.266	8.393	9.899	8.967	11.691	9.224
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	4.526	3.422	3.246	3.210	3.461	3.642	3.585
$ii = 3$	5.797	4.795	4.498	4.606	4.792	5.058	4.924
$ii = 4$	7.992	6.260	5.543	5.900	5.663	6.239	6.266

TABLE 15 – IIR11 forme *Factor* + mémoire SP + LU : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

mémoire	SP	DP	SP	SP	DP
optimisation	*Reg*	*Reg*	*Rot*	*LU*	*LU*
un filtre IIR11 forme					
surface(*best S*)	2891	2966	**2836**	3526	3560
surface(*best E*)	2966	3103	2998	3885	4620
energie(*best S*)	3.88	4.43	2.85	8.34	8.30
energie(*best E*)	2.97	3.09	**1.30**	3.59	3.14
ii(*best E*)	2	2	1	2	1
cascade de deux filtres IIR11 forme					
surface(*best S*)	18682	30215	**18254**	19001	30525
surface(*best E*)	19218	30658	19536	19433	32110
energie(*best S*)	45.58	53.19	39.63	94.21	106.47
energie(*best E*)	26.99	30.84	**18.66**	48.93	46.53
ii(*best E*)	2	2	1	2	1
pipeline de deux filtres IIR11					
surface(*best S*)	3741	3657	**3587**	5667	5703
surface(*best E*)	4444	3831	4886	6575	7724
energie(*best S*)	5.62	6.84	7.39	14.72	14.04
energie(*best E*)	4.62	4.14	**2.07**	6.05	5.80
ii(*best E*)	2	2	1	2	1
fusion de deux filtres IIR11 = un filtre IIR12					
surface(*best S*)	4063	4522	**4029**	6345	7001
surface(*best E*)	4345	5075	5490	12146	11984
energie(*best S*)	9.81	7.53	9.06	27.78	27.07
energie(*best E*)	6.77	5.34	**2.38**	12.85	12.66
ii(*best E*)	3	2	1	3	2

TABLE 16 – IIR forme *Factor*, moyenne des surfaces et des énergies pour des fréquences de synthèse allant de 100 à 600 MHz par pas de 100 MHz

mémoire	SP	DP	SP	SP	DP
optimisation	*Reg*	*Reg*	*Rot*	*LU*	*LU*
un filtre IIR11					
surface(*best S*)	4320	4560	**3944**	5865	6217
surface(*best E*)	5687	5752	6119	10553	10323
energie(*best S*)	12.04	4.21	7.97	18.31	18.72
energie(*best E*)	5.17	3.47	**2.46**	8.26	8.25
ii(*best E*)	2	1	1	3	3
cascade de deux filtres IIR11					
surface(*best S*)	20858	32741	**20498**	23957	36313
surface(*best E*)	23406	35345	26222	29120	41196
energie(*best S*)	70.60	90.47	58.92	95.08	101.45
energie(*best E*)	35.21	31.02	**22.75**	60.14	53.36
ii(*best E*)	3	2	1	3	2
pipeline de deux filtres IIR11					
surface(*best S*)	45882	6072	**5373**	8692	9179
surface(*best E*)	10231	10471	11418	20014	18707
energie(*best S*)	21.46	20.00	18.05	42.56	46.16
energie(*best E*)	9.71	6.37	**4.73**	16.42	16.58
ii(*best E*)	2	1	1	3	2
fusion de deux filtres IIR11 = un filtre IIR12					
surface(*best S*)	5014	5482	**4886**	8167	8994
surface(*best E*)	5716	8104	8184	12632	15549
energie(*best S*)	15.65	18.32	12.80	38.00	35.12
energie(*best E*)	7.71	6.73	**3.43**	14.70	14.51
ii(*best E*)	3	2	1	4	2

TABLE 17 – IIR forme *Delay*, moyenne des surfaces et des énergies pour des fréquences de synthèse allant de 100 à 600 MHz par pas de 100 MHz

ANNEXE SIGMA-DELTA

Cette annexe présente les résultats détaillés de l'implantation du double `if-then-else` de l'algorithme ΣΔ :

- ΣΔ avec 1 mémoire SP et forme *delta* (tab. 18),
- ΣΔ avec 1 mémoire DP et forme *delta* (tab. 19),
- ΣΔ avec 2 mémoires SP et forme *delta* (tab. 20),

- ΣΔ avec 1 mémoire SP et forme *hack* (tab. 21),
- ΣΔ avec 1 mémoire DP et forme *hack* (tab. 21),
- ΣΔ avec 2 mémoires SP et forme *hack* (tab. 23),

- ΣΔ avec 1 mémoire SP et forme *flag* (tab. 24),
- ΣΔ avec 1 mémoire DP et forme *flag* (tab. 25),
- ΣΔ avec 2 mémoires SP et forme *flag* (tab. 26).

Le dernier tableau (tab. 27) récapitule les performances moyennes pour les formes *delta*, *hack* et *flag*

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2288	2596	2924	2957	3013	3077	2809
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	2197	2297	2531	3227	3176	4157	2565
$ii = 3$	2206	2763	2717	3148	3063	3479	2896
$ii = 4$	2211	2770	2940	3057	3376	3332	2948
$ii = 5$	2239	2795	2964	3037	3143	3283	2910
$ii = 6$	2263	2822	2992	3023	3096	3404	2933
$ii = 7$	2242	2810	2971	2993	3072	3367	2909
$ii = 8$	2255	2814	2984	3015	3084	3394	2924
puissance (μW/point)							
$auto$	148.73	258.17	392.63	457.82	585.41	694.68	422.91
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	141.19	237.00	342.77	533.46	670.28	1063.87	498.10
$ii = 3$	141.80	292.30	363.33	572.82	607.22	795.24	462.12
$ii = 4$	140.81	286.17	393.77	512.18	637.56	751.35	453.64
$ii = 5$	141.01	281.39	389.81	496.47	623.23	792.35	454.04
$ii = 6$	142.30	279.41	388.71	484.51	596.26	811.61	450.47
$ii = 7$	141.08	278.00	385.64	473.92	592.13	799.60	445.06
$ii = 8$	141.15	275.05	382.59	473.83	587.01	795.42	442.51
énergie (pJ/point)							
$auto$	2.987	3.884	5.246	6.877	8.206	8.114	5.886
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	2.836	2.380	2.295	2.689	2.703	3.589	2.749
$ii = 3$	4.266	4.397	3.644	4.308	3.654	3.988	4.043
$ii = 4$	5.644	5.736	5.261	5.133	5.111	5.020	5.318
$ii = 5$	7.063	7.047	6.508	6.216	6.243	6.614	6.615
$ii = 6$	8.550	8.394	7.785	7.278	7.165	8.128	7.883
$ii = 7$	9.888	9.742	9.009	8.304	8.300	9.340	9.097
$ii = 8$	11.304	11.014	10.213	9.487	9.402	10.617	10.339

TABLE 18 – $\Sigma\Delta$ + mémoire SP + Delta : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2396	2708	3032	3080	3071	3097	2897
$ii = 1$	2390	2837	3053	3322	3173	3842	3103
$ii = 2$	2323	2463	2696	3363	3439	3675	2993
$ii = 3$	2333	2799	3015	3317	3123	3494	3014
$ii = 4$	2336	2808	2928	3180	3352	3312	2986
$ii = 5$	2363	2833	2955	3087	3150	3148	2923
$ii = 6$	2390	2861	2983	3080	3152	3265	2955
$ii = 7$	2367	2840	2962	3077	3133	3232	2935
$ii = 8$	2380	2852	2972	3099	3136	3211	2942
puissance ($\mu W/point$)							
$auto$	153.73	266.66	399.71	471.95	559.65	659.44	418.52
$ii = 1$	143.18	273.35	401.64	541.32	643.92	864.81	478.04
$ii = 2$	147.31	253.59	357.29	600.98	738.89	895.10	498.86
$ii = 3$	147.92	290.17	404.20	577.39	596.50	776.04	465.37
$ii = 4$	146.22	282.97	375.23	518.38	613.60	762.50	449.82
$ii = 5$	146.09	278.32	370.03	490.89	595.32	694.02	429.11
$ii = 6$	147.25	276.99	367.68	475.57	567.67	713.26	424.74
$ii = 7$	145.83	273.7	364.45	468.61	562.12	697.41	418.69
$ii = 8$	145.76	271.41	361.06	465.14	557.29	688.30	414.83
énergie ($pJ/point$)							
$auto$	3.074	3.999	5.329	7.079	7.835	7.693	5.835
$ii = 1$	1.442	1.387	1.370	1.384	1.317	1.474	1.396
$ii = 2$	2.945	2.535	2.381	3.028	2.978	3.006	2.812
$ii = 3$	4.437	4.352	4.041	4.330	3.578	4.069	4.134
$ii = 4$	5.847	5.658	5.002	5.183	4.908	5.083	5.280
$ii = 5$	7.303	6.957	6.166	6.136	5.953	5.783	6.383
$ii = 6$	8.833	8.308	7.352	7.133	6.811	7.132	7.595
$ii = 7$	10.206	9.577	8.502	8.200	7.869	8.136	8.748
$ii = 8$	11.658	10.854	9.627	9.302	8.916	9.176	9.922

TABLE 19 – $\Sigma\Delta$ + mémoire DP + Delta : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2256	2505	2836	2965	2921	2943	2738
$ii = 1$	2246	2650	2943	3492	3519	3542	3065
$ii = 2$	2197	2912	3610	4333	4659	4464	3696
$ii = 3$	2203	2639	3189	3945	4013	4164	3359
$ii = 4$	2210	2645	2965	3528	3617	4034	3167
$ii = 5$	2238	2671	2991	3320	3417	3454	3015
$ii = 6$	2263	2699	3019	3063	3404	3494	2990
$ii = 7$	2242	2678	2998	3033	3154	3174	2880
$ii = 8$	2254	2689	3009	3056	3167	3234	2902
puissance ($\mu W/point$)							
$auto$	145.43	244.96	377.83	451.33	558.72	660.27	406.42
$ii = 1$	138.48	255.25	396.37	644.38	781.91	775.83	498.70
$ii = 2$	139.35	299.20	535.67	824.72	1079.65	1205.92	680.75
$ii = 3$	138.90	260.05	448.40	693.47	866.85	1043.19	575.14
$ii = 4$	138.20	253.36	397.50	605.21	772.53	974.53	523.56
$ii = 5$	138.50	249.75	391.89	542.72	693.39	811.69	471.32
$ii = 6$	139.94	249.23	387.67	474.05	661.2	796.71	451.47
$ii = 7$	139.00	246.61	381.79	466.77	621.59	729.16	430.82
$ii = 8$	138.98	244.71	377.38	463.39	619.52	735.79	429.96
énergie ($pJ/point$)							
$auto$	2.908	3.674	5.037	6.769	7.822	7.703	5.652
$ii = 1$	1.395	1.296	1.341	1.661	1.612	1.313	1.436
$ii = 2$	2.786	3.003	3.598	4.187	4.385	4.082	3.674
$ii = 3$	4.166	3.900	4.495	5.241	5.241	5.269	4.719
$ii = 4$	5.527	5.066	5.299	6.075	6.204	6.534	5.784
$ii = 5$	6.923	6.243	6.531	6.794	6.944	6.774	6.701
$ii = 6$	8.394	7.475	7.752	7.110	7.944	7.977	7.775
$ii = 7$	9.728	8.630	8.907	8.168	8.702	8.506	8.773
$ii = 8$	11.116	9.786	10.062	9.267	9.911	9.810	9.992

TABLE 20 – $\Sigma\Delta$ + 2 mémoires SP + Delta : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2297	2605	2921	2982	3023	3118	2824
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	2206	2329	2505	3271	3169	4208	2948
$ii = 3$	2213	2770	2984	3335	3184	-	2897
$ii = 4$	2218	2776	2944	3135	3517	3224	2969
$ii = 5$	2246	2800	2968	3085	3166	3382	2941
$ii = 6$	2272	2828	2997	3092	3128	3321	2940
$ii = 7$	2251	2815	2976	3047	3225	3212	2921
$ii = 8$	2263	2822	2988	3046	3239	3223	2930
puissance ($\mu W/point$)							
$auto$	149.35	258.01	370.10	465.02	586.94	661.50	415.15
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	141.82	261.33	337.75	550.51	676.21	1072.59	506.70
$ii = 3$	142.32	288.95	408.59	592.77	624.91	-	411.51
$ii = 4$	141.33	282.02	392.92	529.52	646.52	749.17	456.91
$ii = 5$	141.53	279.57	388.74	502.01	606.83	805.83	454.09
$ii = 6$	142.93	278.31	387.63	492.36	579.41	795.07	445.95
$ii = 7$	141.70	275.95	384.86	481.07	638.28	746.30	444.69
$ii = 8$	141.68	273.41	381.77	477.87	635.58	739.82	441.69
énergie ($pJ/point$)							
$auto$	3.000	3.881	6.179	6.985	8.227	9.932	6.367
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	2.849	2.624	2.261	2.775	2.727	3.619	2.809
$ii = 3$	4.282	4.347	4.098	4.458	3.760	-	4.189
$ii = 4$	5.665	5.652	5.250	5.307	5.183	5.005	5.344
$ii = 5$	7.089	7.001	6.490	6.286	6.079	6.727	6.612
$ii = 6$	8.588	8.361	7.764	7.396	6.963	7.962	7.839
$ii = 7$	9.931	9.670	8.991	8.429	8.947	8.717	9.114
$ii = 8$	11.347	10.948	10.191	9.568	10.180	9.875	10.351

TABLE 21 – $\Sigma\Delta$ + mémoire SP + Hack : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2403	2714	3036	3145	3079	3152	2922
$ii = 1$	2398	2846	3178	3403	3191	4035	3175
$ii = 2$	2333	2494	2619	3423	3460	3801	3022
$ii = 3$	2341	2811	3029	3344	3196	-	2944
$ii = 4$	2344	2814	2938	3254	3592	3352	3049
$ii = 5$	2371	2842	2964	3149	3173	3212	2952
$ii = 6$	2400	2867	2991	3105	3178	3191	2955
$ii = 7$	2377	2846	2970	3136	3109	3109	2925
$ii = 8$	2388	2858	2982	3121	3120	3129	2933
puissance ($\mu W/point$)							
$auto$	154.02	264.34	376.45	484.54	558.39	635.72	412.24
$ii = 1$	143.66	274.40	421.07	546.81	635.04	925.58	491.09
$ii = 2$	147.92	272.29	347.30	611.47	733.39	927.64	506.67
$ii = 3$	148.44	286.68	406.75	573.30	611.89	-	405.41
$ii = 4$	146.73	279.14	375.69	535.69	644.69	775.24	459.53
$ii = 5$	146.6	274.37	370.46	502.05	597.55	695.69	431.12
$ii = 6$	147.85	272.73	368.07	479.94	568.87	682.62	420.01
$ii = 7$	146.44	269.35	364.84	479.42	562.08	661.56	413.95
$ii = 8$	146.26	267.08	361.54	471.41	558.22	655.20	409.95
énergie ($pJ/point$)							
$auto$	3.094	3.976	6.285	7.278	7.827	9.545	6.334
$ii = 1$	1.460	1.405	1.448	1.411	1.311	1.592	1.438
$ii = 2$	2.971	2.735	2.325	3.094	2.969	3.130	2.871
$ii = 3$	4.466	4.312	4.079	4.312	3.682	-	4.170
$ii = 4$	5.882	5.595	5.020	5.368	5.169	5.179	5.369
$ii = 5$	7.343	6.871	6.185	6.286	5.986	5.807	6.413
$ii = 6$	8.884	8.194	7.372	7.209	6.836	6.836	7.555
$ii = 7$	10.263	9.439	8.523	8.400	7.879	7.728	8.705
$ii = 8$	11.713	10.695	9.651	9.438	8.941	8.745	9.864

TABLE 22 – $\Sigma\Delta$ + mémoire DP + Hack : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2263	2513	2952	3041	2929	3143	2807
$ii = 1$	2254	2658	3038	3206	3523	3805	3081
$ii = 2$	2206	2920	3610	4377	4514	4466	3682
$ii = 3$	2212	2646	3186	3975	4022	4230	3379
$ii = 4$	2217	2653	3292	3571	3692	4238	3277
$ii = 5$	2245	2679	3051	3356	3496	3755	3097
$ii = 6$	2272	2706	3078	3096	3412	3547	3019
$ii = 7$	2252	2685	3058	3073	3164	3471	2951
$ii = 8$	2262	2697	3069	3070	3177	3528	2967
puissance ($\mu W/point$)							
$auto$	100.07	217.44	351.46	461.50	594.21	698.57	403.88
$ii = 1$	88.38	217.89	411.78	578.16	827.17	867.32	498.45
$ii = 2$	95.07	269.73	534.11	888.64	1156.08	1358.78	717.07
$ii = 3$	93.68	231.60	442.43	726.98	947.36	1160.03	600.35
$ii = 4$	92.20	219.95	429.51	628.71	858.70	1129.46	559.76
$ii = 5$	91.69	213.36	370.24	557.58	757.06	985.50	495.91
$ii = 6$	91.88	210.52	362.93	483.37	705.24	872.69	454.44
$ii = 7$	91.70	207.87	356.89	475.57	663.87	836.23	438.69
$ii = 8$	90.93	204.17	350.75	465.57	658.47	849.96	436.64
énergie ($pJ/point$)							
$auto$	2.010	3.271	5.868	6.932	8.329	10.488	6.150
$ii = 1$	0.898	1.116	1.406	1.492	1.720	1.480	1.352
$ii = 2$	1.910	2.719	3.604	4.532	4.716	4.619	3.683
$ii = 3$	2.818	3.484	4.448	5.510	5.745	5.877	4.647
$ii = 4$	3.696	4.408	5.750	6.325	6.911	7.573	5.777
$ii = 5$	4.592	5.343	6.181	6.993	7.595	8.252	6.493
$ii = 6$	5.521	6.325	7.269	7.261	8.486	8.751	7.269
$ii = 7$	6.427	7.284	8.338	8.333	9.306	9.779	8.244
$ii = 8$	7.282	8.176	9.363	9.321	10.547	11.356	9.341

TABLE 23 – $\Sigma\Delta$ + 2 mémoire SP + Hack : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2392	2860	2829	3175	3155	3178	2932
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	2301	2280	2510	3270	3188	3570	2853
$ii = 3$	2313	2885	2937	3082	3147	3217	2930
$ii = 4$	2317	2891	2883	3098	3189	3277	2943
$ii = 5$	2345	2919	2905	3214	3379	3317	3013
$ii = 6$	2370	2946	2935	3243	3360	3290	3024
$ii = 7$	2349	2935	2921	3222	3325	3345	3016
$ii = 8$	2361	2948	2927	3233	3328	3477	3046
puissance (μW/point)							
$auto$	155.68	291.98	362.12	505.94	571.20	692.93	429.98
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	148.15	226.47	359.02	571.81	659.36	857.08	470.32
$ii = 3$	149.17	307.61	415.67	543.9	640.71	753.34	468.40
$ii = 4$	147.79	301.29	361.31	488.07	588.58	769.30	442.72
$ii = 5$	148.12	297.13	357.35	542.76	658.53	778.69	463.76
$ii = 6$	149.35	296.24	355.26	538.17	671.53	736.41	457.83
$ii = 7$	148.09	294.00	353.13	531.68	648.64	769.38	457.49
$ii = 8$	148.09	291.55	346.98	526.56	645.03	797.14	459.23
énergie (pJ/point)							
$auto$	3.127	4.392	4.839	6.335	6.864	8.094	5.608
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	2.976	2.274	2.404	2.882	2.659	2.880	2.679
$ii = 3$	4.488	4.627	4.169	4.091	3.855	3.777	4.168
$ii = 4$	5.924	6.039	4.828	4.891	4.719	5.140	5.257
$ii = 5$	7.419	7.441	5.966	6.796	6.597	6.500	6.786
$ii = 6$	8.974	8.900	7.115	8.084	8.070	7.375	8.086
$ii = 7$	10.379	10.303	8.250	9.316	9.092	8.987	9.388
$ii = 8$	11.860	11.674	9.263	10.542	10.332	10.640	10.718

TABLE 24 – $\Sigma\Delta$ + mémoire SP + Flag : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2493	2970	2947	3272	3255	3244	3030
$ii = 1$	2489	2944	3198	3276	3927	3831	3278
$ii = 2$	2423	2418	2636	2925	2810	3153	2728
$ii = 3$	2429	2927	2998	3113	3091	3253	2969
$ii = 4$	2435	2935	2915	3114	3209	3227	2973
$ii = 5$	2462	2961	2936	3217	3312	3252	3023
$ii = 6$	2490	2989	2972	3254	3244	3247	3033
$ii = 7$	2467	2968	2948	3230	3211	3226	3008
$ii = 8$	2479	2980	2964	3242	3224	3237	3021
puissance ($\mu W/point$)							
$auto$	160.39	300.18	361.10	506.40	578.92	674.39	430.23
$ii = 1$	149.69	280.81	414.47	543.13	862.75	944.03	532.48
$ii = 2$	154.04	236.32	368.22	507.20	562.80	736.16	427.46
$ii = 3$	154.36	303.40	413.12	553.36	616.99	742.31	463.92
$ii = 4$	152.69	297.47	354.20	476.24	587.01	726.95	432.43
$ii = 5$	152.51	293.47	347.71	508.33	617.02	729.77	441.47
$ii = 6$	153.63	292.40	346.47	502.12	596.62	687.36	429.77
$ii = 7$	152.22	289.30	343.24	494.31	575.28	681.04	422.57
$ii = 8$	152.16	287.32	339.35	488.81	567.71	673.31	418.11
énergie ($pJ/point$)							
$auto$	3.222	4.516	4.825	6.341	6.957	7.877	5.623
$ii = 1$	1.521	1.438	1.426	1.401	1.808	1.636	1.538
$ii = 2$	3.094	2.373	2.465	2.557	2.270	2.474	2.539
$ii = 3$	4.644	4.564	4.143	4.162	3.712	3.722	4.158
$ii = 4$	6.121	5.962	4.733	4.773	4.706	4.857	5.192
$ii = 5$	7.639	7.349	5.805	6.365	6.181	6.092	6.572
$ii = 6$	9.231	8.784	6.939	7.543	7.170	6.883	7.758
$ii = 7$	10.668	10.138	8.019	8.661	8.064	7.955	8.917
$ii = 8$	12.186	11.505	9.059	9.787	9.093	8.987	10.103

TABLE 25 – $\Sigma\Delta$ + mémoire DP + Flag : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2339	2744	2857	3181	3044	3084	2875
$ii = 1$	2330	2736	2523	3366	3269	3380	2934
$ii = 2$	2280	3157	3414	3616	3675	4036	3363
$ii = 3$	2286	2878	3195	3667	4069	3622	3286
$ii = 4$	2293	2887	2951	3565	3733	3780	3202
$ii = 5$	2321	2915	2978	3393	3640	3555	3134
$ii = 6$	2346	2941	3006	3420	3606	3606	3154
$ii = 7$	2325	2921	2984	3399	3347	3334	3052
$ii = 8$	2337	2933	2995	3411	3371	3346	3066
puissance ($\mu W/point$)							
$auto$	151.14	282.38	357.24	495.51	543.97	671.15	416.90
$ii = 1$	143.14	261.11	326.80	609.59	664.71	841.88	474.54
$ii = 2$	145.06	337.33	490.37	656.77	832.10	1038.5	583.36
$ii = 3$	144.61	298.57	416.25	635.58	904.33	845.02	540.73
$ii = 4$	143.90	294.55	375.22	593.33	762.47	909.66	513.19
$ii = 5$	144.25	293.20	371.72	542.61	712.09	815.94	479.97
$ii = 6$	145.63	293.97	371.39	540.98	683.93	795.66	471.93
$ii = 7$	144.69	292.31	367.80	536.44	632.76	750.79	454.13
$ii = 8$	144.66	291.06	365.27	533.21	629.54	744.68	451.40
énergie ($pJ/point$)							
$auto$	3.036	4.248	4.773	6.204	6.537	7.840	5.440
$ii = 1$	1.455	1.337	1.107	1.573	1.372	1.448	1.382
$ii = 2$	2.914	3.401	3.309	3.324	3.369	3.517	3.305
$ii = 3$	4.351	4.491	4.185	4.805	5.484	4.259	4.596
$ii = 4$	5.768	5.904	5.014	5.958	6.137	6.101	5.813
$ii = 5$	7.225	7.343	6.206	6.794	7.144	6.822	6.922
$ii = 6$	8.750	8.832	7.438	8.126	8.230	7.978	8.226
$ii = 7$	10.141	10.243	8.592	9.399	8.869	8.770	9.336
$ii = 8$	11.585	11.655	9.751	10.676	10.083	9.940	10.615

TABLE 26 – $\Sigma\Delta$ + 2 mémoires SP + Flag : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

mémoire	SP	SP RW	DP	2×SP
optimisation	*Reg*	*Reg*	*Reg*	*Reg*
Sigmadelta Flag				
surface(*best S*)	2750	**2687**	2728	2808
surface(*best E*)	2853	3109	3278	2934
energie(*best S*)	3.95	5.21	2.54	4.32
energie(*best E*)	2.68	1.50	1.54	**1.38**
ii(*best E*)	2	1	1	1
Sigmadelta Delta				
surface(*best S*)	**2679**	2685	2788	2728
surface(*best E*)	2931	2949	3103	3065
energie(*best S*)	5.12	5.37	5.27	5.63
energie(*best E*)	2.75	**1.39**	1.40	1.44
ii(*best E*)	2	1	1	1
Sigmadelta Hack				
surface(*best S*)	2694	**2689**	2790	2797
surface(*best E*)	2948	2962	3175	3081
energie(*best S*)	5.48	5.18	5.13	6.13
energie(*best E*)	2.81	1.39	1.44	**1.35**
ii(*best E*)	2	1	1	1

TABLE 27 – ΣΔ : moyenne des surfaces et des énergies pour des fréquences de synthèse allant de 100 à 600 MHz par pas de 100 MHz

ANNEXE EROSION

Cette annexe présente les résultats détaillés de l'érosion 3×3 :

– érosion avec 1 mémoire SP et optimisation Reg (tab. 28),
– érosion avec 1 mémoire DP et optimisation Reg (tab. 29),

– érosion avec 1 mémoire SP et optimisation Rot (tab. 30),
– érosion avec 1 mémoire DP et optimisation Rot (tab. 31),
– érosion avec 3 mémoires SP entrelacées et optimisation Rot 32),
– érosion avec 1 mémoire SP et buffers lignes et optimisation Rot 33),

– érosion avec 1 mémoire SP et optimisation Red (tab. 34),
– érosion avec 1 mémoire DP et optimisation Red (tab. 35),
– érosion avec 3 mémoires SP entrelacées et optimisation Red (tab. 36),
– érosion avec 1 mémoire SP et buffers lignes et optimisation Red 37),

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2570	2570	2570	2570	2570	2570	2570
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	-	-	-	-	-	-	-
$ii = 4$	-	-	-	-	-	-	-
$ii = 5$	-	-	-	-	-	-	-
$ii = 6$	-	-	-	-	-	-	-
$ii = 7$	-	-	-	-	-	-	-
$ii = 8$	-	-	-	-	-	-	-
$ii = 9$	3091	3091	3091	3091	3091	3094	3092
puissance ($\mu W/point$)							
$auto$	178.59	271.60	366.48	460.42	554.37	648.23	413.28
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	-	-	-	-	-	-	-
$ii = 4$	-	-	-	-	-	-	-
$ii = 5$	-	-	-	-	-	-	-
$ii = 6$	-	-	-	-	-	-	-
$ii = 7$	-	-	-	-	-	-	-
$ii = 8$	-	-	-	-	-	-	-
$ii = 9$	224.21	346.38	470.52	593.69	716.84	821.02	528.78
énergie ($pJ/point$)							
$auto$	19.660	14.950	13.448	12.672	12.206	11.894	14.138
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	-	-	-	-	-	-	-
$ii = 4$	-	-	-	-	-	-	-
$ii = 5$	-	-	-	-	-	-	-
$ii = 6$	-	-	-	-	-	-	-
$ii = 7$	-	-	-	-	-	-	-
$ii = 8$	-	-	-	-	-	-	-
$ii = 9$	20.234	15.630	14.154	13.395	12.938	12.349	14.783

TABLE 28 – érosion + mémoire SP + Reg : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2893	2893	2893	2893	2893	2893	2893
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	-	-	-	-	-	-	-
$ii = 4$	-	-	-	-	-	-	-
$ii = 5$	3206	3206	3208	3208	3206	3206	3207
$ii = 6$	3097	3097	3097	3097	3097	3097	3097
$ii = 7$	2988	2988	2988	2988	2988	2988	2988
$ii = 8$	3001	3001	3001	3001	3001	3001	3001
$ii = 9$	3014	3014	3014	3014	3014	3014	3014
puissance ($\mu W/point$)							
$auto$	202.03	310.28	418.50	526.74	635.01	743.28	472.64
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	-	-	-	-	-	-	-
$ii = 4$	-	-	-	-	-	-	-
$ii = 5$	236.46	368.72	501.24	633.57	765.54	897.86	567.23
$ii = 6$	227.27	353.40	479.54	605.66	731.80	842.30	540.00
$ii = 7$	210.45	323.48	436.51	549.54	662.58	775.66	493.04
$ii = 8$	209.93	321.92	433.91	545.90	657.89	769.91	489.91
$ii = 9$	209.86	321.28	432.71	544.13	655.55	767.01	488.42
énergie ($pJ/point$)							
$auto$	14.160	10.873	9.777	9.229	8.901	8.682	10.271
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	-	-	-	-	-	-	-
$ii = 4$	-	-	-	-	-	-	-
$ii = 5$	11.881	9.263	8.395	7.959	7.693	7.519	8.785
$ii = 6$	13.674	10.631	9.617	9.110	8.806	8.446	10.048
$ii = 7$	14.750	11.336	10.198	9.629	9.288	9.061	10.710
$ii = 8$	16.813	12.891	11.584	10.930	10.538	10.277	12.172
$ii = 9$	18.906	14.472	12.994	12.255	11.811	11.516	13.659

TABLE 29 – érosion + mémoire DP + Reg : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2650	2650	2650	2650	2655	2681	2656
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	3289	3289	3289	3289	3289	3311	3293
$ii = 4$	3260	3260	3260	3260	3260	3294	3266
$ii = 5$	2940	2940	2940	2940	2942	2962	2944
$ii = 6$	2971	2971	2971	2971	2972	2997	2976
$ii = 7$	2984	2984	2984	2984	2988	3006	2988
$ii = 8$	2993	2993	2993	2993	2994	3007	2996
$ii = 9$	3008	3008	3009	3009	3008	3035	3013
puissance ($\mu W/point$)							
$auto$	174.49	266.36	359.83	452.49	540.46	634.50	404.69
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	236.21	367.96	499.71	631.46	757.43	888.84	563.60
$ii = 4$	234.02	364.16	494.29	624.43	748.69	876.46	557.01
$ii = 5$	202.80	311.59	420.40	529.19	637.93	741.18	473.85
$ii = 6$	204.22	312.97	421.72	530.46	639.26	746.09	475.79
$ii = 7$	204.19	312.51	420.83	529.16	635.96	740.65	473.88
$ii = 8$	203.45	311.57	419.70	527.83	636.08	740.73	473.23
$ii = 9$	204.33	312.12	419.97	527.77	634.14	740.92	473.21
énergie ($pJ/point$)							
$auto$	8.837	6.745	6.074	5.729	5.474	5.355	6.369
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	7.276	5.667	5.131	4.862	4.666	4.563	5.361
$ii = 4$	9.530	7.415	6.710	6.357	6.098	5.949	7.010
$ii = 5$	10.270	7.890	7.097	6.700	6.461	6.256	7.446
$ii = 6$	12.384	9.490	8.525	8.042	7.753	7.541	8.956
$ii = 7$	14.425	11.038	9.910	9.345	8.985	8.720	10.404
$ii = 8$	16.407	12.563	11.282	10.641	10.259	9.956	11.851
$ii = 9$	18.521	14.146	12.689	11.960	11.496	11.193	13.334

TABLE 30 – érosion + mémoire SP + *Rot* : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2910	2910	2910	2910	2910	2932	2914
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	3531	3531	3531	3531	3536	3556	3536
$ii = 3$	3504	3504	3504	3504	3506	3517	3507
$ii = 4$	2902	2902	2905	2905	2902	2919	2906
$ii = 5$	2918	2918	2921	2921	2918	2929	2921
$ii = 6$	2922	2922	2926	2926	2922	2931	2925
$ii = 7$	2915	2915	2918	2918	2915	2930	2919
$ii = 8$	2915	2915	2919	2919	2915	2923	2918
$ii = 9$	2959	2959	2963	2963	2959	2976	2963
puissance ($\mu W/point$)							
$auto$	198.13	306.08	414.03	521.99	629.81	731.61	466.94
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	257.54	406.15	554.74	703.32	845.31	996.37	627.24
$ii = 3$	253.46	398.77	543.97	689.25	834.74	972.77	615.49
$ii = 4$	195.61	301.99	409.21	515.70	621.13	723.75	461.23
$ii = 5$	194.45	299.26	404.88	509.79	613.67	717.53	456.60
$ii = 6$	193.38	296.93	401.27	504.91	607.57	708.41	452.08
$ii = 7$	192.92	296.11	400.09	503.38	605.70	705.21	450.57
$ii = 8$	191.91	294.08	397.03	499.28	600.57	700.04	447.15
$ii = 9$	193.67	296.05	399.22	501.69	603.20	703.63	449.58
énergie ($pJ/point$)							
$auto$	8.005	6.184	5.576	5.273	5.089	4.927	5.842
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	5.296	4.176	3.802	3.616	3.477	3.415	3.964
$ii = 3$	7.727	6.078	5.527	5.253	5.089	4.942	5.769
$ii = 4$	7.904	6.101	5.511	5.209	5.019	4.874	5.770
$ii = 5$	9.801	7.542	6.803	6.424	6.186	6.028	7.131
$ii = 6$	11.681	8.968	8.080	7.625	7.340	7.132	8.471
$ii = 7$	13.582	10.424	9.389	8.860	8.529	8.275	9.843
$ii = 8$	15.430	11.823	10.641	10.036	9.658	9.381	11.162
$ii = 9$	17.509	13.382	12.030	11.339	10.906	10.602	12.628

TABLE 31 – érosion + mémoire DP + *Rot* : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	3013	3013	3013	3013	3013	3013	3013
$ii = 1$	3035	3035	3035	3035	3035	3037	3035
$ii = 2$	3029	3029	3029	3029	3029	3029	3029
$ii = 3$	3032	3032	3032	3032	3032	3032	3032
$ii = 4$	3039	3039	3039	3039	3039	3039	3039
$ii = 5$	3062	3062	3063	3063	3062	3063	3063
$ii = 6$	3090	3090	3090	3090	3090	3090	3090
$ii = 7$	3069	3069	3069	3069	3069	3069	3069
$ii = 8$	3081	3081	3081	3081	3081	3081	3081
$ii = 9$	3096	3096	3096	3096	3096	3096	3096
puissance ($\mu W/point$)							
$auto$	210.02	326.10	442.17	558.25	674.33	790.42	500.22
$ii = 1$	195.45	296.49	397.57	498.59	599.60	700.68	448.06
$ii = 2$	203.58	312.81	422.04	531.28	640.51	749.89	476.69
$ii = 3$	200.96	307.36	413.76	520.17	626.58	732.97	466.97
$ii = 4$	199.01	303.12	407.24	511.36	615.47	719.60	459.30
$ii = 5$	198.41	301.63	404.64	507.78	611.29	714.09	456.31
$ii = 6$	199.30	302.10	404.91	507.71	610.51	713.31	456.31
$ii = 7$	198.21	333.75	403.26	505.82	608.35	710.89	460.05
$ii = 8$	197.67	299.26	400.84	502.43	604.02	705.61	451.64
$ii = 9$	197.83	299.09	400.35	501.61	602.87	704.13	450.98
énergie ($pJ/point$)							
$auto$	4.571	3.549	3.208	3.038	2.935	2.867	3.361
$ii = 1$	2.153	1.633	1.460	1.373	1.321	1.286	1.537
$ii = 2$	4.431	3.404	3.062	2.891	2.788	2.720	3.216
$ii = 3$	6.551	5.010	4.496	4.239	4.085	3.983	4.728
$ii = 4$	8.644	6.583	5.896	5.553	5.347	5.209	6.206
$ii = 5$	10.768	8.185	7.320	6.890	6.635	6.459	7.710
$ii = 6$	12.976	9.835	8.788	8.264	7.950	7.740	9.259
$ii = 7$	15.053	12.673	10.208	9.604	9.240	8.998	10.963
$ii = 8$	17.154	12.985	11.595	10.900	10.483	10.206	12.221
$ii = 9$	19.311	14.598	13.027	12.241	11.770	11.456	13.734

TABLE 32 – érosion + 3xmémoire SP + *Rot* : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	6605	6605	6605	6605	6605	6612	6606
$ii = 1$	7945	7945	7945	7945	7945	7962	7948
$ii = 2$	7204	7204	7204	7204	7225	7247	7215
$ii = 3$	7229	7229	7229	7229	7242	7280	7240
$ii = 4$	7265	7265	7265	7265	7290	7371	7287
$ii = 5$	7258	7258	7258	7258	7278	7312	7270
$ii = 6$	7277	7277	7277	7275	7301	7348	7293
$ii = 7$	7259	7259	7259	7259	7287	7312	7273
$ii = 8$	7261	7261	7261	7261	7295	7357	7283
$ii = 9$	7265	7265	7265	7265	7282	7368	7285
puissance ($\mu W/point$)							
$auto$	496.48	865.78	1235.08	1604.42	1973.69	2328.40	1417.31
$ii = 1$	589.31	1011.52	1433.73	1855.94	2278.15	2696.74	1644.23
$ii = 2$	533.60	925.89	1318.16	1710.44	2093.36	2496.43	1512.98
$ii = 3$	534.66	927.36	1320.04	1712.84	2096.04	2498.92	1514.98
$ii = 4$	536.73	929.86	1323.00	1716.26	2104.38	2509.63	1519.98
$ii = 5$	537.37	931.44	1325.49	1719.67	2102.19	2508.85	1520.84
$ii = 6$	537.72	931.40	1325.09	1711.99	2107.84	2516.92	1521.83
$ii = 7$	538.02	932.41	1326.79	1715.48	2104.24	2503.87	1520.14
$ii = 8$	536.47	929.27	1322.05	1714.95	2097.73	2509.15	1518.27
$ii = 9$	537.00	930.61	1324.23	1717.94	2100.05	2519.62	1521.58
énergie ($pJ/point$)							
$auto$	10.804	9.421	8.959	8.729	8.590	8.445	9.158
$ii = 1$	6.539	5.612	5.303	5.148	5.055	4.987	5.440
$ii = 2$	11.612	10.075	9.562	9.306	9.111	9.055	9.787
$ii = 3$	17.429	15.115	14.344	13.959	13.665	13.577	14.681
$ii = 4$	23.312	20.194	19.154	18.636	18.280	18.167	19.624
$ii = 5$	29.163	25.275	23.978	23.332	22.817	22.692	24.543
$ii = 6$	35.009	30.320	28.757	27.865	27.447	27.311	29.451
$ii = 7$	40.858	35.404	33.586	32.569	31.960	31.691	34.345
$ii = 8$	46.554	40.320	38.242	37.205	36.407	36.290	39.170
$ii = 9$	52.419	45.420	43.088	41.924	40.999	40.992	44.140

TABLE 33 – érosion + mémoire SP + 2xbufferligne + Rot : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2013	2013	2013	2013	2018	2045	2019
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	2253	2253	2253	2253	2253	2268	2256
$ii = 4$	2225	2225	2225	2225	2225	2241	2228
$ii = 5$	2069	2069	2069	2069	2069	2078	2071
$ii = 6$	2097	2097	2097	2097	2097	2120	2101
$ii = 7$	2112	2112	2112	2112	2112	2133	2116
$ii = 8$	2115	2115	2115	2115	2115	2128	2117
$ii = 9$	2135	2135	2135	2135	2135	2160	2139
puissance ($\mu W/point$)							
$auto$	130.84	198.15	266.86	334.87	398.18	467.89	299.47
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	153.53	236.29	319.04	401.77	481.75	558.60	358.50
$ii = 4$	151.49	233.01	314.56	396.12	471.14	552.27	353.10
$ii = 5$	137.34	209.19	281.04	352.88	424.74	490.12	315.89
$ii = 6$	138.94	211.08	283.21	355.35	427.48	493.87	318.32
$ii = 7$	139.29	211.21	283.11	355.03	425.13	492.51	317.71
$ii = 8$	138.70	210.58	282.44	354.30	426.16	490.76	317.16
$ii = 9$	139.57	211.27	282.97	354.68	425.35	496.61	318.41
énergie ($pJ/point$)							
$auto$	6.626	5.017	4.505	4.240	4.033	3.949	4.728
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	-	-	-	-	-	-	-
$ii = 3$	4.729	3.639	3.276	3.094	2.968	2.868	3.429
$ii = 4$	6.169	4.744	4.270	4.033	3.837	3.748	4.467
$ii = 5$	6.955	5.297	4.744	4.468	4.302	4.137	4.984
$ii = 6$	8.426	6.400	5.725	5.387	5.185	4.992	6.019
$ii = 7$	9.840	7.460	6.667	6.270	6.006	5.799	7.007
$ii = 8$	11.185	8.491	7.592	7.143	6.873	6.596	7.980
$ii = 9$	12.651	9.575	8.550	8.037	7.711	7.502	9.004

TABLE 34 – érosion + mémoire SP + Red : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2388	2388	2388	2388	2389	2407	2391
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	2685	2685	2685	2685	2687	2714	2690
$ii = 3$	2627	2627	2627	2627	2625	2639	2629
$ii = 4$	2374	2374	2378	2378	2374	2400	2380
$ii = 5$	2379	2379	2383	2383	2379	2402	2384
$ii = 6$	2383	2383	2387	2387	2383	2406	2388
$ii = 7$	2377	2377	2381	2381	2377	2397	2382
$ii = 8$	2384	2384	2388	2388	2384	2400	2388
$ii = 9$	2424	2424	2427	2427	2424	2442	2428
puissance ($\mu W/point$)							
$auto$	159.14	243.12	327.10	411.07	495.13	573.64	368.20
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	190.15	295.00	399.84	504.69	606.54	711.07	451.22
$ii = 3$	183.46	283.59	383.71	483.84	577.71	675.31	431.27
$ii = 4$	159.40	243.30	328.09	412.13	495.02	574.43	368.73
$ii = 5$	157.96	240.42	323.75	406.33	487.82	556.70	362.16
$ii = 6$	156.90	238.24	320.42	401.86	482.24	559.26	359.82
$ii = 7$	156.70	237.91	319.05	401.26	481.55	548.00	357.41
$ii = 8$	156.18	236.51	317.66	398.09	477.50	553.19	356.52
$ii = 9$	157.60	238.15	319.52	400.17	479.80	556.83	358.68
énergie ($pJ/point$)							
$auto$	6.430	4.912	4.405	4.152	4.001	3.863	4.627
$ii = 1$	-	-	-	-	-	-	-
$ii = 2$	3.910	3.033	2.741	2.595	2.495	2.437	2.868
$ii = 3$	5.593	4.322	3.899	3.687	3.522	3.431	4.076
$ii = 4$	6.441	4.915	4.419	4.163	4.000	3.868	4.634
$ii = 5$	7.962	6.059	5.440	5.120	4.918	4.677	5.696
$ii = 6$	9.478	7.195	6.452	6.069	5.826	5.630	6.775
$ii = 7$	11.032	8.375	7.488	7.063	6.781	6.430	7.861
$ii = 8$	12.558	9.508	8.514	8.002	7.679	7.413	8.946
$ii = 9$	14.248	10.765	9.629	9.044	8.675	8.390	10.125

TABLE 35 – érosion + mémoire DP + Red : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	2393	2393	2393	2393	2393	2393	2393
$ii = 1$	2389	2389	2389	2389	2389	2391	2389
$ii = 2$	2374	2374	2374	2374	2374	2374	2374
$ii = 3$	2377	2377	2377	2377	2377	2377	2377
$ii = 4$	2384	2384	2384	2384	2384	2384	2384
$ii = 5$	2415	2415	2415	2415	2415	2415	2415
$ii = 6$	2447	2447	2447	2447	2447	2447	2447
$ii = 7$	2422	2422	2422	2422	2422	2422	2422
$ii = 8$	2438	2438	2438	2438	2438	2438	2438
$ii = 9$	2453	2453	2453	2453	2453	2453	2453
puissance ($\mu W/point$)							
$auto$	160.96	247.71	334.46	421.21	507.96	594.71	377.84
$ii = 1$	150.69	226.53	302.39	378.24	454.08	529.95	340.31
$ii = 2$	154.38	235.63	316.89	398.14	479.39	560.64	357.51
$ii = 3$	152.72	232.15	311.58	391.00	470.42	549.85	351.29
$ii = 4$	151.56	229.44	307.33	385.22	463.10	540.99	346.27
$ii = 5$	153.69	231.83	309.99	388.14	466.28	544.43	349.06
$ii = 6$	155.18	233.34	311.50	389.66	467.82	545.97	350.58
$ii = 7$	153.93	231.88	309.81	387.78	465.72	543.67	348.80
$ii = 8$	153.94	231.28	308.62	385.96	463.29	540.62	347.29
$ii = 9$	154.26	231.42	308.57	385.62	462.87	540.02	347.13
énergie ($pJ/point$)							
$auto$	3.503	2.696	2.426	2.292	2.211	2.157	2.548
$ii = 1$	1.660	1.247	1.110	1.041	1.000	0.973	1.172
$ii = 2$	3.360	2.564	2.299	2.166	2.087	2.034	2.418
$ii = 3$	4.979	3.784	3.386	3.187	3.067	2.988	3.565
$ii = 4$	6.583	4.983	4.450	4.183	4.023	3.916	4.690
$ii = 5$	8.341	6.291	5.608	5.266	5.061	4.925	5.915
$ii = 6$	10.104	7.596	6.760	6.343	6.092	5.925	7.137
$ii = 7$	11.690	8.805	7.843	7.362	7.074	6.881	8.276
$ii = 8$	13.359	10.035	8.927	8.373	8.041	7.819	9.426
$ii = 9$	15.058	11.295	10.040	9.411	9.037	8.786	10.605

TABLE 36 – érosion + 3xmémoire SP + Red : surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

fréquence (MHz)	100	200	300	400	500	600	moyenne
surface (μm^2)							
$auto$	5939	5939	5939	5939	5939	5956	5942
$ii = 1$	6904	6904	6904	6904	6904	6928	6908
$ii = 2$	6519	6519	6519	6519	6519	6550	6524
$ii = 3$	6541	6541	6541	6541	6541	6567	6545
$ii = 4$	6575	6575	6575	6575	6575	6609	6581
$ii = 5$	6571	6571	6571	6571	6571	6607	6577
$ii = 6$	6589	6589	6589	6589	6598	6680	6606
$ii = 7$	6578	6578	6578	6578	6585	6621	6586
$ii = 8$	6583	6583	6583	6583	6596	6637	6594
$ii = 9$	6578	6578	6578	6578	6578	6607	6583
puissance ($\mu W/point$)							
$auto$	447.70	791.39	1135.07	1478.80	1822.43	2160.91	1306.05
$ii = 1$	513.09	892.04	1270.99	1649.95	2028.90	2408.59	1460.59
$ii = 2$	485.05	851.46	1217.86	1584.28	1945.00	2329.76	1402.24
$ii = 3$	475.94	852.76	1219.59	1586.41	1953.22	2329.87	1402.97
$ii = 4$	487.93	855.20	1222.47	1589.73	1952.75	2335.23	1407.22
$ii = 5$	488.84	857.02	1225.21	1593.40	1961.58	2331.35	1409.57
$ii = 6$	489.17	857.00	1224.81	1592.63	1958.23	2351.23	1412.18
$ii = 7$	489.88	858.59	1227.32	1596.04	1960.52	2344.00	1412.73
$ii = 8$	488.37	855.42	1222.63	1589.76	1953.26	2344.29	1408.96
$ii = 9$	488.49	856.24	1224.00	1591.75	1959.51	2327.09	1407.85
énergie ($pJ/point$)							
$auto$	9.743	8.611	8.234	8.045	7.932	7.838	8.400
$ii = 1$	5.693	4.949	4.701	4.577	4.502	4.454	4.813
$ii = 2$	10.556	9.265	8.834	8.619	8.465	8.450	9.032
$ii = 3$	15.515	13.899	13.252	12.928	12.734	12.658	13.498
$ii = 4$	21.193	18.572	17.699	17.262	16.963	16.905	18.099
$ii = 5$	26.529	23.255	22.164	21.618	21.291	21.087	22.657
$ii = 6$	31.848	27.898	26.581	25.922	25.498	25.513	27.210
$ii = 7$	37.202	32.601	31.068	30.302	29.777	29.668	31.770
$ii = 8$	42.380	37.116	35.366	34.489	33.900	33.905	36.193
$ii = 9$	47.683	41.790	39.826	38.844	38.255	37.859	40.710

TABLE 37 – érosion + mémoire SP + 2xbufferligne + Red surface puissance et énergie pour des fréquences de synthèse $\in [100 : 600]$ par pas de 100 MHz

BIBLIOGRAPHIE

[1] M.M. Abutaleb, A. Hamdy, M.E. Abuelwafa, and E.M. Saad. Fpga-based object extraction based on multimodal sigma-delta background estimation. In *International Conference on Computer Control*, pages 1–7, 2008.

[2] R. Allen and K. Kennedy, editors. *Optimizing compilers for modern architectures : a dependence-based approach*, chapter 8,9,11. Morgan Kaufmann, 2002.

[3] Ivan Augé, Frédéric Pétrot, François Donnet, and Pascal Gomez. Platform-based design from parallel c specifications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(12) :1811–1826, 2005.

[4] Marcus Bednara and Jürgen Teich. Automatic synthesis of fpga processor arrays from loop algorithms. *The Journal of Supercomputing*, 26(2) :149–165, 2003.

[5] Critical Blue. Boosting software processing performance with coprocessor synthesis `http://www.criticalblue.com`.

[6] Bluespec. Bluespec esl synthesis extension (ese) to systemc `http://www.bluespec.com/index.htm`.

[7] P. Boulet. Array-ol revisited, multidimensional intensive signal processing specification. Research Report 6113, INRIA, 2008.

[8] calypto. Catapult-c `http://www.calypto.com`.

[9] J. F. Canny. A computation approach to edge detection. *Pattern Analysis Machine Intelligence*, 8,6 :679–698, 1986.

[10] Emmanuel Casseau and Bertrand Le Gal. High-level synthesis for the design of fpga-based signal processing systems. In *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS'09. International Symposium on*, pages 25–32. IEEE, 2009.

[11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High level synthesis for fpgas : From prototyping to deployment. In *Transactions on Computer-Aided design of integrated circuits and systems*, volume 30,4, pages 746–749. IEEE, 2011.

[12] Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. Platform-based behavior-level and system-level synthesis. In *SOC Conference, 2006 IEEE International*, pages 199–202. IEEE, 2006.

[13] C. Consel, J.L. Lawall, and A.-F. Le Meur. A tour of Tempo : A program specializer for the C language. *Science of Computer Programming*, 2004.

[14] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. Gaut : A high-level synthesis tool for dsp applications. In *High-Level Synthesis*, pages 147–169. Springer, 2008.

[15] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *Design & Test of Computers, IEEE*, 26(4) :8–17, 2009.

[16] Philippe Coussy and Andres Takach. Special issue on high-level synthesis. *Design & Test of Computers, IEEE*, 25(5) :393–393, 2008.

[17] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems*, pages 53–69, 2009.

[18] D. Demigny, editor. *Méthodes et Architectures pour le TSI en temps réel*, chapter 3 : optimisations algorithmiques. Hermes, 2001.

[19] D. Demigny, editor. *Méthodes et Architectures pour le TSI en temps réel*, chapter 4 : sur la précision des calculs. Hermes, 2001.

[20] J. Denoulet, G. Mostafaoui, L. Lacassagne, and A. Mérigot. Implementing motion markov detection on general purpose processor and associative mesh. In *Computer Architecture and Machine Perception*. IEEE, 2005.

[21] R. Deriche. Fast algorithms for low-level vision. In *International Conference On Pattern Recognition*, pages 434–438. IEEE, 1988.

[22] R. Deriche. Fast algorithms for low-level vision. *Transaction on Pattern Analysis*, 12,1 :78–87, 1990.

[23] T. Duff. `http://en.wikipedia.org/wiki/Duff's_device`.

[24] M. Fingeroff and T. Bollaert, editors. *High-Level Synthesis - Blue Book*, chapter 4, pages 41–44. Mentor Graphic, 2010.

[25] Motorola / Freescale. Altivec `http://www.freescale.com/webapp/sps/site/overview.jsp?code=DRPPCALTVC`.

[26] Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4) :381–391, 1999.

[27] L. Lacassagne G. Mostafaoui, T. Kunlin. Relaxation markovienne et seuillage par hystérésis pour une détection de mouvement temps réel dans des sequences d'images. In *ISIVC : International Symposium on Image/Video Communications over fixed and mobile networks*, 2004.

[28] E. Goubault, M. Martel, and S. Putot. Fluctuat `http://www.di.ens.fr/~cousot/projects/DAEDALUS/synthetic_summary/CEA/Fluctuat/`.

[29] Sumit Gupta. *SPARK : : A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer, 2004.

[30] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alexandru Nicolau. Spark : A high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461–466. IEEE, 2003.

[31] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization : an overview of the pips project. In *ICS '91 : Proceedings of the 5th international conference on Supercomputing*, pages 244–251, New York, NY, USA, 1991. ACM.

[32] Y. Janin, V. Bertin, H. Chauvet, T. Deruyter, C. Eichwald, O.-A. Giraud, V. Lorquet, and T. Thery. Designing tightly-coupled extension units for the stxp70 processor. In *Design, Automation and Test in Europe Conferece and Exhibition (DATE)*, pages 1052–1053. IEEE, 2013.

[33] Neil D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3) :480–503, 1996.

[34] J.-O. Klein, H. Mathias L. Lacassagne, S. Moutault, and A. Dupret. Low power image processing : Analog versus digital comparison. In *Computer Architecture Machine Perception*. IEEE, 2005.

[35] Manjunath Kudlur, Kevin Fan, and Scott Mahlke. Streamroller : : automatic synthesis of prescribed throughput accelerator pipelines. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 270–275. ACM, 2006.

[36] L. Lacassagne, F. Lohier, and P. Garda. *Méthodes et Architectures pour le TSI en temps réel*, chapter 8 : Optimisation logicielle pour processeurs superscalaires. Hermes, 2001.

[37] L. Lacassagne, F. Lohier, and P. Garda. *Méthodes et Architectures pour le TSI en temps réel*, chapter 10 : Optimisation logicielle pour processeurs VLIW. Hermes, 2001.

[38] L. Lacassagne, A. Manzanera, J. Denoulet, and A. Mérigot. High performance motion detection : Some trends toward new embedded architectures for vision systems. *Journal of Real Time Image Processing*, pages 127–148, october 2008.

[39] L. Lacassagne, A. Manzanera, and A. Dupret. Motion detection : fast and robust algorithms for embedded systems. In *International Conference on Image Analysis and Processing (ICIP)*, pages 3265–3268, 2009.

[40] L. Lacassagne and A. B. Zavidovique. Light speed labeling for risc architectures. In *International Conference on Image Analysis and Processing (ICIP)*, 2009.

[41] Bertrand Le Gal and Emmanuel Casseau. Word-length aware dsp hardware design flow based on high-level synthesis. *Journal of Signal Processing Systems*, 62(3) :341–357, 2011.

[42] Youn-Long Lin. Recent developments in high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2(1) :2–21, 1997.

[43] F. Garcia Lorca, L. Kessal, and D. demigny. Efficient asic and fpga implementations of iir filters for real time edge detection. In *International Conference on Image Processing*, pages 406–409. IEEE, 1997.

[44] A. Manzanera. Sigma-delta background subtraction and the zipf law. In *CIARP*, volume 28-2, pages 42–51. LNCS, 2007.

[45] A. Manzanera and J. Richefeu. robust and computationally efficient motion detection algorithm based on sigma-delta background estimation. In *ICVGIP*. IEEE, 2004.

[46] A. Manzanera and J. Richefeu. A new motion detection algorithm based on sigma-delta background estimation. *Pattern Recognition Letters*, 28-2 :320–328, february 2007.

[47] Grant Martin and Gary Smith. High-level synthesis : Past, present, and future. *Design & Test of Computers, IEEE*, 26(4) :18–25, 2009.

[48] D. Menard, R. Rocher, O. Sentieys, N. Simon, L.-S. Didier, T. Hilaire, B. Lopez, E. Goubault, S. Putot, F. Védrine, A. Najahi, G. Revy, L. Fangain, C. Samoyeau, F. Lemonnier, and C. Clienti. Design of fixed-point embedded systems (defis) french anr project. In *Design and Architectures for Signal and Image Processing*, pages 1–8, 2012.

[49] D. Menard, R. Serizel, R. Rocher1, and O. Sentieys. Accuracy constraint determination in fixed-point system design. *Journal on Embedded Systems (JES),*, 2008 :1–12, 2008.

[50] G. Mostafaoui, C. Achard, M. Milgram, and L. Lacassagne. Extraction de trajectoires basées sur la cinématique dans les séquences d'images. In *GRETSI*, 2003.

[51] A Mozipo, Daniel Massicotte, Patrice Quinton, and Tanguy Risset. Automatic synthesis of a parallel architecture for kalman filtering using mmalpha. In *International conference on parallel computing in electrical engineering (PARELEC 98)*, pages 201–206, 1999.

[52] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Efficient automated synthesis, programing, and implementation of multi-processor platforms on fpga chips. In *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*, pages 1–6. IEEE, 2006.

[53] M. Piccardi. Background subtraction techniques : a review. In *Conference on Systems, Man and Cybernetics*, volume 4, pages 3099–3104. IEEE, 2004.

[54] Robert Schreiber, Shail Aditya, B Ramakrishna Rau, Vinod Kathail, Scott Mahlke, Santosh Abraham, and Greg Snider. High-level synthesis of nonprogrammable hardware accelerators. In *Application-Specific Systems, Architectures, and Processors, 2000. Proceedings. IEEE International Conference on*, pages 113–124. IEEE, 2000.

[55] O Sentieys, J Ph Diguet, and JL Philippe. Gaut : a high level synthesis tool dedicated to real time signal processing application. In *European Design Automation Conference*, 2000.

[56] Jeremy G. Siek and Andrew Lumsdaine. Concept checking : Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, 2000.

[57] C. Stauffer and E. Grimson. Learning patterns of activity using real-time tracking. In *Transactions on PAMI*, volume 22-8, pages 747–757. IEEE, 2000.

[58] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprette. System design using khan process networks : the compaan/laura approach. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, pages 340–345. IEEE, 2004.

[59] synopsys. C-to-silicon `http://www.cadence.com`.

[60] synopsys. Symphony-c-compiler `http://www.synopsys.com`.

[61] Forte Design Systems. Cynthesize closes the esl-to-silicon gap `http://www.forteds.com/products/cynthesizer.asp`.

[62] W. Taha. Metaocaml – a compiled, type-safe multi-stage programming language. Available online from http ://www.metaocaml.org/, March 2003.

[63] S. L. Toral, F. Barrero, and M. Vargas. Development of an embedded vision based vehicle detection system using an arm video processor. In *Conference on Intelligent Transportation Systems*, pages 292–297. IEEE, 2008.

[64] E. Unruh. Prime number computation. Technical Report ANSI X3J16-94-0075/ISO WG21-462., C++ Standard Commitee, 1994.

[65] M. Vargas, S. L. Toral, F. Barrero, and J. M. Milla. An enhanced background estimation algorithm for vehicle detection in urban traffic video. In *Conference on Intelligent Transportation Systems*, pages 784–790. IEEE, 2008.

[66] Todd L. Veldhuizen. C++ templates are turing complete. Technical report, 2000.

[67] A. Verdant, A. Dupret, H. Mathias, P. Villard, and L. Lacassagne. Low power motion detection with low spatial and temporal resolution for cmos image sensor. In *Computer Architecture, Machine Perception and Sensors*, pages 12–18. IEEE, 2006.

[68] A. Verdant, A. Dupret, H. Mathias, P. Villard, and L. Lacassagne. Adaptive multiresolution for low power cmos image sensor. In *International Conference on Image Processing*, pages 185–188. IEEE, 2007.

[69] Robert A Walker and Raul Camposano. *A survey of high-level synthesis systems*. Springer, 1991.

[70] Stephen Wood, David Akehurst, Gareth Howells, and Klaus McDonald-Maier. Array ol descriptions of repetitive structures in vhdl. In *European Conference on Model Driven Architecture, Foundations and Applications*, pages 137–152. Springer, 2008.

[71] H. Ye, L. Lacassagne, J. Falcou, D. Etiemble, L. Cabaret, and O. Florent. High level transforms to reduce energy consumption of signal and image processing operators. In *IEEE International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 247–254, 2013.