Spéculation temporelle pour accélérateurs matériels

Thibaut Marty

To cite this version:

Thibaut Marty. Spéculation temporelle pour accélérateurs matériels. Architectures Matérielles [cs.AR]. Université de Rennes, 2022. Français. NNT : 2022REN1S052 . tel-03927199

HAL Id: tel-03927199
https://theses.hal.science/tel-03927199v1

Submitted on 6 Jan 2023

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

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

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Thibaut MARTY

Spéculation temporelle pour accélérateurs matériels

Thèse présentée et soutenue à Rennes, le 24/03/2022
Unité de recherche : IRISA

Rapporteurs avant soutenance :

Florent de DINECHIN	Professeur des universités, INSA Lyon, France
Alberto BOSIO	Professeur des universités, École Centrale de Lyon, France

Composition du Jury :

Examinateurs :	Florent de DINECHIN	Professeur des universités, INSA Lyon, France
	Alberto BOSIO	Professeur des universités, École Centrale de Lyon, France
	Caroline COLLANGE	Chargée de recherche, Centre Inria de l'Université de Rennes, France
	Roselyne CHOTIN	Maîtresse de conférences, Sorbonne Université, France
Dir. de thèse :	Steven DERRIEN	Professeur des universités, Université de Rennes 1, France

Table des matières

Introduction **7**

Contributions . . . 11

Plan du document . . . 11

1 Contexte **13**

1.1 Compromis performance/puissance . . . 13

1.1.1 Puissance et performance d'un transistor . . . 14

1.1.2 Gestion dynamique de la fréquence et de la tension . . . 15

1.2 Impact de la microarchitecture des processeurs sur les performances et l'énergie . . . 16

1.2.1 Parallélisme d'instruction . . . 17

1.2.2 Ordonnancement dynamique . . . 18

1.2.3 Exécution spéculative . . . 19

1.2.4 Jeu d'instructions vectoriel . . . 20

1.2.5 Hiérarchie mémoire . . . 21

1.2.6 Limites . . . 23

1.3 Architectures explicitement parallèles . . . 25

1.3.1 Processeur à jeu d'instruction explicitement parallèle . . . 26

1.3.2 Processeur graphique . . . 27

1.3.3 Processeur spécialisé pour l'intelligence artificielle . . . 29

1.3.4 Discussion . . . 30

1.4 Architectures dédiées . . . 31

1.4.1 Architecture des FPGA . . . 31

1.4.2 Modèle de programmation . . . 33

1.4.3 Exploration de l'espace de conception . . . 34

1.4.4 Intégration . . . 35

1.5 Conclusion . . . 35

2 Spéculation temporelle pour FPGA **37**
2.1 Spéculation temporelle . . . 37
2.1.1 Fréquence maximale d'un circuit . . . 38
2.1.2 *Overclocking* et *Undervolting* . . . 38
2.1.3 Variabilité . . . 41
2.2 Tolérance aux fautes . . . 43
2.2.1 Fautes, erreurs, pannes . . . 43
2.2.2 Phénomènes physiques pouvant provoquer une faute . . . 44
2.2.3 Erreurs temporelles . . . 46
2.2.4 Comparaison des mécanismes de tolérance aux fautes existants . . . 46
2.2.4.1 Redondance : duplication et triplication . . . 47
2.2.4.2 Codes correcteurs d'erreur . . . 47
2.2.4.3 Redondance basée sur l'arithmétique modulaire . . . 49
2.2.4.4 Détection basée sur le double échantillonnage . . . 52
2.2.4.5 Tolérance aux fautes logicielle . . . 59
2.2.4.6 Tolérance aux fautes au niveau algorithmique . . . 62
2.2.4.7 Synthèse . . . 66
2.3 Conclusion . . . 69

3 Réseaux de neurones convolutifs et FPGA **71**
3.1 Réseaux de neurones convolutifs . . . 71
3.1.1 Architecture . . . 72
3.1.2 Types de couches . . . 74
3.1.3 Optimisations algorithmiques . . . 78
3.1.4 Représentation des données . . . 83
3.1.5 Mises en œuvre sur FPGA . . . 86
3.2 Spéculation temporelle pour accélérateurs de convolution . . . 90
3.3 Impact des erreurs sur la précision . . . 91
3.4 Détection d'erreur pour CNN . . . 93
3.5 Conclusion . . . 93

4 Spéculation temporelle sûre pour réseaux de neurones convolutifs **95**
4.1 ABFT pour les CNN . . . 95
4.1.1 Vue d'ensemble . . . 95
4.1.2 ABFT pour convolution 1D . . . 97

4.1.3 ABFT pour convolution 2D 99
4.1.4 ABFT pour sommes de convolution 2D (CNN) 101
4.1.5 ABFT pour convolution avec pas non unitaire 102
4.1.6 Autres couches de CNN 104
4.2 Accélérateur de convolution tolérant aux fautes 104
4.2.1 Accélérateur de référence 104
4.2.2 Détection d’erreur pour un accélérateur de convolution 105
4.2.3 Mise en œuvre des calculs de sommes de contrôle 107
4.2.3.1 Somme de contrôle de sortie 107
4.2.3.2 Somme de contrôle d’entrée 108
4.2.4 Intégration complète 110
4.2.4.1 Gestion de la fréquence 110
4.2.4.2 Architectures en *streaming* et moteur de calcul unique . . . 112
4.3 Évaluation 112
4.3.1 Gains en performance et pénalités dues aux erreurs 113
4.3.2 Correction de la détection d’erreur 115
4.3.3 Convolution convertie en multiplication de matrice 118
4.3.4 Plateforme expérimentale 119
4.3.5 Amélioration des performances 120
4.3.6 Amélioration de l’efficacité énergétique 120
4.3.7 Gains en performance, en efficacité énergétique et surcoût dû à la détection d’erreur 121
4.3.8 Erreurs observées 124
4.3.9 Faux négatifs 127
4.4 Discussion 127
4.4.1 *Overclocking* et *undervolting* 128
4.4.2 Représentation des données 128
4.4.3 Variations algorithmiques de la convolution 129
4.4.4 Applicabilité à d’autres architectures 130
4.4.5 Applicabilité à d’autres algorithmes et génération automatique . . . 131
4.4.6 Limites 132
4.5 Conclusion 133

Conclusion et travaux futurs **135**
Perspectives de recherche 136
Liste des publications 137

Bibliographie **139**

Introduction

L'histoire a été marquée par des inventions technologiques majeures sans lesquelles l'informatique ne serait pas la même aujourd'hui. Ainsi, certains citent le XIXe siècle comme date de naissance de l'ordinateur avec la machine analytique de Babbage, tandis que d'autres remontent à l'invention du métier Jacquard au XVIIIe siècle ou encore la pascaline de Blaise Pascal au XVIIe. Quoi qu'il en soit, l'invention du transistor en 1947 puis du circuit intégré dix ans plus tard a révolutionné le développement du matériel informatique moderne.

Alors que le transistor de 1947 mesurait plusieurs centimètres de haut, les transistors des circuits intégrés sont aujourd'hui produits avec une finesse de gravure de quelques nanomètres (soit l'équivalent de seulement quelques dizaines d'atomes de silicium !). La réduction progressive de leur taille a permis d'en intégrer de plus en plus au sein d'un même circuit électronique. Moore quantifiait cette augmentation dans sa loi empirique : le nombre de transistors d'un microprocesseur, l'élément au cœur des ordinateurs, doublait tous les deux ans, suivant donc une loi exponentielle.

La réduction de la taille des transistors leur permet également d'être plus rapides et de consommer moins d'énergie. Les progrès en microélectronique offraient donc aux concepteurs d'ordinateurs un triple gain : un plus grand nombre de transistors, des transistors moins gourmands en énergie et une fréquence de fonctionnement plus élevée. Dennard observa en 1974 que l'évolution de la taille des transistors et de leur consommation se compensaient de telle sorte que la puissance électrique nécessaire pour faire fonctionner un processeur n'évoluait pas au cours des différentes générations. La densité de puissance, la puissance dissipée par surface, restait constante. Chaque nouvelle génération de processeurs pouvait donc réaliser des calculs plus rapidement tout en consommant moins d'énergie par calcul. Les performances et l'efficacité énergétique des processeurs s'amélioraient génération après génération.

Néanmoins, à partir d'un certain seuil, la miniaturisation des transistors ne permet plus de garder une puissance constante parce que les courants de fuite deviennent significatifs. L'une des limites de cette croissance exponentielle est donc finalement un problème de dissipation de chaleur : plusieurs dizaines de watts doivent être dissipés sur une petite

surface. Cette limite technologique est appelée le *power wall*.

Aujourd'hui, les progrès en performance et efficacité énergétique des processeurs doivent donc provenir d'autres axes d'amélioration. C'est par exemple le début de l'ère du multi-cœur : pour réaliser plus de calculs par seconde, on réplique les cœurs de processeur identiques sur un même circuit électronique. Utiliser plusieurs cœurs permet de multiplier la puissance dissipée en conservant la même densité de puissance. Cette solution améliore les performances mais n'améliore pas l'efficacité énergétique. De plus, utiliser efficacement une telle architecture *parallèle* n'est pas forcément facile. En effet, l'accélération d'une tâche n'est pas proportionnelle au nombre de cœurs : elle est limitée par la partie *séquentielle* de la tâche (loi d'Amdahl). Par exemple, pour une tâche dont 95 % du temps de calcul peut être parallélisé, l'accélération maximale est asymptotiquement seulement d'un facteur 20 pour un très grand nombre de processeurs.

Une autre approche est la *spécialisation* : il s'agit de concevoir une plateforme spécialisée dans la réalisation d'une tâche (ou d'un type de tâches) plutôt que d'utiliser un processeur généraliste. Les processeurs graphiques (*Graphics Processing Unit*, GPU) en sont un exemple bien connu du grand public. Ceux-ci contiennent de très nombreux processeurs (jusqu'à plusieurs milliers), mais ces processeurs exécutent les mêmes tâches sur différentes données. De plus, ceux-ci sont plus simples qu'un processeur généraliste de la même génération (ils ne contiennent par exemple pas d'unité d'exécution dans le désordre), mais contiennent des circuits permettant d'effectuer efficacement certains types de calculs (calculs graphiques 3D, d'algèbre linéaire en virgule flottante, etc.). Ce fonctionnement s'est avéré utile non seulement pour les calculs graphiques, mais aussi pour d'autres types de calculs. Les GPU, performants et relativement bon marché, ont donc évolué pour devenir plus programmables (calcul générique sur processeur graphique (*General-Purpose Computing on Graphics Processing Units*, GPGPU)). Si ce type d'architecture est moins flexible qu'un processeur généraliste conçu pour le cas moyen, il permet d'obtenir de meilleures performances pour certains types d'applications très intensives en calculs et hautement parallélisable, comme dans les domaines de l'intelligence artificielle, du calcul scientifique, du traitement d'image ou de vidéo, etc. Pour ce type d'applications, un GPU aura une meilleure efficacité énergétique qu'un processeur généraliste à niveau de performance équivalent.

Des architectures encore plus spécialisées s'éloignent du modèle de calcul du processeur qui est basé sur la notion de jeu d'instruction. En effet, si l'on connaît à l'avance les besoins d'une application (ou d'un groupe d'applications), on peut concevoir une architecture ad

hoc pour cette application. Les concepteurs ont alors une très grande liberté : l'architecture peut contenir la mémoire nécessaire pour l'application ; celle-ci peut être gérée plus intelligemment ; des interfaces spécifiques peuvent améliorer les performances des entrées/sorties ; des circuits spécialisés peuvent être utilisés pour réaliser certaines opérations très efficacement ; le niveau de parallélisme peut être finement choisi ; etc. L'architecture pourra alors avoir d'excellentes performances et efficacité énergétique, mais sera dédiée à l'application pour laquelle elle a été développée. Ces différents types de plateformes de calcul offrent un compromis entre l'efficacité et la spécialisation.

Développer des architectures dédiées nécessite des compétences avancées, demande un investissement conséquent en temps et en coût par rapport au développement d'une solution logicielle pour processeur. Une application est donc une bonne candidate au développement d'une architecture spécialisée seulement si les gains à la clé sont suffisamment intéressants. Certaines applications du domaine de l'intelligence artificielle ou du *big data* sont de bons exemples : elles traitent de grands volumes de données, nécessitent beaucoup de calculs et sont aujourd'hui très utilisées. Ces trois facteurs font que les gains en performance et en efficacité énergétique compensent l'effort de développement. Dans le domaine de l'intelligence artificielle, le monde académique et le monde industriel s'intéressent particulièrement depuis une dizaine d'années aux réseaux de neurones convolutifs (*Convolutional Neural Networks*, CNN) qui remplissent ces trois conditions. Par exemple, un seul traitement par le réseau VGG-19 peut nécessiter jusqu'à 144 millions de paramètres et 19 milliards d'opérations. De plus, les CNN permettent un haut niveau de parallélisme et supportent des calculs avec des précisions réduites (permettant réduire l'utilisation de la mémoire, des ressources de calcul et l'énergie nécessaire) voire d'opérateurs spécialisés. Tous ces éléments font que les CNN peuvent être très performants sur des architectures spécialisées et celles-ci permettent d'obtenir une très bonne efficacité énergétique par rapport à une implémentation sur processeur généraliste ou même sur GPU.

Les circuits numériques électroniques utilisés dans ces différentes architectures fonctionnent aujourd'hui presque tous selon une logique synchrone : les calculs sont cadencés à une fréquence donnée. À partir d'une certaine fréquence, les transistors ne sont plus assez rapides : les signaux n'ont plus le temps d'être propagés avant la fin de la période d'horloge. Le calcul impliqué devient alors faux, provoquant ce qu'on appelle une *faute temporelle*. La fréquence maximale à laquelle un circuit électronique peut fonctionner dépend de nombreuses contraintes (le schéma du circuit, les caractéristiques électroniques,

l'environnement comme la température, etc.) qui peuvent être variables (variabilité de fabrication, vieillissement, évolution de l'environnement, etc.). Les outils de conception calculent une fréquence pour laquelle le fonctionnement normal du circuit est garanti en considérant le cas le plus défavorable pour chaque contrainte et en appliquant une marge d'erreur. En pratique, le circuit a donc de grandes chances de fonctionner correctement à une fréquence supérieure. Cette possibilité permet d'améliorer le niveau de performance et d'efficacité énergétique en utilisant des techniques dites de *spéculation temporelle*, aussi connues sous le terme d'*overclocking*. Celles-ci consistent à augmenter la fréquence de fonctionnement du circuit au-delà de sa valeur maximale spécifiée par les outils de conception et à *spéculer* sur le fonctionnement correct du circuit.

L'augmentation de la fréquence permet d'accroître la performance : le nombre de calculs par unité de temps est proportionnel à la fréquence d'horloge. Elle améliore aussi l'efficacité énergétique, c'est-à-dire la quantité d'énergie nécessaire par calcul. En effet, la puissance instantanée augmente de manière affine avec la fréquence mais les calculs sont exécutés plus rapidement : l'énergie nécessaire pour réaliser un calcul donné est au final plus faible.

À cause du risque d'erreur temporelle, il est impératif de combiner la spéculation temporelle avec un moyen de détecter les erreurs afin de pouvoir garantir le bon fonctionnement du circuit. Tous les mécanismes ont un coût en énergie et en ressources, il est donc nécessaire de rendre ces coûts les plus faibles possibles afin de ne pas annihiler les gains dus à la spéculation temporelle.

Il existe des mécanismes de détection d'erreur génériques, comme la duplication des calculs, mais ceux-ci ont un coût rédhibitoire et sont incompatibles avec l'objectif d'augmentation des performances et de l'efficacité énergétique. Les travaux existants portant sur la spéculation temporelle se basent donc sur des mécanismes de détection d'erreur au niveau circuit (Ernst et al. 2003 ; J. Nunez-Yanez 2013 ; J. M. Levine et al. 2012). Ces mécanismes se basent sur un double échantillonnage du signal : les deux échantillons permettent de déduire qu'une faute temporelle a eu lieu. Pour certains types de calculs, dont les CNN, d'autres méthodes plus spécifiques peuvent être utilisées comme la redondance basée sur l'arithmétique modulaire. Cependant, toutes ces approches limitent leur couverture d'erreur pour limiter leur surcoût et nécessitent une grande précision sur l'analyse du comportement temporel du circuit afin d'implanter correctement les circuits spécifiques.

Dans ce travail, nous nous focalisons sur une autre approche : la détection d'erreur

au niveau algorithmique (*Algorithm Based Fault Tolerance*, ABFT). L'ABFT est une méthode permettant de détecter les erreurs en examinant une propriété de l'algorithme implémenté par le circuit, plutôt qu'en examinant le comportement du circuit lui-même. Elle ne dépend donc pas des spécificités du circuit, mais uniquement de l'algorithme protégé. Cette approche est robuste, a une très bonne couverture d'erreur et utilise très peu de ressources. En revanche, une solution est spécifique à un algorithme et est difficile à généraliser.

Contributions

Les travaux de cette thèse portent sur l'utilisation de la spéculation temporelle pour améliorer les performances et l'efficacité énergétique d'architectures dédiées ou accélérateurs matériels. Nous étudions l'utilisation de la spéculation temporelle combinée à un mécanisme de tolérance aux fautes permettant de détecter les erreurs temporelles. Nous proposons une nouvelle méthode de tolérance aux fautes au niveau algorithmique pour l'algorithme de convolution utilisé dans les réseaux de neurones convolutifs. Nous montrons comment l'*overclocking* et ce nouveau mécanisme de détection d'erreur permettent d'augmenter le débit et l'efficacité énergétique d'accélérateurs de convolution, quels gains nous obtenons pour quel coût et quelles sont les possibilités et limites de cette approche.

Plan du document

Ce document est organisé en quatre chapitres.

Dans le premier chapitre, nous présentons le contexte dans lequel ce travail s'inscrit. Nous présentons et comparons différentes plateformes actuelles sous l'angle du compromis entre performance, efficacité énergétique et puissance.

Dans le deuxième chapitre, nous présentons la spéculation temporelle dans le contexte du fonctionnement des circuits électroniques numériques, particulièrement pour les circuits de logique programmable. Nous situons ensuite les erreurs temporelles dues à la spéculation temporelle dans le contexte du domaine de la tolérance aux fautes. Nous présentons enfin plusieurs techniques existantes permettant de détecter de telles erreurs.

Dans le troisième chapitre, nous exposons le fonctionnement des CNN et les techniques utilisées pour concevoir des architectures dédiées pour CNN. Nous présentons les travaux existants concernant la spéculation temporelle et les CNN, ainsi que ceux concernant la

détection d'erreur pour CNN.

Dans le quatrième chapitre, nous présentons un nouveau mécanisme de tolérance aux fautes au niveau algorithmique pour les convolutions des CNN permettant d'utiliser la spéculation temporelle pour cet algorithme afin d'augmenter ses performances et l'efficacité énergétique. Nous présentons ensuite comme utiliser ce mécanisme dans une architecture dédiée accélérant le calcul de convolution. Nous évaluons ensuite les bénéfices et les limites de notre approche.

Chapitre 1

Contexte

Les plateformes de calcul modernes, comme les processeurs, sont construites à partir de transistors, un composant électronique élémentaire. Aujourd'hui, les calculs réalisés par ces transistors sont organisés selon une logique synchrone, agencée par cycles. À chaque unité de temps (un cycle), les transistors utilisent de l'énergie pour réaliser un « travail ». L'enchaînement de ces calculs permet de réaliser des tâches plus complexes. Les *performances* d'une machine, le nombre de calcul par unité de temps, dépend de la durée de ce cycle, de la complexité des calculs réalisés à chaque cycle et de la faculté de combiner ces calculs élémentaires pour réaliser des tâches plus complexes (compilation, ordonnancement, etc.). Les performances d'une machine sont étroitement liées à la puissance nécessaire à son fonctionnement. Son *efficacité énergétique* est la quantité d'énergie utilisée pour réaliser un calcul.

Dans ce chapitre, nous présentons et comparons différentes plateformes de calcul actuelles sous l'angle du compromis entre performances, efficacité énergétique et puissance.

1.1 Compromis performance/puissance

En électronique numérique, les transistors sont utilisés en « tout ou rien » : un transistor peut avoir deux états stables distincts (représentant une information binaire) ou être en train de commuter, c'est-à-dire de changer d'état. Quelle que soit leur technologie, les transistors ont besoin de puissance électrique pour fonctionner, même lorsqu'ils ne changent pas d'état (la puissance statique) et de puissance supplémentaire pour commuter (la puissance dynamique). Ainsi, plus les transistors commutent, plus ils nécessitent de puissance dynamique.

Le temps nécessaire pour qu'un transistor commute varie selon plusieurs paramètres, dont sa tension d'alimentation. Augmenter la tension d'alimentation des transistors leur permet d'être plus rapides, donc d'augmenter la fréquence du circuit, ce qui revient à accroître le nombre de cycles par seconde. Il y a cependant une limite : notamment, cette

puissance doit être dissipée par le boitier du circuit sous forme de chaleur. Augmenter la tension d'alimentation et la fréquence accroît les performances aux prix de puissances statique et dynamique plus élevées.

Via leur fonctionnement, les technologies de transistors offrent donc un compromis entre les performances de calculs et la puissance électrique nécessaire, quel que soit le type de plateforme. Dans la suite de cette section, nous présentons plus en détails la relation entre les performances d'un transistor et la puissance nécessaire à son fonctionnement. Nous présentons ensuite un cas d'usage de ce compromis : la gestion dynamique de la fréquence de la tension.

1.1.1 Puissance et performance d'un transistor

La puissance dissipée par un circuit CMOS peut être modélisée comme suit (Chandrakasan, Sheng et Brodersen 1992). Elle est la somme de sa puissance statique P_{stat} et de sa puissance dynamique P_{dyna}. La puissance statique est le produit du courant de fuite des transistors du circuit I_{fuite} par leur tension d'alimentation V_{DD}. Elle ne varie donc pas selon l'activité du circuit. En revanche, la puissance dynamique est liée à la commutation des transistors et est proportionnelle au carré de la tension d'alimentation, à la fréquence de commutation des transistors (qui est la fréquence de fonctionnement f multipliée par le facteur d'activité α) et la capacité de charge C.

$$P_{stat} = V_{DD} I_{fuite} \tag{1.1}$$

$$P_{dyna} = \alpha C f V_{DD}^2 \tag{1.2}$$

Les paramètres C et I_{fuite} sont liés au circuit (technologie de transistors utilisée, nombre de transistors, chemin critique, etc.) et ne peuvent pas être changés. Le paramètre α correspond à la probabilité qu'un transistor commute : il dépend donc directement de l'activité du circuit. Par conséquent, seules la tension d'alimentation et la fréquence peuvent être changées pour modifier la puissance dissipée par le circuit [1].

Augmenter la tension d'alimentation du circuit rend la commutation des transistors plus rapide et donc réduit donc le temps de propagation des signaux. Le circuit peut alors être utilisé à une fréquence plus élevée. Plus précisément, la tension d'alimentation et la

1. Il est toutefois possible de désactiver une partie du circuit pour limiter sa dissipation de puissance (*dark silicon*), ce qui peut être modélisé comme une diminution du paramètre α.

fréquence maximale sont liées par la formule :

$$f_{max} = \frac{1}{KC_{out}} \frac{(V_{DD} - V_{th})^2}{V_{DD}} \tag{1.3}$$

où K, C_{out} sont des paramètres dépendant du circuit et V_{th} est un paramètre des transistors (la tension de seuil).

Pour une technologie et un circuit donnés, on peut donc déterminer des couples (f_{max}, V_{DD}) pour lesquels la performance du circuit est maximisée pour une puissance donnée. Diminuer la tension et la fréquence réduit la puissance dissipée par le circuit, mais les performances seront moindres.

1.1.2 Gestion dynamique de la fréquence et de la tension

Une même technologie de transistors peut donc être utilisée à différents niveaux de puissance et de performance. Ces niveaux sont choisis selon l'objectif à atteindre et différentes contraintes. Pour un circuit destiné au calcul intensif, on cherche à maximiser les performances du circuit tout en respectant une puissance maximale et des contraintes thermiques (la puissance devant être dissipée sous forme de chaleur). Pour un circuit embarqué, il est préférable de minimiser l'énergie utilisée sous contrainte de performance minimale.

Par exemple, la figure 1.1 montre les différents points de fonctionnement, les couples (f_{max}, V_{DD}), qu'un processeur ARM Cortex-A9 peut utiliser, ainsi que la relation entre tension, fréquence et puissance. Le point le plus haut nécessite environ 16 fois plus de puissance que le point le plus bas, pour une fréquence 2,1 fois plus élevée. Cela permet au concepteur d'un système utilisant ce processeur de choisir le point de fonctionnement correspondant aux contraintes du système.

Dans certains cas, lorsque les contraintes varient au cours du temps, il peut être intéressant de pouvoir sélectionner dynamiquement le niveau de performance et de puissance : c'est la gestion dynamique de la fréquence et de la tension (*Dynamic Voltage and Frequency Scaling*, DVFS). Le DVFS permet d'adapter le point de fonctionnement selon les contraintes à chaque instant : la fréquence peut être ajustée en fonction de la charge de travail et la tension réduite pour limiter la puissance nécessaire. Ce mécanisme est notamment utilisé dans les processeurs grand public, car la quantité de travail dépend de l'utilisation et varie, le système peut être alimenté par batterie ou non, etc.

Cependant, la tension et la fréquence ne peuvent pas être accrues indéfiniment. La

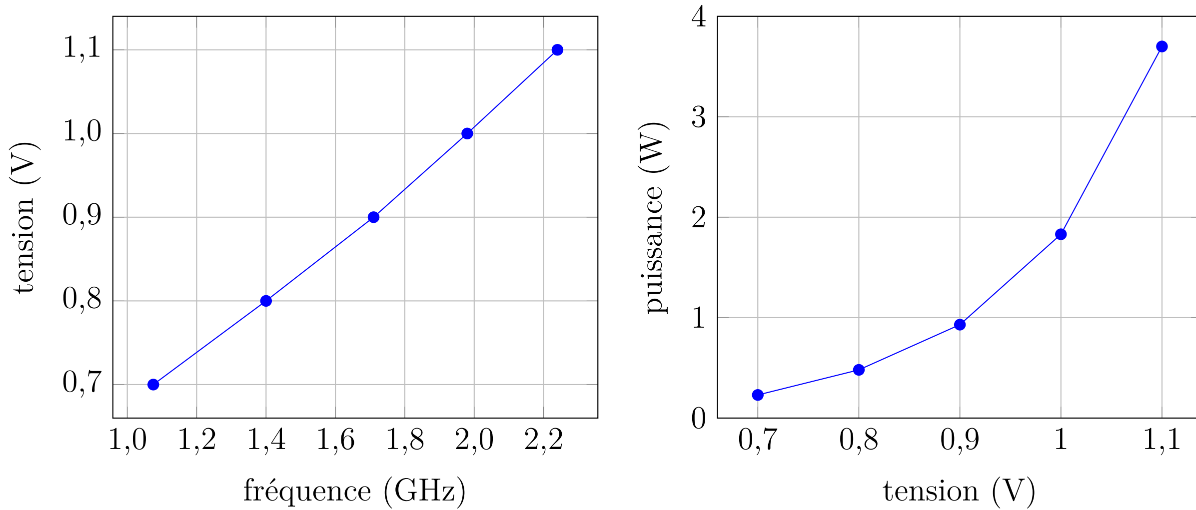


FIGURE 1.1 – Points de fonctionnement en tension et fréquence pour un processeur ARM Cortex-A9 (CESANA, FLATRESSE et CAUCHY 2012)

tension et la fréquence ne sont donc qu'un aspect du compromis entre performance et puissance.

1.2 Impact de la microarchitecture des processeurs sur les performances et l'énergie

L'amélioration de la technologie des transistors permettait de rendre les circuits électroniques plus performants et plus efficaces en énergie. À chaque nouvelle génération de transistor, leur miniaturisation permettait d'en implanter de plus en plus sur une même surface de silicium. Notamment, le nombre de transistors dans un processeur doublait à peu près tous les deux ans (loi de Moore). La tension d'alimentation baissait et la fréquence maximale des circuits augmentait proportionnellement à la réduction de la taille, mais la densité de puissance restait constante (*Dennard scaling*). Les nouveaux processeurs pouvaient donc réaliser des calculs plus rapidement (performance) en consommant moins d'énergie par calcul (efficacité énergétique).

En parallèle à l'amélioration de la technologie des transistors, les concepteurs de processeurs et autres circuits cherchent des solutions *architecturales* pour améliorer les performances et l'efficacité énergétique. Ces améliorations s'appliquent à l'organisation des

données et des calculs. Elles sont devenues d'autant plus importantes depuis la fin des lois empiriques de Moore et de Dennard dans les années 2000 parce que celles-ci ne sont plus vérifiées.

Cette section présente certaines de ces améliorations clés : le *pipeline*, l'ordonnancement dynamique, l'exécution spéculative, les jeux d'instructions vectoriels et la hiérarchie mémoire.

1.2.1 Parallélisme d'instruction

Augmenter le nombre d'instructions exécutées par unité de temps permet d'améliorer les performances d'un processeur. Augmenter la fréquence augmente le nombre de cycles par unité de temps, mais il y a une limite à cette approche. Pour améliorer davantage les performances, il faut donc augmenter le nombre d'instructions exécutées par cycle (*Instructions per cycle*, IPC) en exploitant le parallélisme au niveau instruction (*Instruction-Level Parallelism*, ILP).

L'exécution d'une instruction requiert plusieurs étapes comme la récupération de l'instruction, son décodage, l'accès à la mémoire, l'exécution arithmétique ou l'écriture du résultat dans les registres. Ces étapes n'utilisent pas forcément les mêmes structures du processeur. Cela signifie que lorsqu'une étape est en cours, les structures du processeur pour les autres étapes sont inutilisées. En divisant le circuit d'exécution d'une instruction en plusieurs étages indépendants et en *recouvrant* l'exécution de plusieurs instructions successives, plusieurs étages peuvent être utilisés *parallèlement*. Chaque instruction est alors étalée sur plusieurs cycles d'horloge. L'IPC augmente parce que plus d'instructions sont exécutées chaque seconde. C'est une microarchitecture en *pipeline*. La figure 1.2 illustre l'exécution parallèle d'une suite d'instructions.

Avec un *pipeline*, la fréquence d'horloge doit être suffisamment basse pour que l'étage le plus lent puisse être exécuté. Il est donc opportun d'*équilibrer* les étages du *pipeline* afin d'obtenir la fréquence la plus élevée possible et que chaque étage utilise efficacement ses ressources. Tous les processeurs modernes, hormis quelques exceptions parmi les plus simples, utilisent un *pipeline*. Par exemple, les processeurs Intel grand public récents utilisent un *pipeline* à 14 étages.

Il est aussi possible d'utiliser plusieurs unités d'exécution (en *pipeline* ou non) parallèles et de dispatcher le flux d'instructions du programme sur ces unités d'exécution : c'est une microarchitecture superscalaire.

Cependant, une microarchitecture en *pipeline* ou superscalaire nécessite que le pro-

instruction \ cycle	0	1	2	3	4	5	6	7	8	9	10	11
0	IF	ID	EX	MEM	WB							
1		IF	ID	EX	MEM	WB						
2			IF	ID	EX	MEM	WB					
3				IF	ID	EX	MEM	WB				
4					IF	ID	EX	MEM	WB			
5						IF	ID	EX	MEM	WB		
6							IF	ID	EX	MEM	WB	
7								IF	ID	EX	MEM	WB

FIGURE 1.2 – Trace d'exécution de 7 instructions en 11 cycles d'horloge pour un processeur RISC. Au cycle 6, l'instruction 6 commence (à l'étage IF) tandis que, parallèlement, l'instruction 2 finit (à l'étage WB). Chaque instruction nécessite 5 cycles et à chaque cycle 5 instructions sont en cours d'exécution. Sans *pipeline*, l'instruction 1 devrait être exécutée après la fin de l'exécution de l'instruction 2.

cesseur vérifie que l'exécution superposée des instructions ne viole pas la sémantique des instructions (aléa ou *hazard*). En effet, l'architecture du jeu d'instructions (*Instruction Set Architecture*, ISA) spécifie la sémantique des instructions telles qu'exécutées dans un modèle séquentiel et la microarchitecture doit s'y conformer. Ainsi, si une instruction nécessite une donnée calculée par une instruction précédente proche, il faut arrêter le *pipeline*[2] (*stall*), c'est-à-dire arrêter l'exécution des instructions ultérieures, jusqu'à ce que la donnée soit disponible (*data hazard*). De manière similaire, si une instruction est un saut conditionnel, le processeur ne connaît pas l'instruction suivante immédiatement (selon si le saut est pris ou non) et doit donc arrêter le *pipeline* (*control hazard*). Enfin, si une instruction provoque un défaut de cache, la pénalité due au *stall* s'ensuivant peut-être de plusieurs centaines de cycles. Ces aléas réduisent donc l'IPC effectif.

1.2.2 Ordonnancement dynamique

Un moyen de limiter l'impact des aléas sur l'IPC est de réorganiser les instructions afin d'éviter les *stalls* : c'est l'ordonnancement dynamique. Le processeur n'exécute alors plus les instructions dans l'ordre spécifié par le programme mais dans le désordre (*Out-of-order execution*, OOO). Pour cela, il gère une fenêtre d'instructions dans laquelle une instruction peut être exécutée dès que ses opérandes sont prêts. Par exemple, en cas de défaut de cache, des instructions ne dépendant pas de cet accès mémoire pourront être exécutées pendant le temps d'attente. Cependant, le processeur doit toujours s'assurer de

2. Certains *data hazards* peuvent cependant être résolus avec des « raccourcis » dans le pipeline fournissant un résultat aux instructions suivantes le plus tôt possible : le *forwarding*.

respecter la sémantique du programme. Les instructions arithmétiques peuvent donc être réorganisées en respectant les diverses dépendances, mais pas les instructions d'accès à la mémoire et les branchements.

L'ordonnancement dynamique permet d'éviter les *stalls* mais a un coût important en ressources et en énergie à cause de sa complexité.

1.2.3 Exécution spéculative

Bien que l'exécution OOO permette d'augmenter l'IPC, celui-ci est limité par les branchements et les instructions d'accès à la mémoire. Les premières limitent l'IPC parce que le processeur ne peut réordonner uniquement les instructions qu'il est certain d'exécuter. Les instructions rentrent dans la fenêtre d'exécution seulement lorsque les branchements sont résolus. Les secondes parce que exécuter les accès à la mémoire dans l'ordre permet de s'assurer que la sémantique du programme (basée sur l'exécution dans l'ordre) est respectée.

L'exécution spéculative permet de lever ces limitations : le principe est d'autoriser l'exécution d'instructions qui *peuvent* être nécessaires et d'exécuter les instructions avec accès mémoire dans le désordre.

Pour les branchements, une branche est considérée comme prise et la fenêtre d'instructions est remplie avec les instructions de cette branche, permettant leur exécution dans le désordre. Le processeur enregistre les instructions terminées dans une mémoire, le *Re-Order Buffer* (ROB). Finalement, le processeur sait si la branche prise était la bonne lors de la résolution du branchement. Dans ce cas, il valide les instructions correspondantes dans le ROB, dans l'ordre original du programme. Dans le cas contraire, il les annule, ainsi que la fermeture transitive des instructions ayant utilisé des résultats d'instructions invalides. Un mécanisme de prédiction de branchement permet de choisir la branche considérée comme prise selon diverses heuristiques afin d'augmenter le niveau l'IPC effectif.

Les opérations de lecture et d'écriture en mémoire des instructions sont gérées via une mémoire, la *Load-Store Queue* (LSQ), permettant de déterminer si des dépendances mémoires ont été violées à cause de l'exécution dans le désordre. Cette mémoire enregistre les adresses des données accédées ainsi que leur valeur (dans le cas l'écriture) lors de l'exécution spéculative des instructions d'écriture (*store*) et de lecture (*load*). Les lectures récupèrent les données depuis la LSQ si une écriture correspondante a eu lieu et que cette écriture précède la lecture dans l'ordre original du programme, sinon depuis la mémoire.

Les écritures sont valides, sauf si une lecture correspondante suivant l'écriture dans l'ordre original a déjà eu lieu. Dans ce cas, l'instruction de la lecture ainsi que la fermeture transitive des instructions ayant utilisé cette valeur sont annulées grâce au ROB et réinjectées dans la fenêtre d'exécution afin de les exécuter à nouveau ultérieurement. Les écritures spéculatives sont reportées en mémoire uniquement lorsque l'instruction correspondante est validée, permettant leur exécution séquentielle conformément à la sémantique du programme. Lors de la validation d'une instruction, les informations associées dans la LSQ sont retirées.

Tout comme l'ordonnancement dynamique, l'exécution spéculative augmente l'IPC mais a un coût important en mémoire, en logique et en énergie. Michaud et al. ont observé empiriquement que le nombre d'instructions exécutables parallèlement est proportionnel à la racine carrée de la taille de la fenêtre (MICHAUD, SEZNEC et JOURDAN 2001). Augmenter l'IPC exploitable via l'exécution spéculative est donc de plus en plus coûteux, plus que linéairement, pour le niveau d'IPC recherché.

1.2.4 Jeu d'instructions vectoriel

Plutôt que d'exploiter le parallélisme intrinsèque à un programme en l'analysant dynamiquement, il est possible d'exposer celui-ci explicitement dans les instructions. Dans ce cas, c'est le compilateur et non le processeur, qui extrait le parallélisme du programme par des analyses statiques. Il faut alors que le jeu d'instructions du processeur soit conçu pour exprimer ce parallélisme.

Certains jeux d'instructions (comme les familles x86 et ARM) ont été étendus par des extensions vectorielles. Ces extensions (MMX, SSE, AVX, 3DNow!, NEON, etc.) ajoutent des instructions permettant de traiter des ensembles (vecteurs) de données, plutôt qu'une donnée unique. Par exemple, les extensions AVX-512 peuvent traiter des vecteurs de 512 bits, soit par exemple 16 nombres à virgule flottante simple précision, 64 entiers 8 bits, etc. Les calculs d'une instruction peuvent alors être traités efficacement par la microarchitecture du processeur et ordonnancés (statiquement) sur les ALU du processeur. Elle permet aussi d'améliorer la récupération des données en mémoire en requérant parfois que les données vectorisées soient contigües et alignées et en augmentant la densité des données « utiles » dans le programme (moins d'instructions sont utilisées pour autant d'adresses de données). La figure 1.3 donne un exemple de programme assembleur mettant en œuvre une telle extension pour un type de calcul intensément utilisé pour certains algorithmes d'intelligence artificielle. Ce fonctionnement est de type instruction unique,

```
mac:
    xorl        %r8d, %r8d
.loop:
    movzbl      (%rdi,%r8), %eax
    imulb       (%rsi,%r8)
    movswl      %ax, %ecx
    movzbl      1(%rdi,%r8), %eax
    imulb       1(%rsi,%r8)
    cwtl
    addl        %eax, %ecx
    movzbl      2(%rdi,%r8), %eax
    imulb       2(%rsi,%r8)
    addl        (%rdx,%r8), %ecx
    cwtl
    addl        %eax, %ecx
    movzbl      3(%rdi,%r8), %eax
    imulb       3(%rsi,%r8)
    cwtl
    addl        %eax, %ecx
    movl        %ecx, (%rdx,%r8)
    addq        $4, %r8
    cmpq        $64, %r8
    jne         .loop
    ret
```

```
mac:
    vpdpbusd    (%rdi), %zmm1, %zmm0
    vmovdqa64   %zmm0, (%rdi)
    ret
```

```
void mac(
  int8_t *l,
  int8_t *r,
  int32_t *res
) {
  for(int i = 0; i < 16; i++) {
    int16_t s0 = l[4*i+0] * r[4*i+0];
    int16_t s1 = l[4*i+1] * r[4*i+1];
    int16_t s2 = l[4*i+2] * r[4*i+2];
    int16_t s3 = l[4*i+3] * r[4*i+3];
    res[i] += s0 + s1 + s2 + s3;
  }
}
```

FIGURE 1.3 – Programmes assembleur x86-64 et C équivalents pour une fonction `mac` effectuant la multiplication exacte de quatre paires d'entiers 8 bits et leur accumulation dans un entier 32 bits, répété 16 fois. Le programme de gauche est sans extension vectorielle et celui de droite avec l'extension AVX-512 *Vector Neural Network Instructions* (VNNI). Le programme est plus court (une seule instruction de calcul), sans boucle, les accès mémoires sont réguliers et les différents calculs peuvent être efficacement ordonnancés.

données multiples (*Single Instruction Multiple Data*, SIMD) selon la taxonomie de Flynn. L'avantage de cette approche est de permettre de meilleures performances si l'extension est utilisée tout en gardant la compatibilité descendante. En revanche, il ne permet que d'exploiter le parallélisme de données.

1.2.5 Hiérarchie mémoire

Comme que les processeurs peuvent exécuter plus d'instructions par seconde, l'accès à la mémoire devient plus critique pour les performances. En effet, un processeur, ou n'importe quelle plateforme de calcul, a besoin d'accéder aux données à traiter (et éventuellement au programme à exécuter), stockées dans une mémoire. Or, les performances des mémoires s'améliorent plus lentement que celles des processeurs : l'écart se creuse d'année en année. Pour limiter l'impact de la mémoire sur les performances des processeurs, les solutions doivent donc être architecturales (WULF et MCKEE 1995).

La mémoire doit optimiser plusieurs métriques indépendantes :

— la capacité : la quantité d'information stockée ;
— la latence : le temps d'attente pour accéder à une information ;
— le débit : la quantité d'information transférée par unité de temps ;
— la puissance électrique nécessaire ;
— le coût.

Aucune technologie de mémoire n'est optimale sur tous ces points. Au contraire, elles offrent différents compromis. Une manière d'obtenir une mémoire avec de bonnes performances sur toutes les métriques est d'utiliser une *hiérarchie mémoire*.

Plutôt que d'implémenter une unique mémoire avec une seule technologie, une hiérarchie mémoire utilise plusieurs mémoires de différentes tailles tirant profit des avantages des différentes technologies. Ces mémoires sont organisées en niveaux hiérarchiques. Chaque niveau a une plus grande capacité, une plus grande latence, un plus faible débit, une plus faible puissance et un plus faible coût (par octet) que le niveau précédent. Si une donnée requise est absente d'un niveau (*miss*), elle est (récursivement) rapatriée depuis le niveau supérieur. Si la donnée est présente (*hit*), il n'y a pas besoin d'utiliser le niveau supérieur. Les données modifiées ou créées sont aussi reportées au niveau supérieur lorsque c'est nécessaire. Chaque niveau contient donc un sous-ensemble, potentiellement mis à jour, des données du niveau supérieur. Le temps d'accès à une donnée dépend donc directement du niveau le plus bas dans lequel la donnée se trouve.

La plupart des programmes ou applications n'accèdent pas aux données uniformément mais tendent à réutiliser les données et instructions utilisées précédemment (localité temporelle) et les données et instructions adjacentes (localité spatiale) (Hennessy et Patterson 2011). Ces propriétés sont exploitées pour augmenter la probabilité de trouver une donnée dans un niveau faible de la hiérarchie lorsque le processeur en a besoin. Les données sont copiées dans un niveau inférieur par *lignes*, en copiant donc une certaine quantité de données adjacentes à la donnée requise qui peuvent potentiellement être utiles. Certains processeurs implémentent aussi un *prefetcher* pour copier les données avant que le processeur ne les requiert en spéculant sur le fait qu'elles seront utilisées (par exemple, en copiant la ligne suivant la ligne requise). Les données et instructions utilisées ont donc plus de chances de se trouver dans les niveaux inférieurs : malgré quelques temps d'accès longs, le temps d'accès moyen peut être beaucoup plus faible que les temps d'accès des niveaux supérieurs. Grâce à la localité, la hiérarchie mémoire permet donc d'obtenir *virtuellement* une mémoire ayant une faible latence et un haut débit grâce aux premiers

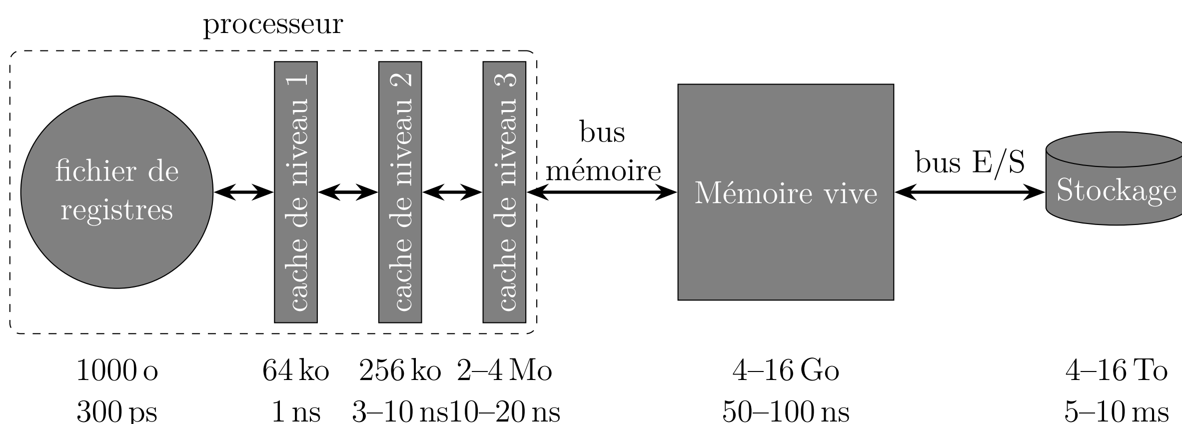


FIGURE 1.4 – Hiérarchie mémoire : capacité et latence d'accès (HENNESSY et PATTERSON 2011)

niveaux et une grande capacité grâce aux derniers niveaux pour un coût et une puissance limités.

Par exemple, dans un processeur moderne, la hiérarchie mémoire couvre typiquement le fichier de registres [3] (temps d'accès inférieur à un cycle d'horloge), deux ou trois niveaux de caches et la mémoire vive extérieure au processeur, comme illustré par la figure 1.4. On peut aussi considérer le stockage de masse, non volatile, comme faisant partie de la hiérarchie mémoire avec une taille pouvant aller jusqu'à plusieurs téraoctets et des temps d'accès de 50 µs pour les mémoires flash à 10 ms pour les disques durs. La plage de temps d'accès recouvre donc huit ordres de grandeur et celle des capacités dix ordres de grandeur.

La hiérarchie mémoire a un impact important sur les performances des processeurs, mais aussi sur leur puissance électrique et leur efficacité énergétique. Par exemple, Dally et al. montrent que 70 % de l'énergie d'un processeur embarqué est utilisée pour gérer les données et instructions, alors que seule 6 % est utilisée pour réaliser les calculs (voir le détail figure 1.5) (DALLY et al. 2008). Au total, l'énergie requise pour exécuter une instruction est 15 à 50 fois l'énergie utilisée pour réaliser les calculs.

1.2.6 Limites

Bien que ces améliorations microarchitecturales augmentent les performances des processeurs, elles ont un coût élevé en ressources et en puissance. Par exemple, un cache

3. Contrairement aux caches dont le contenu est géré automatiquement à partir des données requises, les registres sont gérés explicitement par le programme.

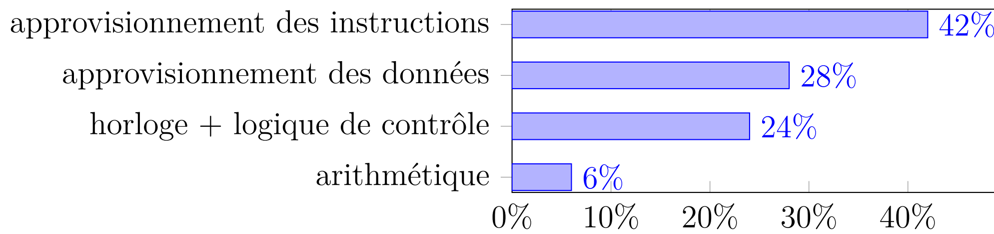


FIGURE 1.5 – Répartition de la consommation d'énergie pour un processeur embarqué. 70 % de l'énergie est utilisée pour gérer les données et instructions contre seulement 6 % pour réaliser les calculs dont uniquement 59 % sont « utiles » pour l'application (DALLY et al. 2008).

ayant plus de capacité permet en général de meilleures performances pour un coût énergétique élevé, les caches représentant une part importante de la puissance nécessaire aux processeurs. De la même manière, un *prefetcher* permet dans certains cas d'accélérer un programme, mais les données copiées inutilement peuvent diminuer l'efficacité énergétique (HENNESSY et PATTERSON 2011). Le même raisonnement s'applique à l'exécution spéculative : puisque certaines instructions sont invalidées, l'énergie ayant servi à leur exécution inutile est gaspillée et l'efficacité énergétique diminuée. L'exécution dans le désordre, une microarchitecture superscalaire ou dans une moindre mesure une architecture en *pipeline* nécessitent d'analyser les dépendances entre les instructions à la volée et donc de la logique et de l'énergie. Le choix d'utiliser ou non ces améliorations dépend du compromis entre les performances et le coût matériel et énergétique voulu pour le processeur lors de sa conception.

Les choix de conception de processeurs, comme la taille des caches, leur nombre, leur stratégie de remplacement, le nombre de registres, leur taille, la taille de la fenêtre de spéculation, etc. sont en général basés sur le *cas moyen*. En effet, les processeurs sont des machines génériques et on ne sait généralement pas quel type de programme ils devront exécuter, ou bien les programmes exécutés seront variables. Ces choix peuvent donc être pertinents pour certaines applications mais au contraire inefficaces voire contre-productifs pour d'autres.

Les compilateurs peuvent prendre en compte certains détails microarchitecturaux pour optimiser les performances de programmes compilés pour un processeur donné. Par exemple, si un compilateur connaît la taille des caches, il peut essayer d'organiser les calculs pour que les données tiennent dans les caches ou pour maximiser leur réutilisation et limiter le taux de *cache miss*. Un compilateur peut aussi essayer d'exposer au mieux le

parallélisme d'instruction dans le programme pour que le mécanisme d'OOO fonctionne le mieux possible (mais il décrira le programme selon un formalisme séquentiel). De façon plus générale, un compilateur peut essayer d'éviter les sauts conditionnels, d'éloigner les instructions interdépendantes, de dérouler les boucles, etc. Ces optimisations peuvent donner de bons résultats si la compilation est ciblée sur un processeur particulier, mais pas forcément si la compilation cible un jeu d'instructions sans cibler une microarchitecture.

Comme montré précédemment, la gestion de la mémoire et des instructions représente une grosse part de la consommation d'un processeur. De plus, Dally et al. montrent que seul 59 % des calculs arithmétiques (sur l'ALU du processeur) sont « utiles », c'est-à-dire des calculs sur des données pour l'application et non des adresses, des indices de boucles, etc. Les processeurs sont des machines généralistes, capables d'exécuter n'importe quel programme, mais n'ont pas une bonne efficacité énergétique. Par exemple, l'énergie par opération d'un processeur grand public est autour de 20 nJ/op tandis qu'un ASIC peut atteindre 5 pJ/op, soit 4000 fois moins.

Dans certains cas, lorsqu'on connaît l'application ou le type d'application qu'une plateforme devra exécuter, il est donc pertinent de ne pas utiliser un processeur généraliste afin d'obtenir de meilleures performances et une meilleure efficacité énergétique. C'est notamment le cas pour les applications de calcul intensif, comme l'intelligence artificielle, le traitement d'image, etc. Dans la suite de cette section, nous présentons quelques architectures explicitement parallèles plus spécialisées.

1.3 Architectures explicitement parallèles

Pour obtenir de meilleures performances dans les processeurs basés sur un jeu d'instruction utilisant une sémantique séquentielle (comme la famille x86), le processeur doit lui-même essayer d'exploiter l'ILP en analysant le programme et en ordonnançant au mieux les instructions selon les capacités de sa microarchitecture. Comme vu précédemment, cette approche a un coût énergétique élevé.

L'exploitation du parallélisme d'un programme et l'ordonnancement des instructions peuvent être relégués à la phase de compilation (par opposition à la phase d'exécution). La plateforme est alors explicitement parallèle : son jeu d'instruction permet au compilateur d'exprimer explicitement le parallélisme d'instruction. Cette section présente quelques machines explicitement parallèles.

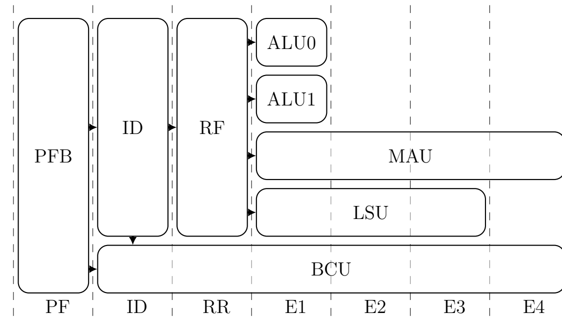


FIGURE 1.6 – Exemple de *pipeline* simplifié de VLIW à 5 voies de Kalray (B. D. de DINECHIN et al. 2013). Le processeur peut exécuter jusqu'à 5 instructions par cycle (en plus du parallélisme du *pipeline*). Les deux premières voies sont utilisées pour les calculs génériques (ALU) ; la suivante est spécialisée dans les calculs de multiplication-accumulation entiers et les calculs à virgule flottante (MAU) ; la suivante dans les accès mémoire (LSU) et la dernière dans les branchements et le contrôle (BCU). Les voies MAU et LSU supportent aussi un sous-ensemble des instructions destinées aux voies ALU. La récupération des instructions (PFB), le décodage du *bundle* d'instructions (ID) et l'accès en lecture aux registres (RF) ont chacun lieu dans le même cycle pour toutes les instructions parallèles.

1.3.1 Processeur à jeu d'instruction explicitement parallèle

Les processeurs *Very Long Instruction Word* (VLIW) utilisent un jeu d'instructions spécialisé permettant d'exposer le parallélisme des instructions : à chaque cycle, le processeur exécute un paquet (*bundle*) d'instructions. Chaque instruction est exécutée sur une unité d'exécution prédéfinie (dont certaines peuvent être spécialisées, par exemple en gérant la mémoire ou les opérations arithmétiques complexes). Le compilateur doit donc ordonnancer les instructions du programme *bundle* par *bundle* en essayant de paralléliser au mieux. La figure 1.6 montre un *pipeline* d'un VLIW.

Exposer explicitement le parallélisme des instructions permet d'obtenir une meilleure efficacité énergétique. En effet, le processeur est rendu plus simple : il n'a pas à calculer les dépendances entre instructions, à spéculer, etc. La compilation, elle, devient plus difficile mais le compilateur a un budget de temps et d'énergie pour trouver le meilleur ordonnancement possible bien supérieur au budget dans le cas de l'ordonnancement dynamique.

Cependant, en pratique, toutes les unités d'exécution ne sont pas utilisées à chaque cycle. De plus, l'ordonnancement statique ne peut pas *réagir* à des évènements tels que des *cache miss*, contrairement à l'ordonnancement dynamique. Enfin, un programme compilé pour une architecture de VLIW sera incompatible avec d'autres VLIW, même si un seul paramètre change (nombre de voies, profondeur du *pipeline*, spécialisation des voies, etc.) :

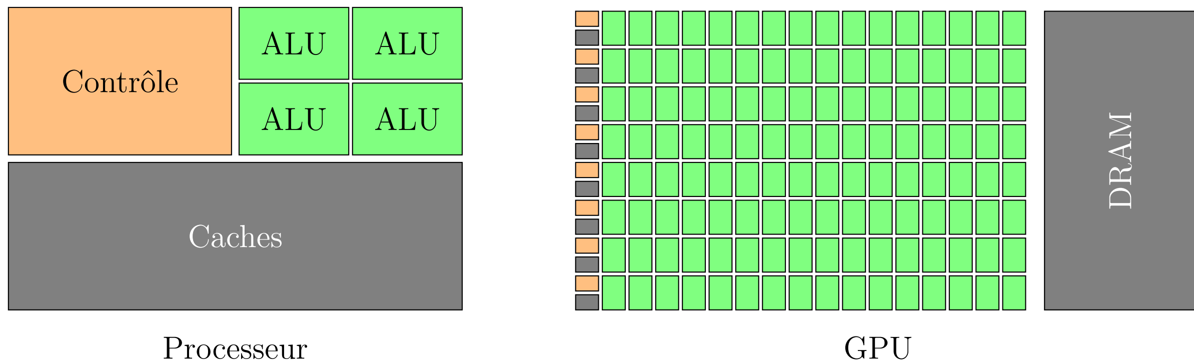


FIGURE 1.7 – Comparaison des éléments principaux d'un processeur et d'un GPU de 8 SM de 16 cœurs

il n'est pas portable. Contrairement à un processeur généraliste qui expose la même ISA quelle que soit sa microarchitecture, le programme pour VLIW devra être compilé spécifiquement pour chaque architecture. Ce type de processeur convient donc pour exécuter des noyaux d'exécution très optimisés.

1.3.2 Processeur graphique

Les processeurs graphiques (*Graphics Processing Unit*, GPU) ont une architecture hautement parallèle. Ils étaient spécialisés à l'origine dans les calculs graphiques. Ce type de calcul nécessite d'appliquer les mêmes suites de traitement à de nombreuses données (par exemple, des coordonnées 2D ou 3D (vertex), les pixels d'une image, etc.), en utilisant principalement une arithmétique à virgule flottante. Les GPU ont donc une architecture adaptée à ce type de calcul. Un GPU ne remplace pas un processeur mais est utilisé pour accélérer certains calculs.

Un GPU contient de nombreuses unités de calcul (jusqu'à plusieurs milliers) et sa propre hiérarchie mémoire (caches et mémoire vive). Ces unités d'exécution sont organisées en groupes (*Streaming Multiprocessor*[4] (SM)). Les cœurs d'un SM exécutent le même programme sur des données différentes. Le contrôle (comme le décodage des instructions) peut alors être partagé pour plusieurs cœurs, ce qui permet d'utiliser plus de ressources pour les calculs « utiles ». Par exemple, l'architecture Fermi (« NVIDIA's Next Generation CUDA Compute Architecture » 2009) contient 16 SM de 32 cœurs. La figure 1.7 illustre l'architecture simplifiée d'un GPU comparée à celle d'un processeur.

4. Les termes utilisés reprennent la terminologie CUDA.

Ce modèle d'exécution est de type instruction unique, *thread* multiples (*Single Instruction, Multiple Threads*, SIMT) (Lindholm et al. 2008). Les noyaux de calculs (*kernels*) à exécuter sur le GPU sont organisés en blocs de *threads* (fils d'exécution). Les *threads* d'un bloc exécutent les mêmes instructions sur différentes données (chaque bloc est exécuté par un SM, chaque cœur exécutant un *thread*). Les blocs peuvent être exécutés dans n'importe quel ordre, séquentiellement ou parallèlement. Leur synchronisation est cependant possible à certains points grâce à des barrières. Contrairement au modèle SIMD, les *threads* peuvent contenir des branchements et les branches prises peuvent différent entre les *threads* d'un même bloc. La gestion de la mémoire est critique pour les performances. En effet, les données doivent être transférées explicitement depuis la mémoire de l'hôte, puis l'inverse. Les copies mémoire et les calculs sont recouverts pour cacher la latence. Un GPU a donc un débit élevé, mais une latence relativement importante. Le programme à accès à une mémoire par *thread*, une mémoire partagée par bloc et une mémoire globale. Les accès mémoire des *threads* sont regroupés si possible pour minimiser le nombre de transactions. Un programme performant doit donc minimiser le nombre de transactions en organisant correctement sa mémoire et en utiliser les caches.

Si cette architecture convient bien aux calculs graphiques, les GPU ont commencé à être utilisés les années 2000 pour du calcul généraliste (calcul générique sur processeur graphique (*General-Purpose Computing on Graphics Processing Units*, GPGPU)) et sont devenus plus programmables via des interfaces de programmation comme CUDA et OpenCL. CUDA permet d'exploiter des GPU depuis un langage de programmation usuel comme C++, malgré le modèle d'exécution très différent d'un processeur.

Les GPU sont performants pour les applications de calcul intensif : rendu 3D, calcul scientifique, applications multimédias, intelligence artificielle. Ils sont pertinents pour les applications qui nécessitent un haut débit et peu de réutilisation de données. En revanche, la latence relativement importante peut être problématique pour certaines applications, comme parfois en intelligence artificielle où le résultat doit être connu le plus rapidement possible. Ils sont plus efficaces en énergie que les processeurs généralistes pour les calculs en virgule flottante avec 225 pJ contre 1700 pJ par opération en virgule flottante. Au sein d'une même génération, les GPU nécessitent généralement plus de puissance que les processeurs (par exemple 365 W pour un GPU GeForce GTX 509 contre 45 W pour un processeur Intel i7 3770T, soit 8 fois plus) mais peuvent, selon les applications, avoir une meilleure efficacité énergétique (Mittal et Vetter 2014).

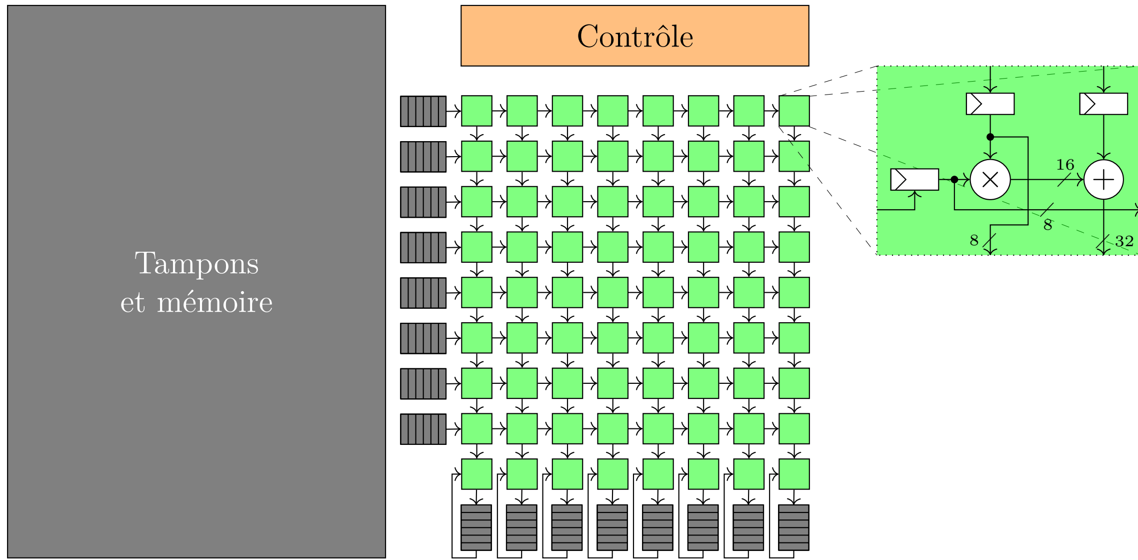


FIGURE 1.8 – Éléments principaux d'un TPU : mémoire et réseau systolique calculant la multiplication de matrice (représentée avec seulement 8×8 éléments). Les éléments principaux du *processing elements* (PE) sont montrés à droite : un multiplieur 8 bits, un additionneur 32 bits et des registres pour les opérandes (à gauche) et la somme partielle (à droite).

1.3.3 Processeur spécialisé pour l'intelligence artificielle

Si les GPU sont utilisés dans le domaine de l'intelligence artificielle, les importants besoins en performance et la spécificité de certains calculs de ce domaine ont poussé plusieurs sociétés à développer des circuits spécialisés dans ce domaine comme les *Tensor Processing Unit* (TPU) de Google. Déployée en 2015, la première version permet de réaliser une inférence de réseau de neurones profond 15 à 30 fois plus vite qu'un processeur ou GPU avec une efficacité énergétique 30 à 80 fois supérieure (JOUPPI et al. 2017). Ces niveaux de performance sont atteints grâce à la spécialisation du circuit.

Le cœur du circuit est une grille de 256×256 opérateurs de multiplication-accumulation capable de fournir un débit de 92 Top s^{-1}. Ces opérateurs travaillent sur des entiers 8 bits, une précision suffisante pour les applications cibles, car ce format permet de meilleures performances et une meilleure efficacité énergétique. En effet, la multiplication entière 8 bits nécessite 6 fois moins d'énergie et 6 fois moins de ressources matérielles que la multiplication en flottant IEEE 754 16 bits et l'addition 13 fois moins d'énergie et 38 fois moins de ressources matérielles (JOUPPI et al. 2017). La multiplication de matrice est organisée en un réseau systolique (QUINTON et ROBERT 1989) tel qu'illustré figure 1.8.

Cette organisation permet de transférer les données directement d'un opérateur à un autre, simplifie le contrôle et évite les lectures en mémoire SRAM coûteuses en énergie. Le réseau systolique représente 24 % de la surface du circuit et les différents tampons 35 %. Le contrôle ne représente que 2 % de la surface du circuit, beaucoup moins que pour un processeur ou un GPU (Jouppi et al. 2017). Le TPU contient aussi un circuit spécialisé pour d'autres calculs utilisés dans les réseaux de neurones, comme l'*activation*, la normalisation, le *pooling*, etc.

Le TPU fonctionne grâce à un processeur *hôte* qui le programme grâce à une douzaine d'instructions spécifiques. Ces instructions ont un niveau d'abstraction beaucoup plus élevé que les instructions classiques de processeur, comme *calculer une multiplication de matrice de telle taille sur les données à telles adresses*. La mémoire est gérée explicitement entre le système hôte et le TPU. Un programmeur peut cependant l'utiliser plus simplement via une interface de programmation plus abstraite comme Tensorflow (Abadi et al. 2016).

1.3.4 Discussion

Grâce au parallélisme explicite, la machine n'a plus besoin de certains mécanismes coûteux comme une unité d'exécution dans le désordre et cela permet en général d'obtenir une meilleure efficacité énergétique. Grâce à la meilleure efficacité énergétique, il devient possible d'augmenter le nombre d'unités de calculs tout en restant sous les contraintes de puissance et d'enveloppe thermique. Ainsi le TPU contient 65 536 unités de calcul spécialisées pour réaliser des multiplications de matrice ; un GPU peut contenir plusieurs milliers de cœurs et le processeur *manycore* MPPA-256 contient 256 cœurs VLIW comme celui présenté ci-dessus. Ces machines permettent donc d'obtenir de bonnes performances pour les applications hautement parallèles.

Un GPU et un VLIW restent des machines programmables, basées sur un jeu d'instruction *généralistes* (le GPU, s'il est spécialisé, contient par exemple des instructions de branchement). Au contraire, le TPU est spécialisé dans une seule tâche : réaliser les calculs nécessaires pour l'inférence d'un réseau de neurones. Il n'est pas possible d'utiliser un TPU pour d'autres tâches et son « jeu d'instruction » est très limité. C'est une architecture dédiée. Cette spécialisation permet au TPU d'être plus performant qu'un GPU et d'avoir une meilleure efficacité énergétique pour cette tâche.

Dans la section suivante, nous présentons les architectures dédiées d'une manière plus générale.

1.4 Architectures dédiées

L'objectif d'une architecture dédiée est d'améliorer les performances ou l'efficacité énergétique d'une plateforme pour une tâche spécifique. Intuitivement, une architecture dédiée utilise les mêmes éléments de base qu'une architecture généraliste (hiérarchie mémoire, bus mémoires, unités de calculs, etc.) mais une plus grosse partie de ces éléments est utilisée pour les calculs « utiles » à la tâche. Ainsi, à l'instar du TPU, les performances et l'efficacité énergétique sont meilleures que sur une machine généraliste pour la tâche choisie.

En revanche, le coût du développement d'une telle architecture jusqu'à la production de circuits électroniques (*Application-Specific Integrated Circuit* (ASIC)) est beaucoup plus élevé que le coût du développement d'une solution logicielle, ainsi que beaucoup plus long (quelques mois voire années). Il faut que les gains espérés (en performance, coût de fonctionnement, volume de vente, etc.) soit suffisamment importants pour que le développement d'une architecture dédiée soit intéressant. Les *Field-Programmable Gate Array* (FPGA), circuits reconfigurables, offrent un compromis entre le coût et le temps de développement, et la performance et l'efficacité énergétique. Ils permettent en effet d'éviter les dernières étapes du développement et la phase très coûteuses d'un ASIC. S'ils sont utilisés dans la phase de prototypage des ASIC, on les trouve aussi dans les produits finaux grâce à leur coût plus faible (par exemple en cas de faible volume) et leur aspect reconfigurable (ils permettent des corrections, des évolutions, etc.).

1.4.1 Architecture des FPGA

Les FPGA sont des composants électroniques dont le fonctionnement peut être modifié par configuration : on dit qu'ils sont *reconfigurables*, ou encore qu'il s'agit de *logique programmable*. Ils sont basés sur une grille de composants relativement simples et d'un réseau d'interconnexion. Le comportement de chaque composant, ainsi que leur interconnexion, peut être modifié par quelques bits de configurations. La reconfiguration permet ainsi d'obtenir n'importe quel circuit d'électronique numérique tant que les ressources présentes sont suffisantes. Un FPGA peut donc être vu comme un circuit prêt à l'emploi permettant d'implanter des architectures spécifiques.

Le composant principal est la *lookup Table* (LUT). Celle-ci permet le calcul de n'importe quelle fonction booléenne : les bits de configuration déterminent la fonction booléenne qui sera calculée sur les entrées de la LUT selon le principe d'une table de vérité.

Les LUT sont généralement de 4 ou 6 entrées et une sortie, nécessitant 16 ou 64 bits de configuration.

Les LUT sont regroupées dans des composants appelés *slices* (ou *logic elements*). Les *slices* contiennent plusieurs LUT, des bascules, ainsi que la logique suffisante pour réaliser efficacement des opérations communes (un registre à décalage, de la mémoire ou la retenue d'une addition) tout en restant, encore une fois, génériques.

Les FPGA modernes contiennent aussi des composants plus spécialisés implémentant des fonctions couramment utilisées. Par exemple, des multiplieurs et des additionneurs entiers câblés (*Digital Signal Processing*, DSP) ou des blocs mémoire (BRAM). Si ces composants sont toujours configurables pour être utilisés de différentes manières, leur spécialisation permet de réduire les ressources nécessaires par rapport à une implémentation avec des LUT, mais aussi d'augmenter les performances et l'efficacité énergétique de ces fonctions communes. Les fabricants de FPGA tentent de trouver un équilibre entre ces composants plus spécialisés et efficaces et les composants génériques pour convenir à tout type d'application.

L'interconnexion permet de router l'information, au niveau bit, entre les différents composants via des *switch matrix* configurables. Enfin, les FPGA contiennent des éléments gérant les entrées/sorties (IOB), des boucles à phase asservie pour générer les signaux d'horloges, un circuit permettant sa configuration, etc. La figure 1.9 illustre l'agencement des principaux éléments pour un petit FPGA.

Les circuits logiques programmables proposent un grain de configuration très fin. Ils permettent aux concepteurs de configurer les éléments logiques pour réaliser n'importe quelle fonction logique ; stocker et router les informations au niveau bit ; gérer directement les entrées/sorties ; mais aussi utiliser plusieurs horloges et configurer leur fréquence ; etc. Autrement dit, la couche d'abstraction au-dessus des composants électroniques élémentaires est assez faible. L'utilisation des FPGA n'est donc pas limitée à un usage de calculateur numérique spécialisé : ils peuvent aussi être utilisés pour prototyper ou réaliser des circuits spécifiques exploitant des caractéristiques des circuits électroniques. Ils permettent donc d'implémenter des types d'application qui ne sont pas réalisables sur d'autres architectures (si elles ne contiennent pas déjà des composants spécifiques nécessaires), comme certains générateurs de nombres aléatoires ou la détection du temps de propagation d'un signal.

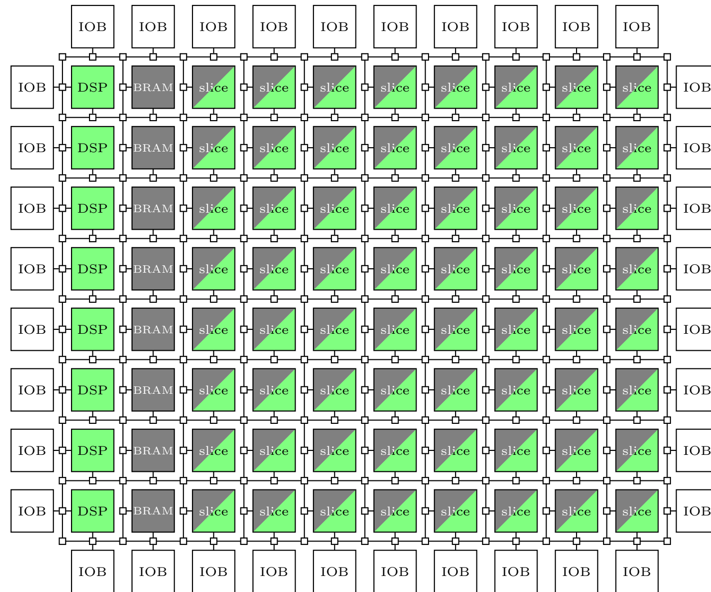


FIGURE 1.9 – Architecture schématique simplifiée d'un petit FPGA. Les différents éléments (ici DSP, BRAM, slices et autres) sont disposés en grille et interconnectés par un réseau via des *switch matrix*. Certains éléments ne sont pas représentés, comme les boucles à phase asservie, les connexions de propagation de retenue entre les *slices*, les éléments utiles à la configuration, etc.

1.4.2 Modèle de programmation

Si l'architecture matérielle d'un FPGA est régulière et relativement simple, la complexité de la logique programmable se trouve dans la description du circuit considéré et dans la génération de la configuration. Celle-ci, appelée *bitstream*, est générée par une chaîne de traitement, à l'instar de la compilation.

La description du comportement attendu du FPGA est écrite dans un langage de description de matériel (*Hardware Description Language*, HDL). Les HDL, comme le VHDL ou le Verilog, utilisent l'abstraction *Register Transfer Level* (RTL) pour décrire des architectures matérielles. Cette abstraction permet de décrire un circuit électronique synchrone en se basant sur les registres et les fonctions logiques combinatoires. En particulier, cette abstraction n'utilise pas une sémantique séquentielle comme les langages de programmations, mais décrit des *comportements*. Elle permet notamment de décrire plus facilement des comportements parallèles.

Plusieurs phases transforment la description de matériel en *bitstream*. Les outils de *synthèse logique* produisent d'abord une implémentation au niveau porte logique, la *netlist*. Cette première phase contient, entre autres, des simplifications logiques afin de réduire

le nombre de portes logiques nécessaires puis la génération du circuit à base de portes logiques. La *netlist* est ensuite projetée sur les ressources disponibles (slices, DSP, BRAM, etc.) selon les contraintes du FPGA cible (entrées/sorties, délais, etc.). Enfin, la phase de placement et routage place ces ressources sur l'architecture du FPGA cible et trouve le routage des signaux nécessaires. Cette phase doit respecter les contraintes architecturales, les contraintes de délai, les contraintes de puissance, etc. tout en essayant d'obtenir le meilleur résultat possible. Finalement, le *bitstream* peut être généré.

La description de matériel est complexe : les HDL sont « bas niveau » et offrent peu d'abstractions. Afin d'améliorer la productivité lors de la création d'un circuit, les outils de synthèse de haut niveau (*High-Level Synthesis*, HLS) permettent d'utiliser un niveau d'abstraction plus élevé. Ces outils analysent un programme écrit dans un langage « haut niveau » (comme C, C++, Matlab ou certains langages spécifiques comme SystemC) et génèrent une description de matériel correspondante en HDL. Ils transforment donc une description *algorithmique* du matériel en une description *comportementale*. La transformation se base sur une analyse sémantique, des optimisations spécifiques pour le matériel, la recherche de parallélisme, l'ordonnancement des opérations, etc.

1.4.3 Exploration de l'espace de conception

Le niveau de flexibilité dans la configuration des FPGA permet de réaliser des architectures dédiées pour les besoins d'une application et offrent de nombreux choix de conception. Par exemple, une architecture dédiée peut utiliser n'importe quelle largeur de donnée au bit près ; des opérateurs spécialisés (multiplications par des constantes (F. de Dinechin et al. 2019), opérateurs approchés (Sentieys et al. 2021), etc.) ; des mémoires de la taille nécessaire et avec autant d'accès que nécessaire ; etc. Certains choix portent sur l'organisation des calculs (tailles des blocs de calcul, flots de données, réutilisation des opérateurs et des données, *pipeline*, architectures systoliques, etc.) voire sur les différents algorithmes répondant au besoin (Y. Liang et al. 2019).

Les choix de conception ajoutent des degrés de liberté qui augmentent la taille de l'*espace de conception*, l'ensemble des solutions possibles. Tous les *points* de cet espace ne sont pas pertinents : ils ne respectent pas les contraintes de l'architecture du FPGA utilisé (bande passante, ressources, mémoire, etc.) ou ne sont pas sur le front de Pareto (d'autres points ont des qualités supérieures dans toutes les dimensions). Les points pertinents sont multiples et offrent des compromis entre différents objectifs à optimiser (débit, latence, efficacité énergétique, puissance, énergie, précision ou qualité, etc.). Par exemple, le choix

de la taille des mots offre un compromis entre qualité (due à la quantification des données) et performance et efficacité énergétique (dues aux opérateurs plus petits, aux mémoires plus petites, etc.).

Il faut cependant noter que les caractéristiques d'un point de l'espace de conception (comme les ressources utilisées, l'efficacité énergétique) ne peuvent pas toutes être connues facilement (sans créer l'architecture et effectuer tout ou partie du long processus de synthèse logique). Il est toutefois possible de les estimer en utilisant un modèle analytique afin de sélectionner des solutions proches du front de Pareto.

1.4.4 Intégration

Les FPGA peuvent être utilisés en autonomie (par exemple avec une interface réseau) ou conjointement avec un processeur hôte de la même manière que les GPU et les TPU. Dans ce dernier cas, le FPGA est utilisé par l'hôte comme *accélérateur matériel* pour les calculs les plus critiques. On peut retrouver le FPGA seul ou intégré avec les(s) processeur(s) sur un même système sur puce (*System On a Chip*, SOC), interconnectés par un bus mémoire. Par exemple, les FPGA haut de gamme de Xilinx (architecture Ultrascale) sont disponibles avec l'architecture Zynq (le FPGA et les six processeurs ARM sont interconnectés par un bus *Advanced Microcontroller Bus Architecture* (AMBA)) ou seuls. On retrouve notamment le FPGA seul sur une carte destinée aux centres de données (Alveo U280) permettant de le connecter à un système hôte via un bus PCI Express, comme un GPU.

1.5 Conclusion

Nous avons présenté dans ce chapitre différentes plateformes de calcul offrant différents compromis entre performance, efficacité énergétique et puissance. Pour une application donnée, la plateforme la plus pertinente sera choisie en fonction des contraintes de l'application (minimum de performance, maximum de puissance et thermique, maximum d'énergie, etc.) et des contraintes de développement (temps, coût). Toutes les plateformes et la technologie des transistors évoluent afin de permettre de meilleures performances et une meilleure efficacité énergétique au fur et à mesures des évolutions technologiques et des améliorations architecturales. L'avènement de nouveaux paradigmes de calcul et de nouvelles architectures pourra, dans le futur, rendre les applications existantes plus

performantes et plus efficaces ou répondre à des nouveaux besoins. Avec la technologie d'aujourd'hui, il est parfois possible d'aller plus loin en performance et efficacité énergétique pour une application donnée et une plateforme donnée grâce à la spéculation temporelle.

Chapitre 2

Spéculation temporelle pour FPGA

Dans le chapitre précédant, nous avons présenté le compromis entre les performances d'une plateforme de calcul, son efficacité énergétique et la puissance. Dans ce chapitre, nous montrons comment il est possible d'améliorer les performances et l'efficacité énergétique d'un accélérateur matériel basé sur un FPGA grâce à la spéculation temporelle. Celle-ci consiste à utiliser une fréquence supérieure à la fréquence maximale du circuit (*overclocking*) ou une tension en deçà de sa tension nominale (*undervolting*). Le circuit opère alors dans la *marge d'erreur* de fréquence et de tension : la correction des calculs n'est plus garantie. On peut toutefois ajouter des mécanismes de détection d'erreur et de tolérance aux fautes afin d'apporter cette garantie.

Afin de pouvoir mettre en œuvre la spéculation temporelle, la plateforme doit exposer un contrôle de la fréquence (ou de la tension) du circuit. C'est pourquoi les FPGA sont particulièrement pertinents, car ils permettent d'utiliser différentes horloges dans un même circuit, de modifier leur fréquence, de réaliser l'interface électronique entre deux domaines d'horloge (par exemple pour accéder à une mémoire externe ou pour communiquer), de paralléliser les calculs simplement (pour limiter l'impact de la détection d'erreur), etc.

Dans ce chapitre, nous définissons la spéculation temporelle et donnons le cadre dans lequel elle peut s'appliquer. Nous présentons aussi le contexte du domaine la tolérance aux fautes, dont des techniques de détection d'erreur pertinentes pour détecter les erreurs temporelles.

2.1 Spéculation temporelle

Dans cette section, nous commençons par expliquer brièvement ce qu'est la « fréquence maximale » d'un circuit et comment elle est calculée par les outils de synthèse. Nous montrons ensuite qu'utiliser une fréquence supérieure permet d'augmenter les performances et

l'efficacité énergétique d'un accélérateur matériel : c'est la spéculation temporelle. Enfin, nous présentons les différentes sources de variabilité qui obligent à utiliser un mécanisme de tolérance aux fautes pour pouvoir utiliser la spéculation temporelle.

2.1.1 Fréquence maximale d'un circuit

Dans un circuit séquentiel synchrone, les bascules du circuit échantillonnent les signaux à chaque front montant du signal d'horloge. La fréquence d'horloge maximale f_{max} à laquelle le circuit fonctionne correctement dépend du plus grand délai de propagation entre deux bascules. Ce délai dépend des transistors utilisés, des bascules utilisées, du nombre de transistors que le signal doit traverser, etc. mais aussi d'autres facteurs (voir section 2.1.3 ci-dessous). Les outils de synthèse déterminent la fréquence maximale à laquelle le circuit est garanti de fonctionner au cours de l'analyse statique temporelle (*Static Timing Analysis*, STA) (Salman et al. 2007). L'analyse prend donc en compte le circuit, la technologie utilisée, etc. et s'assure que le circuit fonctionne dans toutes les conditions prévues. Les outils considèrent donc, pour cette analyse, le cas le plus défavorable de toutes les conditions : le *pire cas*.

Plus précisément, la STA analyse un graphe orienté acyclique représentant le circuit (Kahng et al. 2011). Chaque nœud représente un élément du circuit (porte, fil), avec l'information de son délai de propagation. Chaque arête représente les connexions entre les différents éléments. La STA vérifie les contraintes pout chaque nœud. Notamment, elle calcule la *slack* : la différence entre le délai d'arrivée requis (dépendant de la fréquence demandée) et le délai calculé. Toutes les *slacks* doivent être positives pour que le circuit soit validé. De plus, elle vérifie les contraintes sur les éléments de stockage (bascules D, verrous) : pour que l'échantillonnage par ces éléments soit fiable, le signal doit être stable autour du front montant d'horloge comme illustré par la figure 2.1. La STA vérifie donc que le signal ne varie pas ni trop tard, ni trop tôt.

2.1.2 *Overclocking* et *Undervolting*

L'analyse statique détermine la fréquence maximale f_{max} en fonction de plusieurs paramètres dont la tension d'alimentation V_{DD}. Comme montré section 1.1, un circuit peut fonctionner selon plusieurs couples (f_{max}, V_{DD}). Les points de fonctionnement sont déterminés en fonction d'un modèle et sont les optima de Pareto d'après ce modèle : ce sont les points pour lesquels la fréquence ne peut être augmentée et la tension ne peut être

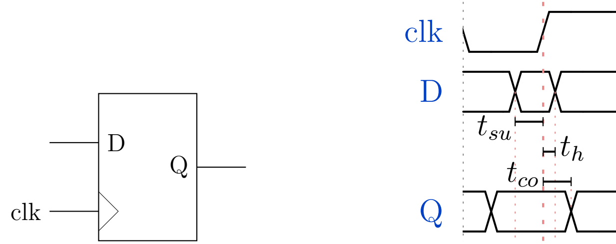


FIGURE 2.1 – Caractéristiques temporelles d'une bascule D : le signal `D` doit être stable avant le front montant (*setup time* : t_{su}) et après (*hold time* : t_h) pour être correctement échantillonné lors du front montant du signal d'horloge `clk`. La bascule présente aussi un délai de propagation (*clock-to-output* : t_{co}).

diminuée. (Il s'agit aussi des optima de Pareto pour la puissance et les performances.) Le modèle utilisé provient de l'analyse statique et considère donc le pire cas. Les points de fonctionnement ne sont donc pas forcément optimaux au sens de Pareto dans la réalité : on peut parfois augmenter la fréquence ou baisser la tension d'alimentation en gardant le circuit fonctionnel.

Il est possible de dévier d'un point de fonctionnement dans quatre directions, comme illustré par la figure 2.2 :

— *underclocking* : seule la fréquence est diminuée ;
— *overvolting* : seule la tension d'alimentation est augmentée ;
— *overclocking* : seule la fréquence est augmentée ;
— *undervolting* : seule la tension d'alimentation est diminuée.

L'*underclocking* réduit la puissance dynamique (voir l'équation 1.2 page 14) proportionnellement aux performances. Cependant, la puissance statique est inchangée. L'*overvolting* augmente la puissance sans augmenter les performances. Ces deux options donnent donc des points de fonctionnement moins intéressants, suboptimaux. Elles reviennent simplement à choisir une fréquence inférieure à f_{max} pour une tension d'alimentation donnée.

L'*overclocking* accroît les performances proportionnellement à la puissance dynamique, sans augmenter la puissance statique. La puissance est augmentée mais l'énergie consommée par le circuit pour réaliser un calcul donné diminue. En effet, la puissance dynamique est proportionnelle à la fréquence, mais pas la puissance statique. S'il faut N cycles pour réaliser le calcul, celui-ci prendra $\frac{N}{f}$ secondes, donc l'énergie E (le produit du temps par la puissance) est :

$$E = \frac{N}{f}\left(P_{stat} + P_{dyna}\right) = \frac{N}{f}\left(P_{stat} + \alpha C f V_{DD}^2\right) = N\left(\frac{P_{stat}}{f} + \alpha C V_{DD}^2\right)$$

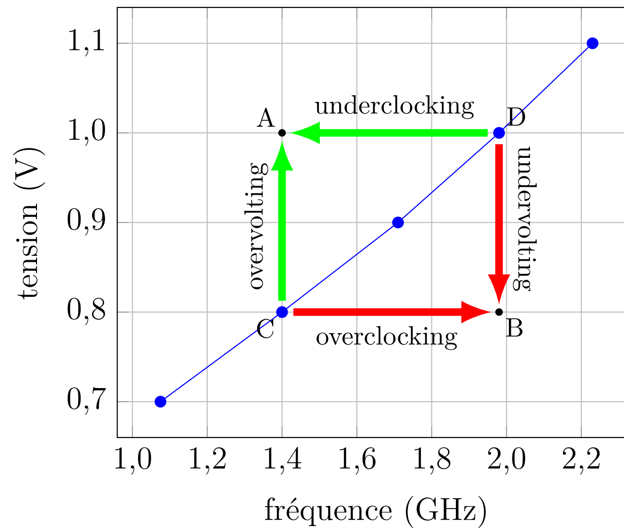


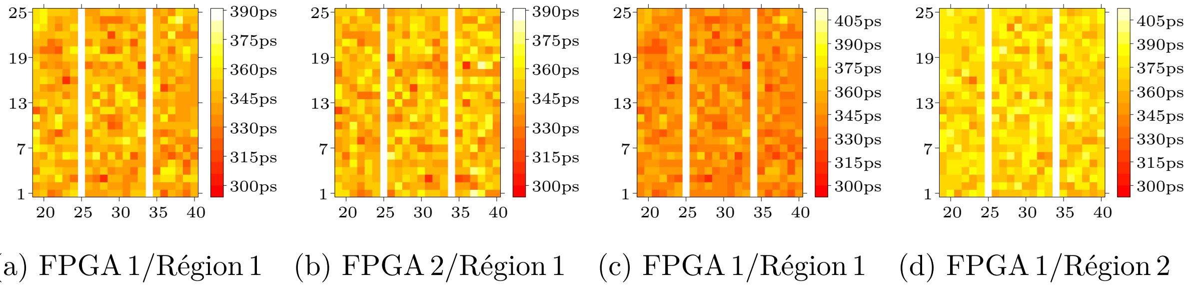
FIGURE 2.2 – Déviations des points de fonctionnement en tension et fréquence. L'*underclocking* et l'*overvolting* (en vert, comme le point de fonctionnement A) sont sûrs mais suboptimaux. L'*overclocking* et l'*undervolting* (en rouge, comme le point de fonctionnement B) améliorent les performances et l'efficacité énergétique sans garantie de bon fonctionnement. Un point de fonctionnement irrégulier sous la courbe peut être *considéré* comme de l'*overclocking* ou de l'*underclocking*, selon le point de fonctionnement de référence choisi (par exemple C ou D).

L'énergie consommée pour réaliser ce calcul diminue donc lorsqu'on augmente la fréquence : l'efficacité énergétique est alors améliorée.

Comme pour les deux premiers cas de figure, l'*undervolting* revient au même que l'*overclocking* : baisser la tension d'alimentation réduit les puissances statique et dynamique sans modifier les performances ce qui augmente l'efficacité énergétique.

Si l'*underclocking* et l'*overvolting* sont suboptimaux mais ne présentent aucun danger, l'*overclocking* et l'*undervolting* améliorent l'efficacité énergétique en utilisant un point de fonctionnement irrégulier. Ce point de fonctionnement étant placé au-delà du résultat de l'analyse statique, le circuit n'a plus la garantie de fonctionner correctement. C'est pour cela que nous nommons cette technique « spéculation temporelle » : il s'agit de *parier* sur les capacités temporelles du circuit pour assurer un fonctionnement correct.

La fréquence f_{max} est déterminée par les délais les plus longs, aussi appelés critiques. Les chemins les plus longs seraient a priori les plus impactés, mais il est difficile de s'assurer que seuls ceux-ci seront impactés ou qu'ils seront impactés avant d'autres chemins (voir les résultats expérimentaux section 4.3.8).



(a) FPGA 1/Région 1 (b) FPGA 2/Région 1 (c) FPGA 1/Région 1 (d) FPGA 1/Région 2

FIGURE 2.3 – Variabilité inter-puces et intra-puces. Les cartes de chaleurs représentent le temps de propagation d'une grille de 21 × 25 LAB de FPGA Altera. Les figures (a) et (b) représentent les délais de la même région de deux FPGA différents. Les figures (b) ((a) avec une échelle différente) et (d) représentent les délais de deux régions différentes du même FPGA. Figures de Gojman et al. (GOJMAN et al. 2013).

2.1.3 Variabilité

Le comportement temporel d'un accélérateur matériel dépend de nombreuses paramètres et est donc difficile à prévoir. Comme expliqué ci-dessus, les outils de synthèse déterminent la fréquence f_{max} selon des plages nominales de ces paramètres. Ils prennent en compte la plage de tension d'alimentation, la plage de température, la qualité de fabrication du circuit, la durée de vie prévue du produit (prenant en compte le vieillissement), etc. ; et utilisent à chaque fois le pire cas possible. Si une fréquence inférieure ou égale à f_{max} garantit le fonctionnement correct de l'accélérateur en toutes circonstances, celui-ci peut donc également fonctionner à une fréquence supérieure lorsque le pire cas des paramètres n'est pas atteint. La connaissance de cette fréquence seuil en dessous de laquelle le circuit fonctionne est essentielle pour utiliser l'*overclocking* tout en évitant les erreurs. À cause des sources de *variabilité*, cette fréquence peut évoluer au cours du temps et être difficile à déterminer.

On peut distinguer la variabilité en deux catégories : inter-puces et intra-puces. Lors de la fabrication de circuits intégrés, la gravure des transistors n'est pas suffisamment précise pour obtenir des circuits rigoureusement identiques (*process variation*). Les caractéristiques physiques des transistors diffèrent donc légèrement d'un circuit à l'autre, mais aussi au sein d'un circuit : largeur, longueur, etc. Cela implique que leur vitesse de commutation et l'énergie nécessaire varie entre circuits. La variation est d'autant plus marquée que la gravure est fine. Or, le comportement temporel des FPGA est très sensible à ces variations (Edward STOTT et al. 2013). Gojman et al. ont mesuré les temps de propagation d'une partie de 18 FPGA Cyclone III de Altera (gravés en 65 nm) avec une

résolution de 3,2 ps (Gojman et al. 2013). Ils montrent que ces délais ont un coefficient de variation $\sigma/\mu = 4\%$ à 1,2 V. Ils ont aussi montré qu'en baissant la tension d'alimentation de 1,2 V à 0,9 V d'un Cyclone IV (gravé en 60 nm), le coefficient de variation passe de $\sigma/\mu = 4.3\%$ $\sigma/\mu = 5.8\%$. L'*undervolting* (et donc certainement l'*overclocking*) empire donc la variabilité inter-puces. La figure 2.3 montre un exemple de mesures de temps de propagation entre deux FPGA, ainsi qu'entre deux régions (censées être identiques) d'un même FPGA.

Les sources de variabilité intra-puces, environnementales ou dynamiques, modifient le comportement physique, donc temporel d'un circuit électronique au cours du temps :

— La tension d'alimentation. Celle-ci est normalement stabilisée mais peut dériver ou il peut y avoir des chutes de tension lors d'appels de courant (Rebaud et al. 2011) ;
— Le vieillissement, voir section 2.2.2 ;
— La température. Levine et al. montrent que la fréquence maximale mesurée d'un accélérateur matériel sans observer d'erreur varie en fonction de la température : ils mesurent 325 MHz à 8 °C contre 297 MHz à 130 °C (les points intermédiaires montrent une variation monotone), soit une variation totale de 8 % (J. M. Levine et al. 2012). En revanche, Salami et al. montrent qu'un CNN sur FPGA obtient de meilleurs résultats en *undervolting* à une température plus élevée (50 °C) qu'à une température basse (34 °C) (Salami et al. 2020). La température a donc une influence, peut-être difficile à prévoir, sur le comportement temporel d'un FPGA ;
— Les données traitées. Le temps nécessaire pour obtenir le résultat d'un opérateur peut dépendre de ses opérandes ou de la variation de ceux-ci. Par exemple, pour un simple opérateur à propagation de retenue, si les opérandes ne varient pas, le résultat est prêt immédiatement sans attendre la propagation jusqu'au poids fort. Dans ce cas, la fréquence maximale mesurée sans erreur peut être plus élevée que dans le cas moyen.

Ces sources de variabilité intra-puces empêchent d'utiliser un système de calibration ou une mesure hors-ligne pour obtenir la fréquence maximale à laquelle on n'obtient pas d'erreur. *Pour utiliser l'*overclocking *ou l'*undervolting *de façon sûre, il est donc nécessaire de garantir l'absence d'erreur via un mécanisme de tolérance aux fautes.*

2.2 Tolérance aux fautes

Cette section présente les éléments clés pour comprendre le domaine de la tolérance aux fautes de manière générale puis plus précisément dans le cadre de la spéculation temporelle. Nous commençons par définir ce qu'est une faute, une erreur ou une panne ; présentons quelques classifications utiles et listons les phénomènes physiques qui peuvent mener à une faute. Nous définissons ensuite ce que sont les erreurs temporelles. Enfin, nous comparons différents mécanismes de tolérance aux fautes existants avant de conclure.

2.2.1 Fautes, erreurs, pannes

Une *faute* est un évènement pouvant potentiellement modifier l'état ou le comportement d'un système par rapport à son fonctionnement normal.

Les fautes peuvent être de différent types. Selon leur type, différents mécanismes de détection et de correction d'erreur peuvent être pertinents ou non. Plusieurs qualificatifs permettent de classer les fautes (Avizienis, Laprie et Randell 2001), dont :

— leur cause : les fautes peuvent être dues à défaut de conception (par exemple, un programme contenant un bogue) ; à un phénomène physique (par exemple, un rayonnement à haute énergie) ou à une interaction (par exemple, une attaque malicieuse) ;

— leur persistance : les *fautes transitoires* sont des fautes apparaissant provisoirement, impactant le bon fonctionnement pendant un court moment. Par exemple, si une bascule contient une valeur erronée pendant un cycle, la bascule pourra échantillonner une valeur correcte au prochain cycle. Au contraire, les *fautes permanentes* subsistent une fois qu'elles surviennent. C'est le cas des fautes de types *stuck-at-zero* ou *stuck-at-one* : un signal est bloqué à zéro ou à un. Elles peuvent venir d'une défaillance du circuit, par exemple d'un défaut de fabrication ;

— leur conséquence : les fautes peuvent être *actives* ou *masquées*. Elles provoquent une *erreur* dans le premier cas, mais pas dans le second. Par exemple, une faute de type *stuck-at-one* est active si le signal devrait être à 0 et masquée si le signal devrait être à 1.

De manière orthogonale, les fautes peuvent être modélisées à différents niveaux. Si une faute se produit toujours au niveau physique, elle peut être considérée selon plusieurs niveaux d'abstraction, en particulier pour les FPGA (E. Stott, Sedcole et P. Cheung 2010) :

— logique : la faute est modélisée au niveau d'un fil particulier ;
— *fabric* : la faute est modélisée au niveau d'un composant du FPGA (LUT, registre, etc.) ;
— grille : la faute est modélisée au niveau du FPGA lui-même (par exemple dans le contexte d'une grappe de FPGA) ;
— application : la faute est modélisée au niveau applicatif.

Une même cause physique peut provoquer une faute différente selon le niveau d'abstraction choisi. Par exemple, un rayon cosmique impactant la *configuration* d'un FPGA peut être considéré comme une faute permanente (niveau logique ou *fabric*) ou une faute transitoire (niveau grille ou application). Dans ce dernier cas, une reconfiguration suffit à rétablir le fonctionnement correct (Hiemstra et Kirischian 2012).

Pour les circuits numériques, une erreur peut être qualifiée plus spécifiquement :
— erreur simple bit : un seul bit est erroné sur une donnée (communiquée ou stockée) ;
— erreur multi-bit : plusieurs bits sont erronés simultanément.

Enfin, les erreurs peuvent mener à une *panne*, qui survient lorsque le système dévie de son fonctionnement normal, ne rend plus le service qui est attendu. Les erreurs peuvent aussi être masquées, par exemple si la donnée erronée n'est finalement pas utilisée.

La tolérance aux fautes est un moyen d'éviter les pannes *en présence* de fautes actives. Son fonctionnement général peut être spécifié en deux parties (Avizienis, Laprie et Randell 2001) la *détection d'erreur* et le *rétablissement* (*recovery*). Le rétablissement se déroule lui-même en deux phases : la *gestion de l'erreur* et la *gestion de la faute*. La gestion de l'erreur permet de remettre le système dans un état sans erreur. La gestion de la faute permet d'éviter que cette faute se produise à nouveau.

La classification des fautes et des erreurs permet de les modéliser, de spécifier sous quelles conditions un système de tolérance aux fautes fonctionne ou non, etc.

2.2.2 Phénomènes physiques pouvant provoquer une faute

Nous nous intéressons particulièrement aux fautes d'origine physique. Connaître quels phénomènes physiques sont susceptibles de survenir permet de modéliser correctement les fautes d'un système.

Perturbation par une particule isolée (*Single-Event Upset*, SEU) Des particules élémentaires (comm un neutron ou un proton) à haute énergie, ou « rayons cosmiques », peuvent irradier le circuit et induire une charge électrique. Cette charge peut

changer l'état d'une bascule et donc provoquer une faute (Binder, Smith et Holman 1975).

Avec les améliorations technologiques, notamment la miniaturisation des transistors et l'augmentation de leur nombre dans un circuit, les plateformes, dont les FPGA, deviennent de plus en plus sensibles à ces perturbations (Davis et P. Y. K. Cheung 2014). Le problème est aggravé dans des conditions extrêmes comme l'aviation et le spatial. On peut observer en moyenne 14.4 SEU par jour sur un FPGA en orbite basse terrestre (Quinn et al. 2012).

Vieillissement Les transistors des circuits numériques « vieillissent », c'est-à-dire se dégradent avec le temps. Plusieurs phénomènes induisent ce vieillissement : les effets *Hot Carrier Injection (HCI)*, *Positive Bias Temperature Instability (PBTI)* et *Negative Bias Temperature Instability (NBTI)* (Kiamehr, Firouzi et Tahoori 2013). Ces effets augmentent la tension de seuil V_{th} des transistors et, par conséquent, augmentent leur délai de commutation (voir équation 1.3 page 15). Si les délais de propagation du circuit augmentent trop, alors les signaux ne seront parfois pas prêts pour l'échantillonnage au front d'horloge suivant, causant une faute temporelle.

Fabrication La fabrication des circuits électroniques est un processus complexe et précis. En conséquence, les caractéristiques des circuits et de leurs transistors sont variables et peu prédictibles (*process variation*). Seule une partie des circuits produits est complètement fonctionnelle (le *yield*). Leur bon fonctionnement est donc testé à plusieurs niveaux. La variation dans le processus de fabrication induit une variation dans les caractéristiques des circuits, même parmi les circuits qui fonctionnent correctement. Dans ses spécifications, le constructeur garantit donc des caractéristiques *minimales* pour un ensemble de circuits [1]. En pratique, une partie des circuits a donc des caractéristiques meilleures que ces spécifications. L'utilisateur peut utiliser un circuit dans des conditions hors spécification, mais, comme le fonctionnement n'est plus garanti, il y a un risque de faute et ce risque est variable. De plus, il est possible qu'une partie du circuit ne soit pas correctement testé et que celui-ci ne se comporte pas comme spécifié ce qui peut provoquer des fautes, même dans des conditions normales d'utilisation.

1. Le constructeur peut mesurer (ou tester) certaines de ces variations afin d'augmenter les spécifications minimales pour une partie des produits. Par exemple, il peut mesurer les caractéristiques temporelles des circuits afin de les trier et les vendre sous différentes classes plus ou moins rapides (*speed grades*). Les circuits présentent malgré tout des variabilités, même au sein d'une classe.

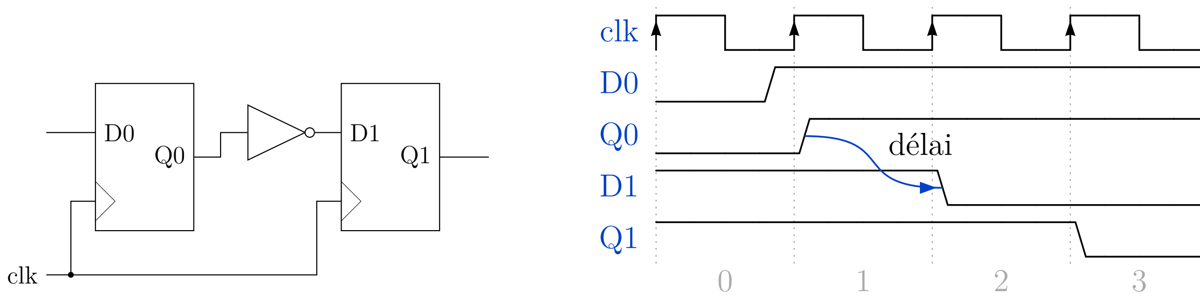


Figure 2.4 – Exemple d'erreur temporelle. Le chronogramme, à droite, représente l'évolution des signaux d'entrée et de sorties des deux bascules D du circuit représenté à gauche. Si le délai de propagation d'un circuit combinatoire entre deux bascules (ici une porte NON entre `Q0` et `D1`) est trop long, la bascule peut échantillonner une valeur erronée. Au cycle 2, le signal `Q1` prend pour valeur 1 plutôt que 0 à cause du délai de propagation.

2.2.3 Erreurs temporelles

Le concepteur d'un circuit choisit normalement la fréquence de fonctionnement du circuit pour être inférieure ou égale à la fréquence résultante de l'analyse, f_{max}. Dans les faits, si les conditions ne sont pas toutes défavorables, le circuit peut fonctionner à une fréquence supérieure. Dans ce cas, le fonctionnement correct du circuit n'est plus garanti et devrait être vérifié par d'autres moyens.

Des fautes peuvent donc survenir quand la fréquence sélectionnée est trop élevée pour les conditions d'utilisation courantes. Plus précisément, quand le délai de propagation d'un circuit combinatoire est plus long que la période d'horloge. Ces fautes sont transitoires : elles surviennent pendant un cycle d'horloge. Elles peuvent perdurer sur plusieurs cycles d'horloge mais ne sont pas persistantes. Elles sont d'origine physique : elles sont dues au comportement temporel du circuit. Elles peuvent être gérées en réajustant la fréquence ou la tension pour éviter d'autres fautes de ce type. Ces fautes peuvent provoquer des erreurs qu'on appelle « erreurs temporelles ». La figure 2.4 illustre ce phénomène.

2.2.4 Comparaison des mécanismes de tolérance aux fautes existants

Cette section présente et compare quelques approches permettant la tolérance aux fautes.

2.2.4.1 Redondance : duplication et triplication

Pour protéger un calcul, une communication ou une mémoire, il est possible de simplement multiplier les instances de l'élément à protéger. En dupliquant (*DMR* (Dual Modular Redundancy)) l'élément protégé et en comparant les deux résultats, il devient possible de détecter une faute. En utilisant trois instances (*TMR* (Triple Modular Redundancy)), il devient aussi possible de corriger une faute (ou plus précisément, de la masquer) en utilisant un vote majoritaire. Les instances peuvent être réparties dans le temps (effectuer plusieurs calculs à la suite sur le même matériel, communiquer plusieurs fois la même information, etc.) ou dans l'espace (multiplier le matériel, les instances sont exécutées parallèlement). Cette approche se fonde sur l'hypothèse d'une faute unique : si deux fautes identiques surviennent, celles-ci peuvent ne pas être détectées, que ce soit pour la duplication ou la triplication. On peut rechercher à maximiser les différences entre les multiples instances afin de limiter ce risque (différentes technologies, différentes équipes conception ou de développement, etc.).

L'avantage de cette approche est son universalité : elle peut s'appliquer à n'importe quel type de calcul, aux communications sur n'importe quel support ou à la mémorisation ; elle s'applique à n'importe quelle technologie ou à n'importe quel niveau d'abstraction (de la porte logique à l'application). Son inconvénient est le surcoût : 100 % pour le DMR et 200 % pour le TMR. Ce surcoût peut être en temps ou en matériel, mais il est aussi comptabilisé en énergie.

2.2.4.2 Codes correcteurs d'erreur

Des mécanismes de correction d'erreur ont été créés pour protéger les communications (Poli et Huguet 1992). La chaîne de bit à transmettre provient de l'*émetteur*, est codée par le *codeur* avant d'être transmise sur un canal de communication puis décodée par le *décodeur* avant d'être fournie au *récepteur*. Ces mécanismes permettent de détecter un certain nombre de bits erronés durant la transmission, ainsi que de corriger un certain nombre d'entre grâce au codage de la chaîne de bits. Cette section présente certains de ces codes par souci d'exhaustivité sans rentrer dans les détails.

Bits de parité Le moyen le plus simple de vérifier si un seul des bits d'un mot binaire est erroné est de vérifier si sa parité a changé. En effet, le changement d'un seul bit d'un mot binaire inverse la parité du mot. Par exemple, en prenant la convention de parité *paire*, on ajoutera un bit au codage du mot afin que le nombre de bits à 1 du mot codé

(soit la somme de ses bits sur 1 bit) soit pair. Par exemple, pour le mot 1010101, on ajoutera le bit 0. Ce mécanisme simple permet de détecter une erreur et d'en corriger aucune.

Sommes de contrôle Les sommes de contrôle sont une généralisation du bit de parité : elles réalisent la somme (au sens large) du mot sur plus qu'un bit. La somme de contrôle est donc fonction de l'ensemble du mot. Si un ou plusieurs bits du mot change, la somme de contrôle change (en dehors des cas de collision). Les algorithmes de sommes de contrôle sont construits pour différents objectifs et pour respecter différentes propriétés selon l'application (détection d'erreur, empreinte numérique, fonction de hachage cryptographique ou non, etc.) : risque de collision (uniformité), efficacité de l'implémentation logicielle ou matérielle, efficacité du stockage, etc. Le codage consiste à calculer la somme de contrôle et à l'ajouter à la chaîne de bits à transmettre. Le décodage consiste à calculer la somme de contrôle sur les bits utiles reçus et de vérifier si elle est égale à la somme de contrôle reçue. Tout comme le bit de parité, ce mécanisme ne permet pas la correction d'erreur.

Codes de Hamming Les codes de Hamming (Hamming 1950), en revanche, permettent la correction d'erreur. Ces codes calculent plusieurs *bits de parité* pour des sous-ensembles des bits du mot à coder de manière à pouvoir détecter deux bits erronés ou corriger un seul bit erroné. Par exemple, le code de Hamming $(7,4)$ ajoute trois bits de parité (r_1, r_2, r_3) pour quatre bits de données (u_1, u_2, u_3, u_4). C'est donc un code *systématique* : il laisse intacte la chaîne de bit originale, la redondance est incluse uniquement dans les bits supplémentaires :

$$r_1 = u_1 + u_2 + u_4$$
$$r_2 = u_1 + u_3 + u_4$$
$$r_3 = u_2 + u_3 + u_4$$

Pour vérifier qu'il n'y a pas d'erreur sur les 7 bits transmis, le décodeur vérifie les conditions de parité (les trois dans le cas $(7,4)$). Si les conditions de parité ne sont pas respectées, il y a alors une ou plusieurs erreurs de transmission. Dans notre exemple, en considérant qu'un seul bit est erroné :

- si r_1 et r_2 sont incohérents : le bit u_1 est inversé ;
- si r_1 et r_3 sont incohérents : le bit u_2 est inversé ;

— si r_2 et r_3 sont incohérents : le bit u_3 est inversé ;
— si r_1, r_2 et r_3 sont incohérents : le bit u_4 est inversé ;
— si r_1, r_2 ou r_3 est incohérent : l'erreur a eu lieu sur un des bits de parité.

Le code permet donc de corriger une erreur dans tous les cas.

Les codes de Hamming sont *parfaits*, c'est-à-dire qu'ils ne contiennent aucune redondance superflue pour leurs capacités de détection et de correction.

Ces codes, ainsi que de nombreux autres non présentés ici, peuvent aussi être appliqués à la mémorisation d'information. En effet, on peut considérer une mémoire comme un canal de communication : l'écriture en mémoire équivaut à l'émetteur et la lecture au récepteur. Comme des fautes peuvent survenir dans les composants mémoires (notamment des SEU), il est courant de protéger les données avec des codes correcteurs d'erreur simples. C'est le cas, par exemple, des barrettes de mémoire à code correcteur d'erreurs ou des caches et fichiers de registres de certains processeurs. Les blocs RAM des FPGA Xilinx *7 Series* intègrent aussi un mécanisme de détection et de correction pouvant être activé (« 7 Series FPGAs Memory Resources User Guide » 2019). Dans ce cas, chacune des 512 lignes de 72 bits de la mémoire utilise 8 bits pour la redondance. Le contrôleur de la mémoire peut détecter deux erreurs et corriger une erreur.

Cependant, les codes correcteurs d'erreur ne peuvent pas être utilisés pour protéger des erreurs de *calculs*. Pour cela, il existe d'autres mécanismes qui sont présentés dans les sections suivantes.

2.2.4.3 Redondance basée sur l'arithmétique modulaire

À la place de simplement dupliquer le calcul pour détecter les erreurs, il est possible d'utiliser une autre forme de redondance basée sur des propriétés arithmétiques des opérations utilisées. Cette approche est intéressante dans le cas où la forme de redondance est moins coûteuse (en temps et en ressources).

Par exemple, il est possible de remplacer l'arithmétique utilisée, classiquement une arithmétique modulaire modulo 2^n pour une représentation en binaire sur n bits (appelée « binaire » ci-après), par une arithmétique modulaire avec un diviseur plus faible $M < 2^n$. Les mots seront alors représentés sur $\lceil log_2(M) \rceil$ bits, nécessitant moins de mémoire et les opérateurs peuvent être plus simples.

Il est possible d'utiliser plusieurs diviseurs parallèlement, ce qui donne un Système Modulaire de Représentation (*Residue Number System*, RNS) (Mohan 2002). Le cas

d'un diviseur unique en est un cas dégénéré.

Utiliser plusieurs diviseurs (par exemple $\{3, 5\}$) plutôt qu'un seul diviseur plus grand (par exemple $\{15\}$) nécessite plus d'opérateurs, mais chaque opérateur est plus simple. Les RNS sont par exemple utilisés pour réaliser des multiplications-accumulations sur des mots de grande taille ou en cryptographie pour réaliser des multiplications modulaires (MOHAN 2016).

Le choix des diviseurs est critique : certains bénéficient d'optimisations algorithmiques qui permettent de réduire le coût des conversions et des opérateurs (MOHAN 2016). La conversion d'un nombre binaire en multiples restes (appelée conversion directe) peut être réalisée de plusieurs manières (MOHAN 2016). Par exemple, les diviseurs $\{2^k - 1, 2^k + 1\}$ permettent d'obtenir des algorithmes particulièrement efficaces pour la conversion. La conversion vers le reste de la division par $2^k - 1$ peut se réaliser d'une manière similaire à la « preuve par 9 » utilisée en base 10 (CAMPBELL et al. 2015) : le mot d'entrée, représenté en binaire, peut-être interprété en base 2^k : chaque chiffre de la base est représenté en binaire par k bits. Ces chiffres sont sommés par paire, avec un opérateur calculant la somme modulo $2^k - 1$. Cet opérateur est un additionneur classique sauf pour le résultat $2^k - 1$ qui devient 0. Ces résultats sont ensuite sommés, toujours modulo $2^k - 1$, par un arbre de réduction, jusqu'au résultat final sur k bits. Ces opérateurs peuvent être légèrement optimisés parce que leurs opérandes ne peuvent pas être $2^k - 1$. Une telle conversion nécessite $\left\lceil log_2\left(\frac{n}{2}\right)\right\rceil$ étages de sommes et $\frac{n}{2} - 1$ opérateurs, où n est le nombre de bits du mot d'entrée. Cette opération est représentée par la figure 2.5. Il existe des architectures de convertisseurs pour d'autres diviseurs (MOHAN 2016).

La conversion inverse, c'est-à-dire retrouver le nombre original, binaire, depuis un ensemble de restes, est possible. Entre autres, si les diviseurs sont choisis premiers entre eux (ce qui est le cas le plus intéressant), le théorème des restes chinois stipule l'existence et l'unicité d'une solution à la conversion inverse (MOHAN 2016). La solution est un nombre entre zéro et le produit des diviseurs. (Un nombre supérieur à ce produit ne peut pas être retrouvé, mais la conversion donnera le reste de la division de ce nombre par ce produit.) Il faut donc choisir des diviseurs assez grands et en nombre suffisant selon la valeur maximale du nombre original. Cependant, calculer ce résultat est possible, mais coûteux (MOLAHOSSEINI et al. 2010), c'est pourquoi cette opération n'est pas utilisée dans le cadre de la tolérance aux fautes. Au lieu de retrouver la solution, on préfère utiliser deux conversions directes et comparer les restes. De plus, cela enlève la contrainte sur le nombre et la grandeur des diviseurs pour que la conversion inverse donne le nombre

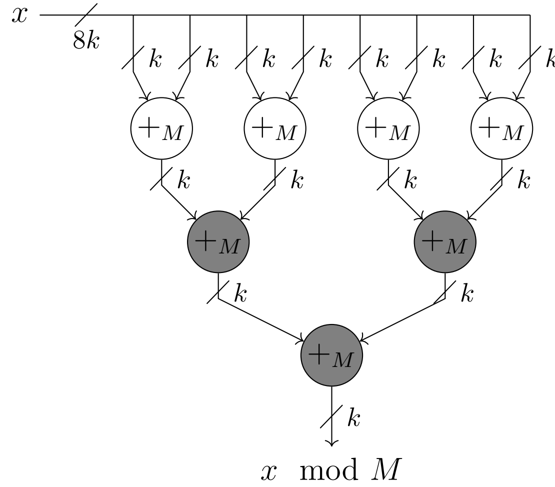


Figure 2.5 – Conversion d'un nombre binaire en reste de la division par $M = 2^k - 1$. Les opérateurs représentés en blanc sont des additionneurs classiques, sauf pour le résultat $2^k - 1$ qui est remplacé par 0. Les opérateurs représentés en gris ont été optimisés davantage parce que leurs opérandes ne peuvent pas être $2^k - 1$. Le même principe peut-être appliqué à n'importe quelle largeur de mot d'entrée (Campbell et al. 2015).

original. Le choix des diviseurs est alors simplement un compromis entre le coût de la tolérance aux fautes et sa couverture. La détection d'erreur se déroule alors de la manière suivante (la figure 2.6 donne un exemple de circuit) :

1. chaque entrée du calcul est convertie dans sa représentation modulaire ;
2. les opérations du calcul sont effectuées par les opérateurs classiques ;
3. parallèlement, les opérations redondantes sont effectuées par les opérateurs modulaires ;
4. les résultats du calcul sont convertis dans leurs représentations modulaires ;
5. les résultats convertis et les résultats des opérations redondantes sont comparés.

Il faut que toutes les opérations du calcul satisfassent cette propriété pour que cette approche fonctionne : le reste du résultat de l'opération doit être égal au reste du résultat de l'opération redondante. Autrement dit, les résultats des deux calculs doivent toujours être égaux dans le cas où il n'y a pas de faute. C'est le cas de l'addition, la soustraction et la multiplication. Par exemple, si le diviseur choisi est 3, l'opération $7 + 11 = 18$ serait protégée par l'opération redondante $1 + 2 \equiv 0 \mod 3$ parce que 7 et 11 sont congrus à 1 et 2 modulo 3. On vérifie bien que 18 est congru à 0 modulo 3. Les opérateurs modulaires d'addition et de multiplication peuvent être optimisés pour une implémentation

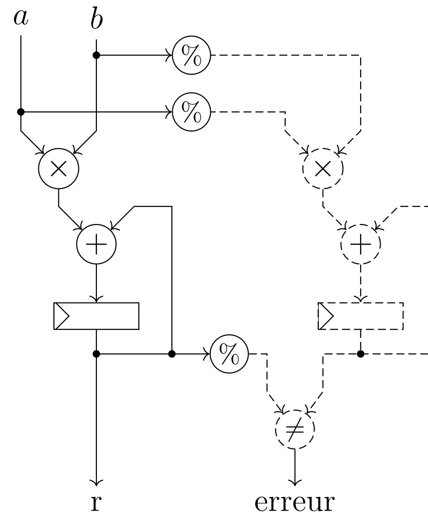


FIGURE 2.6 – Exemple de détection d'erreur basée sur l'arithmétique modulaire pour un opérateur de multiplication-accumulation. Le chemin de donnée est dupliqué par des opérateurs modulaires. Les opérateurs notés % calculent le(s) reste(s) de leur entrée. Les lignes pleines représentent les chemins de données en base classique, les lignes pointillées les chemins de données dans la (les) base(s) redondante(s).

matérielle (PIESTRAK 1994).

En cas d'erreur, l'objectif est de maximiser les chances de détection, ce qui dépend des diviseurs choisis. En effet, l'erreur n'est pas détectée si la différence entre le bon résultat et le résultat erroné est un multiple du produit des diviseurs choisis. Dans ce cas, la comparaison des restes donnera un faux positif. Dans l'exemple précédent, si le calcul donne le résultat erroné 36, alors le reste est aussi 0 et l'erreur n'est pas détectée. Le choix des diviseurs offre donc un compromis entre le coût de la détection d'erreur et le taux de couverture d'erreurs.

La comparaison des restes peut être effectuée à chaque opérateur (CAMPBELL et al. 2015) ou sur un calcul complet (PIESTRAK et PATRONIK 2014) si tous les opérateurs du calcul le permettent. Ces possibilités offrent un compromis entre le coût de la détection d'erreur et la latence entre l'occurrence d'une erreur et sa détection.

2.2.4.4 Détection basée sur le double échantillonnage

Il est possible de détecter les erreurs temporelles grâce au double échantillonnage. Cette méthode se base sur l'observation d'un signal à un grain plus fin que l'horloge du circuit afin de suivre son évolution plus précisément. Les observations (ou échantillons)

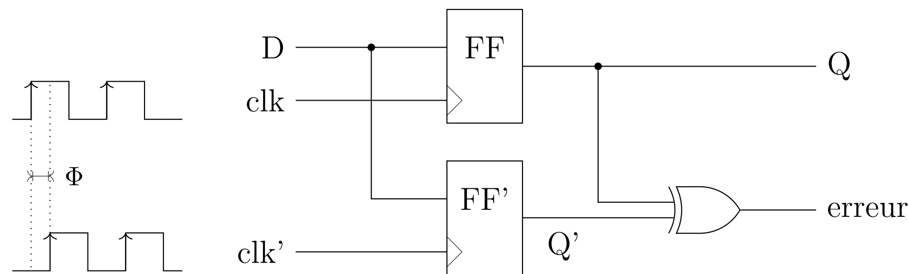


FIGURE 2.7 – Bascule Razor (ERNST et al. 2003). Les deux bascules sont contrôlées par deux horloges (`clk` et `clk'`) ayant la même fréquence mais en décalage de phase. La sortie `Q` de la bascule principale (`FF`) est utilisée normalement dans le circuit et pour détecter une erreur temporelle. La sortie `Q'` de la bascule secondaire (`FF'`) est utilisée pour détecter une erreur temporelle et fournit la bonne valeur en cas d'erreur.

supplémentaires permettent de savoir si le signal a évolué avant ou après le front montant de l'horloge. Elles permettent donc de déduire le comportement temporel du circuit. Plusieurs techniques utilisent ce principe de double échantillonnage, afin d'éviter, de détecter ou de corriger les erreurs temporelles.

Razor La première de ces techniques, Razor, a été mise en œuvre pour réduire la consommation énergétique de processeurs utilisant un *pipeline* (ERNST et al. 2003). Elle permet par exemple d'augmenter l'efficacité énergétique d'un processeur ARM Cortex-M3 de 60 % et son débit de 100 % (FOJTIK et al. 2013). La tension de fonctionnement du processeur est contrôlée dynamiquement en fonction du taux d'erreur temporelles détectées. Cela permet de minimiser la tension selon les contraintes courantes (température, données, etc.) et donc de réduire la consommation d'énergie.

L'approche se base sur des « bascules Razor » (voir figure 2.7). Les bascules Razor remplacent les bascules D classiques des registres du *pipeline* du processeur pour les protéger des erreurs temporelles. Elles contiennent deux bascules D qui sont contrôlées par deux horloges déphasées. Ces paires de bascules mettent donc en œuvre le double échantillonnage en échantillonnant le même signal à deux instants distincts. Si la fréquence de fonctionnement du circuit est trop élevée et que le signal n'est pas prêt pour l'échantillonnage de la bascule principale, celle-ci enregistre une valeur erronée. En revanche, la bascule déphasée échantillonne le signal plus tardivement. Si le déphasage est suffisant comparé au délai de propagation du signal, la bascule déphasée enregistre la bonne valeur (les niveaux de tensions sont choisis pour assurer que ce soit le cas). Dans ce cas, les deux bascules présentent deux valeurs différentes. L'erreur peut donc être détectée en

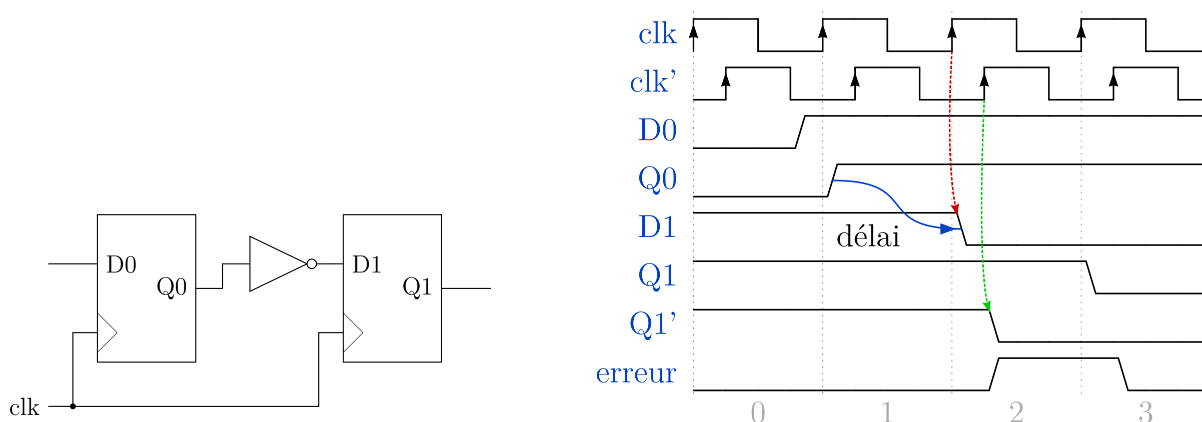


Figure 2.8 – Détection d'erreur temporelle avec Razor sur l'exemple de la figure 2.4. La seconde bascule est remplacée par une bascule Razor. Le délai de propagation est trop long et le signal n'est pas correctement échantillonné par la bascule principale (flèche rouge), mais il l'est par la bascule secondaire (flèche verte). L'erreur est donc détectée grâce à la différence entre `Q1` et `Q1'`.

comparant les sorties des bascules avec une porte XOR. Ce cas de figure est illustré par la figure 2.8.

Lorsque la sortie de la porte XOR est vraie, on sait que :

— une erreur temporelle est survenue ;

— la bascule déphasée contient la bonne valeur.

Dans ce cas, le contrôleur du processeur insère une bulle dans le *pipeline* et tous les étages du *pipeline* sont calculés à partir des valeurs enregistrées dans les bascules déphasées ce qui permet de corriger l'erreur. Il n'y a donc pas besoin d'autre mécanisme de correction. De plus, ce mécanisme assure qu'une même erreur ne peut se reproduire indéfiniment : malgré la bulle, le *pipeline* avance forcément. La pénalité d'une erreur temporelle est d'un cycle processeur.

Online Slack Measurement Contrairement à Razor qui permet de détecter les erreurs temporelles lorsqu'elles surviennent, cette approche se base sur l'estimation précise de la *slack* pour éviter les erreurs. La mesure de la *slack* permet de contrôler finement la fréquence (ou la tension) d'un circuit en réponse à ses variations (Joshua M. Levine, Edward Stott et P. Y. Cheung 2014) afin de minimiser la consommation d'énergie du FPGA (J. M. Levine et al. 2012).

Cette *slack* est mesurée via un double échantillonnage. Le principe est similaire à celui de la bascule Razor, mais le déphasage de la seconde horloge est contrôlable. Celle-ci est

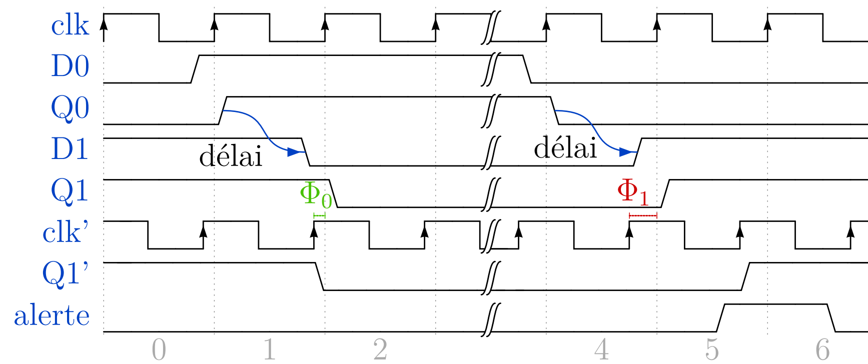


FIGURE 2.9 – Estimation de la *slack* pour le circuit de la figure 2.4 où la seconde bascule est remplacée le mécanisme *Online Slack Measurement*. La phase de l'horloge secondaire `clk'` effectue un balayage qui permet d'inférer la *slack*. Dans la première partie (cycles 0 à 2), la phase Φ_0 est suffisamment faible et les deux bascules échantillonnent la bonne valeur. Dans la seconde partie (cycles 4 à 6), la phase Φ_1 est trop importante et la bascule secondaire n'échantillonne pas la nouvelle valeur de `D1`. Les signaux d'alerte sont utilisés en correspondance avec le balayage de la phase pour estimer la *slack*. Les signaux du circuit principal (`D0`–`Q0`–`D1`–`Q1`) n'ont en principe pas d'erreur temporelle.

configurée en *avance* de phase : la bascule déphasée `FF'` échantillonne le signal *avant* la bascule principale `FF`. La phase est modifiée à la volée pour effectuer un balayage. Les sorties des deux bascules sont comparées avec une porte OU et le résultat est échantillonné en opposition de phase (sur le front descendant de l'horloge principale). Cela permet, sans modifier la fréquence de fonctionnement du circuit, d'estimer précisément la *slack* en observant la différence entre les deux bascules (J. M. LEVINE et al. 2013).

En ajoutant une marge d'erreur pour que les bascules principales n'échantillonnent jamais des signaux erronés, le système est supposé ne jamais avoir d'erreur et n'a pas besoin de mécanisme de correction. La *slack* doit cependant être mesurée à une fréquence suffisamment élevée pour que le système puisse réagir correctement à toute modification de celle-ci à causes des différentes sources de variabilité.

Une limite de cette approche est que le balayage de la phase pour estimer la *slack* est effectué en ligne, pendant le calcul sur les données réelles. Or, les données influencent les délais de propagations des signaux. Par exemple, si les signaux d'entrées d'un circuit combinatoire ne varient pas, alors le délai de propagation ne peut pas être mesuré parce que les signaux de sortie ne varient pas. La *slack* peut apparaître virtuellement infinie dans ce cas. Cette erreur de mesure est inévitable. Certains circuits peuvent donc ne pas être correctement testés et la fréquence ou la tension choisies peuvent les rendre dysfonctionnels.

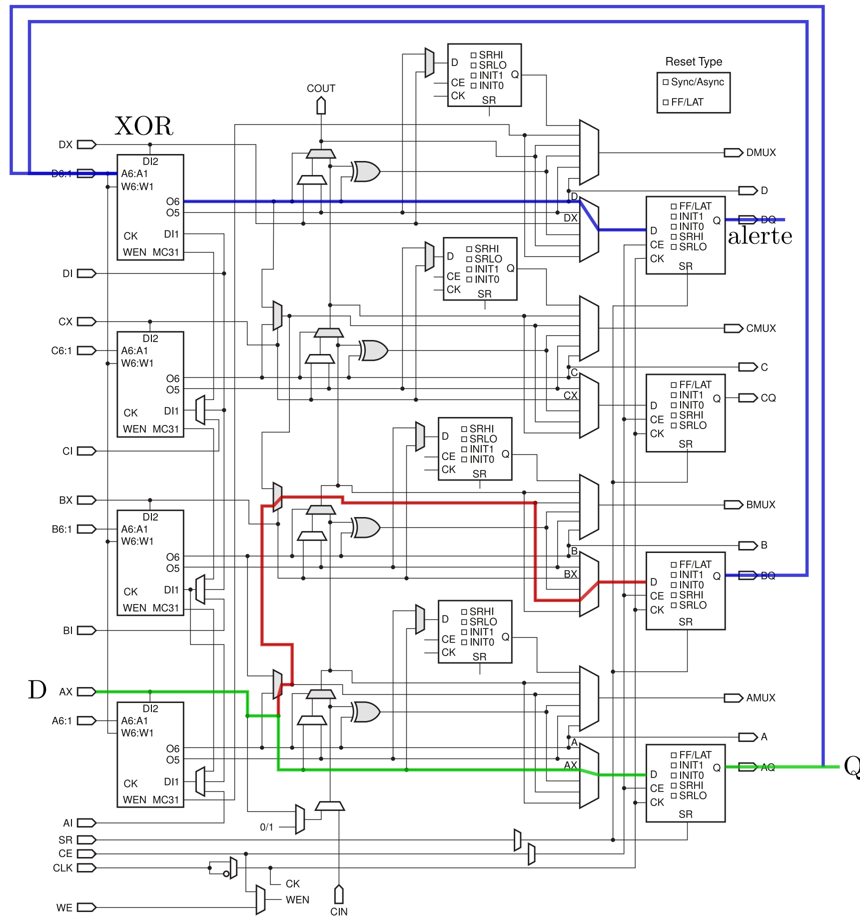


FIGURE 2.10 – Détecteur Elongate (J. L. NUNEZ-YANEZ 2018) basé sur une SLICEM Xilinx *7 Series* (« 7 Series FPGAs Configurable Logic Block User Guide (UG474) » 2016). Le circuit vert correspond au fonctionnement de base de la bascule D. Les circuits rouges et bleus permettent la détection de *slack* critique sur cette bascule D. Le premier retarde le signal d'entrée, le second compare les sorties des deux bascules D.

Elongate Une autre approche ne mesurant pas la *slack* mais permettant de détecter quand celle-ci devient critique (trop faible) a été proposée par Nunez-Yanez : Elongate (J. NUNEZ-YANEZ 2013). Comme pour la mesure de la *slack*, l'information peut être fournie à un système de DVFS afin d'ajuster la fréquence et la tension de manière préventive pour un circuit implémenté sur FPGA. Cette méthode a été utilisée pour améliorer l'efficacité énergétique d'un réseau de neurones binaire (J. L. NUNEZ-YANEZ 2018). Elle a permis d'augmenter les performances de 86,8 % et l'efficacité énergétique de 86,3 %.

Celle-ci ne base pas le double échantillonnage sur deux horloges déphasées, mais sur la mise en place d'un retard sur le signal à protéger. L'approche tire profit de l'architecture

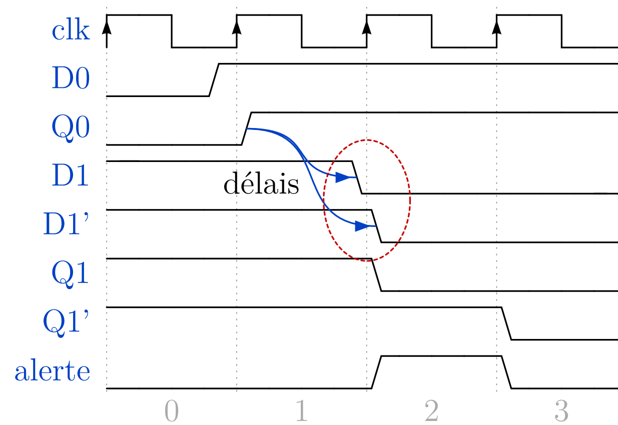


FIGURE 2.11 – Détection de *slack* critique avec Elongate (J. NUNEZ-YANEZ 2013) sur l'exemple de la figure 2.4. Le retard supplémentaire sur l'entrée de la bascule (`D1'`) permet de détecter que la *slack* est devenue critique, mais il n'y a pas encore d'erreur.

des FPGA (voir figure 2.10). Ceux-ci contiennent des *slices* contenant chacun, entre autres, des bascules et des LUT. Les deux bascules `FF` et `FF'` sont placées dans le même *slice*. Alors que le signal est directement routé vers la bascule `FF`, il est routé vers la bascule `FF'` en passant par plusieurs multiplexeurs du *slice*. Cette différence suffit à allonger le délai de propagation du signal. Les deux bascules échantillonnent donc au même moment deux signaux *différents*. Les sorties des deux bascules sont routées vers une LUT du *slice* pour les comparer. La différence de délai des signaux permet de détecter lorsque le signal `D` devient stable trop peu de temps avant le front montant de l'horloge. Dans ce cas, on sait que la *slack* du chemin protégé atteint une limite critique. Cependant, même si la *slack* atteint un seuil critique, il n'y a pas encore d'erreur temporelle et le signal `Q` est correct. La figure 2.11 montre un chronogramme de détection de *slack* critique basée sur ce principe.

Inconvénients de ces méthodes Ces méthodes ont un surcoût important : en plus du coût fixe, il faut, pour chaque bascule protégée, ajouter une autre bascule, une porte XOR et une bascule pour le signal d'erreur. Cela augment aussi la complexité du routage. Ce surcoût serait prohibitif si elles protégeaient tous les registres. Toutes ces approches proposent donc un compromis entre la couverture d'erreur et le surcoût de la détection d'erreur. Plus la couverture est grande, plus le coût est important. Cela amène un nouveau problème : comment choisir les registres protégés ? Il faut bien sûr protéger les registres qui seront impactés par des erreurs temporelles. Les registres les plus sensibles aux erreurs temporelles sont ceux qui sont connectés aux chemins les plus critiques. Elles font

l'hypothèse que l'ordre des chemins selon leur criticité ne varie pas à causes des différentes sources de variabilité : les chemins les plus critiques restent les plus critiques (J. M. Levine et al. 2013). Dans ce cas, protéger seulement les registres les plus critiques doit suffire, pour un coût moindre. Cependant, si l'hypothèse n'est pas déraisonnable pour les variabilités dans le temps (température, données), la variabilité concerne aussi les différences entre les circuits physiques (fabrication, vieillissement). Un chemin non protégé peut donc, en pratique, être critique et conduire à une erreur, alors que les protections en place, sélectionnées selon un modèle du comportement temporel du FPGA, ne détectent pas d'erreur. De plus, si les chemins sont bien équilibrés, alors il peut ne pas y avoir beaucoup de différence entre les chemins les plus critiques et les autres, ce qui amplifie ce risque. Le fonctionnement correct de ces circuits n'est donc pas strictement garanti.

De plus, les délais de propagations des chemins sont dérivés du placement et du routage du circuit *avant* l'injection des mécanismes de protection (que ce soit pour un ASIC ou un FPGA). L'injection des mécanismes de protection a un impact sur le comportement temporel du circuit, rendant la sélection finale des registres encore plus complexe.

Levine et al. proposent de protéger les registres ayant une *slack* proche de n% de la *slack* la plus courte (J. M. Levine et al. 2012). Pour n à 5 %, 10 %, 15 % et 20 %, ils mesurent un surcoût moyen en nombre d'éléments logiques respectivement de 0,95 %, 2,7 %, 4,7 % et 6,2 %. Nunez-Yanez présente une approche permettant de maximiser la protection tout en restant sous la barre des 5 % de surcoût (J. L. Nunez-Yanez 2018).

Pour que les approches fonctionnent, les délais combinatoires entre le chemin de données et les deux bascules doivent être contrôlés. En effet, elles ne fonctionnent pas si le délai pour la bascule déphasée est plus court que le délai de la bascule principale. En revanche, si le délai pour la bascule principale est plus court que le délai de la bascule déphasée, alors les approches fonctionnent mais sont moins efficaces. La priorité lors du placement et du routage peut être donnée au délai de la bascule principale pour que celle-ci ait un délai plus court que la bascule déphasée et garantir le bon fonctionnement. Le routage vers les bascules déphasées peut alors avoir un délai plus élevé, ce qui induit une différence entre la *slack* observée et la *slack* réelle pour les bascules principales. Ce délai supplémentaire peut être ignoré (mais cela réduit l'intérêt du mécanisme), ou un processus de calibration peut être utilisé pour l'évaluer (J. M. Levine et al. 2012).

Dans l'approche de Nunez-Yanez, la différence de délai entre les deux signaux est très courte : c'est le temps de propagation de quelques multiplexeurs. À cause de la variabilité dans la fabrication des circuits, ce délai ne peut pas être exactement connu. Dans le pire

cas, l'ordre des deux délais (bascule principale et secondaire) pourrait s'inverser. Dans ce cas, la *slack* critique peut toujours être détectée mais la valeur de la bascule principale est fausse. Le circuit calcul donc un résultat faux qui n'est pas corrigé. À moins de vérifier le fonctionnement de toutes les bascules protégées, cette approche ne donne aucun moyen de détecter ce cas de figure.

2.2.4.5 Tolérance aux fautes logicielle

Des approches pour tolérer les fautes sont adaptées à un contexte logiciel : l'exécution d'un programme sur un processeur généraliste. Par exemple, Reis et al. proposent une technique combinant la redondance des instructions avec la vérification de l'intégrité de flot de contrôle (*Control-Flow Integrity*, CFI) nommée SWIFT (Reis et al. 2005).

La redondance des instructions est réalisée par une transformation de compilateur qui duplique les instructions d'un programme et ajoute des contrôles aux points clés (instructions *store* et branchements). Les points de contrôle vérifient que les valeurs redondantes sont cohérentes. Dans le cas contraire, une erreur a été détectée. Lorsque les instructions sont ordonnancées par le compilateur (ou par l'ordonnanceur dynamique matériel), celui-ci essaye d'utiliser au maximum les ressources disponibles du processeur en maximisant l'IPC (voir section 1.2.1). Ainsi, le surcoût en temps de cette technique est en pratique inférieur à 100 % parce que les instructions supplémentaires peuvent être ordonnancées dans les « trous » de l'ordonnancement du programme d'origine.

Si la mémoire du processeur utilise déjà un contrôle des erreurs (voir section 2.2.4.2), alors les données en mémoire n'ont pas besoin d'être dupliquées. Cela permet de réduire l'impact de la méthode sur la quantité de mémoire utilisée et donc sur les performances des caches. La figure 2.12 montre un exemple d'une telle transformation.

La vérification de l'intégrité du flot de contrôle sert à s'assurer que les instructions de contrôle de flot (sauts et branchements) déplacent bien le compteur de programme à l'instruction voulue. La technique proposée est basée sur les travaux de Oh et al. qui permet de vérifier que les sauts pris sont valides, c'est-à-dire font partie du *graphe de flot de contrôle* du programme (Nahmsuk Oh, Shirvani et Edward J. McCluskey 2002). Cette technique, conjointement à la vérification de l'adresse du saut (voir ci-dessus), permet la vérification de l'intégrité du flot de contrôle.

Le compilateur assigne une constante unique s appelée signature à chaque bloc de

```
lw      a0, -12(s0)      ;   charge a
lw      t0, -12(s0)      ; * charge a
lw      a1, -16(s0)      ;   charge b
lw      t1, -16(s0)      ; * charge b
add     t2, a0, a1       ; * a + b
add     a0, a0, a1       ;   a + b
bne     a0, t2, .erreur  ; * comparaison
sw      a2, -20(s0)      ;   stocke
```

```
int add(int a, int b) {
  return a + b;
}
```

FIGURE 2.12 – Programme assembleur (RISC-V) effectuant une simple addition, équivalent au programme C à droite. La marque * indique les instructions ajoutées pour la détection d'erreur : les chargements depuis la mémoire, l'addition, ainsi que la comparaison des deux résultats avant le *store*. En cas d'erreur, le programme saute vers une routine d'erreur.

base [2]. Un registre G est utilisé pour contenir la signature du bloc courant. Lors d'un branchement, la valeur de ce registre est mise à jour avec le résultat du ou-exclusif de sa valeur courante (la signature du bloc de base source du branchement) et une constante de différence d (qui est le ou-exclusif entre les signatures du bloc source et du bloc destination : $d = s_{source} \oplus s_{dest}$). Lorsqu'il y a plusieurs sources possibles, le registre G est aussi mis à jour avec le ou-exclusif de sa valeur courante et d'une constante d'ajustement placée dans un second registre D avant le branchement. Cette étape supplémentaire permet d'obtenir la bonne signature pour plusieurs blocs sources valides, sans que tous les blocs sources aient la même signature. La valeur du registre est ensuite comparée à la signature du bloc courant et le programme branche vers une routine de gestion d'erreur s'il y a une différence. Le compilateur précalcule ces constantes (constantes de différence et constantes d'ajustement) afin que le résultat des opérations de ou-exclusif soit égal à la signature du bloc courant, mais uniquement lorsqu'un branchement valide a été effectué. La figure 2.13 montre un exemple de graphe de flot avec cette protection.

Cette technique protège des branchements invalides : si une adresse de branchement est erronée ; si une instruction de branchement n'est pas exécutée ou si une instruction est exécutée à tort comme une instruction de branchement. Dans certains cas, quand un saut invalide a lieu vers une adresse autre que le début d'un bloc de base, l'erreur est finalement détectée au branchement suivant parce que la valeur de G n'aura pas été correctement mise à jour.

Cette technique nécessite de bloquer un registre sur toute l'exécution du programme,

2. Un bloc de base est la plus grande suite d'instructions forcément séquentielles, c'est-à-dire sans instruction de contrôle de flot (sauf possiblement la dernière instruction) et dont les instructions ne sont pas la cible de sauts et branchements (sauf la première instruction).

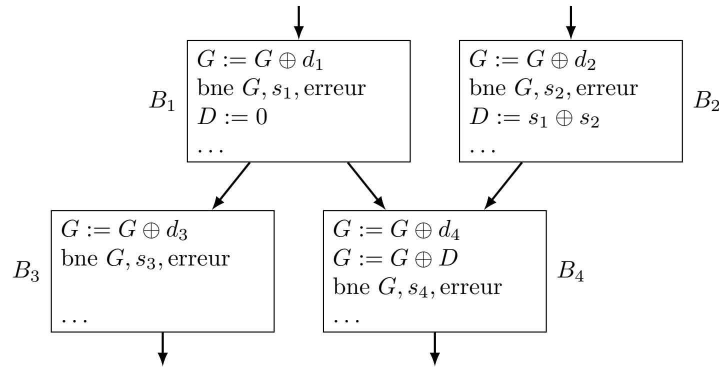


FIGURE 2.13 – Graphe de flot de contrôle montrant la vérification de l'intégrité du flot de contrôle. Chaque nœud représente un bloc de base et les arêtes les branchements valides. Le pseudo-code montre la vérification des signatures des blocs. Les noms en minuscule (s et d) représentent des constantes précalculées par le compilateur, tandis que les noms en majuscule (G et D) représentent des registres. Par exemple, la constante d_3 vaut $s_1 \oplus s_3$. Quand le contrôle passe du bloc B_1 au bloc B_3, le registre G prend pour valeur $G \oplus d_3 = s_1 \oplus (s_1 \oplus s_3) = s_3$. Le branchement est alors valide. En revanche, s'il y avait un branchement invalide entre B_2 et B_3, le registre G aurait pour valeur $s_2 \oplus d_3 = s_2 \oplus s_1 \oplus s_3$, ce qui permet, avec des constantes appropriées, de détecter l'erreur.

un second registre lors d'un branchement à sources multiples et n'a besoin que de peu d'instructions par branchement (deux ou trois ou-exclusifs, une ou deux affectations et un branchement conditionnel). De plus, elle ne nécessite pas de mémoire : les différentes constantes sont placées directement dans les instructions.

Ces techniques ont l'avantage de ne pas nécessiter de matériel spécialisé. Cependant, elles ne protègent pas de tout type d'erreur (par exemple, une instruction exécutée à tort comme une instruction *store*) et ont un surcoût, bien qu'inférieur à 100 %, important. Reis et al. rendent compte d'un surcoût en temps d'exécution de 41 % en moyenne géométrique sur une suite de *benchmarks*. Le surcoût en nombre d'instructions est de 123 % mais le niveau d'IPC effectif est amélioré par un facteur de 48 %.

Cependant, si ces techniques peuvent être pertinentes dans un contexte logiciel, elles le sont bien moins dans un contexte matériel. En effet, un accélérateur matériel n'utilise généralement pas d'instructions, d'ordonnancement dynamique ou encore de flot de contrôle complexe. Dupliquer les calculs est plus simple grâce à la souplesse de ce type d'architecture. Le contrôle est généralement effectué par un composant centralisé (plutôt que par des instructions dispersées) qui peut aussi être dupliqué ou protégé d'une autre manière.

2.2.4.6 Tolérance aux fautes au niveau algorithmique

La tolérance aux fautes au niveau algorithmique (*Algorithm Based Fault Tolerance*, ABFT) est une approche permettant de détecter (et parfois corriger) les erreurs en examinant une propriété de l'algorithme à protéger. L'ABFT se base généralement sur la vérification d'un *invariant* de l'algorithme, c'est-à-dire une propriété qui reste vraie lorsque l'algorithme est exécuté. Lorsque l'invariant n'est pas vérifié, on sait qu'une erreur est survenue. Cette approche ne s'appuie donc pas sur les détails de l'implémentation (instructions, mémoires, circuits, fréquences, etc.) mais uniquement sur l'algorithme protégé. Une méthode permettant de protéger un algorithme est donc efficace quels que soient les détails de son implémentation. Cette approche est robuste, a une très bonne couverture d'erreur et peut utiliser très peu de ressources. En revanche, une méthode est spécifique à un algorithme et n'est pas généralisable.

Le terme ABFT a été proposé par Huang et Abraham pour des techniques de détection d'erreur pour des opérations matricielles en 1984 (Huang et Abraham 1984) : multiplication, addition, produit scalaire, décompositions LU, transposition. D'autres méthodes ont été proposées pour la factorisation QR (Reddy et Banerjee 1990) ; les méthodes itératives pour résoudre les équations aux dérivées partielles (Roy-Chowdhury, Bellas et Banerjee 1996) ; etc.

Pour illustrer le principe de l'ABFT, nous présentons ici son application à la protection d'un produit de matrice par Huang et Abraham. Considérons deux matrices A de taille $m \times n$ et B de taille $n \times p$. La matrice $C = A \times B$ de taille $m \times p$ est le produit de A et B et est définie par :

$$C_{i,j} = \sum_{k=0}^{n-1} A_{i,k} B_{k,j}$$

Pour détecter les erreurs dans ce calcul, nous pouvons calculer deux valeurs qui devraient être égales (l'invariant). L'une de ses valeurs, que nous appelons somme de contrôle d'entrée, est calculée sur les opérandes de l'opération, tandis que l'autre, la somme de contrôle de sortie, est calculée sur la sortie. La somme de contrôle de sortie σ est simplement la

réduction en somme de la matrice résultat :

$$\sigma = \sum_{i=0}^{m-1} \sum_{j=0}^{p-1} C_{i,j}$$

En substituant la définition de $C_{i,j}$ dans cette équation et en simplifiant l'équation, nous trouvons la formulation de la somme de contrôle d'entrée ρ à partir des opérandes :

$$\rho = \sum_{i=0}^{m-1} \sum_{j=0}^{p-1} \left(\sum_{k=0}^{n-1} A_{i,k} B_{k,j} \right)$$
$$\rho = \sum_{k=0}^{n-1} \left(\sum_{i=0}^{m-1} A_{i,k} \sum_{j=0}^{p-1} B_{k,j} \right)$$

Ce calcul peut être effectué via une multiplication de matrice classique où les opérandes comportent des redondances :

— une ligne est ajoutée à l'opérande gauche A. Chaque valeur de cette ligne est la somme de sa colonne correspondante ;
— une colonne est ajoutée à l'opérande droit B. Chaque valeur de cette colonne est la somme de sa ligne correspondante.

Plus formellement, les deux matrices opérandes sont remplacées par les matrices de taille $(m+1) \times n$ et $n \times (p+1)$ suivantes :

$$A'_{i,j} = \begin{cases} A_{i,j} & \text{si } i < m \\ \sum_{s=0}^{m-1} A_{s,j} & \text{si } i = m \end{cases} \qquad B'_{i,j} = \begin{cases} B_{i,j} & \text{si } j < p \\ \sum_{s=0}^{p-1} B_{i,s} & \text{si } j = p \end{cases}$$

Le résultat C' de taille $(n+1) \times (p+1)$ contient la matrice résultat originale C ainsi que la somme de contrôle d'entrée dans le coin inférieur droit. La figure 2.14 donne une représentation graphique et un exemple numérique de la méthode.

Le calcul de la somme de contrôle de sortie nécessite seulement $mp - 1$ sommes, celui de la somme de contrôle d'entrée nécessite $n(m+p-1) - 1$ sommes et n produits. Cette méthode nécessite donc moins de calculs que, par exemple, la duplication du calcul.

Sous l'hypothèse qu'une seule erreur peut avoir lieu, cette méthode permet aussi de corriger une erreur. Le résultat C' contient en effet la somme de contrôle de sortie mais aussi deux vecteurs supplémentaires (l'un vertical et l'autre horizontal). Ces vecteurs sont

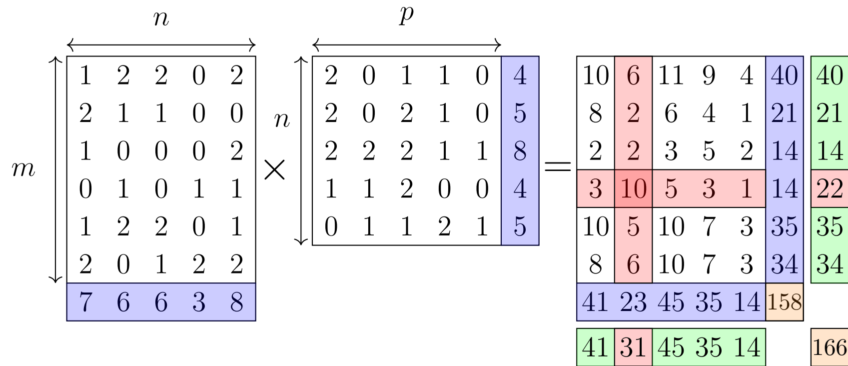


Figure 2.14 – Exemple numérique de la méthode d'ABFT pour multiplication de matrice. Les vecteurs bleus sont les vecteurs sommes additionnels sur les matrices opérandes et dans le résultat. Les vecteurs verts sont les vecteurs sommes sur la matrice résultat. Les valeurs oranges sont les deux sommes de contrôle : 158 est la somme de contrôle d'entrée et 166 est la somme de contrôle de sortie (la somme de l'un des vecteurs verts). L'occurrence d'une erreur est détectée parce que les sommes de contrôle sont différentes. La différence entre les vecteurs bleus et verts permet de localiser l'erreur et de la corriger (en rouge), sous l'hypothèse qu'une seule erreur a eu lieu. L'erreur numérique est de $23 - 31 = 14 - 22 = -8$, la bonne valeur n'est donc pas 10 mais $10 - 8 = 2$.

calculés par produit matriciel des vecteurs sommes d'un opérande par l'autre opérande. Ils doivent donc correspondre à la somme des lignes et des colonnes de la matrice C. On peut donc détecter l'occurrence d'une erreur mais aussi sa position en calculant ces vecteurs sommes et en les comparant : la position de l'erreur sur chaque paire de vecteurs sommes donne la ligne et la colonne où se trouve l'erreur. L'erreur numérique est égale à la différence des valeurs du vecteur et la valeur erronée de C peut donc être corrigée en y additionnant cette différence (voir la figure 2.14 pour un exemple numérique).

Cette méthode montre un compromis entre le surcoût et la capacité de correction d'erreur. En effet, pouvoir corriger une erreur nécessite de calculer les quatre vecteurs sommes afin de la localiser et de connaître l'erreur numérique. Le calcul des deux vecteurs sommes sur la sortie nécessite $2mp - m - p$ sommes alors que le calcul de la somme de contrôle de sortie seule nécessite $mp - 1$ sommes. De la même manière, le calcul des deux vecteurs sommes correspondant via les opérandes nécessite deux produits vecteur-matrice, alors que le calcul de la seule somme de contrôle d'entrée nécessite un produit vecteur-vecteur. Enfin, la comparaison des vecteurs nécessite n opérations de comparaison. Dans certains cas, il peut donc être plus intéressant de ne pas avoir la capacité de corriger les erreurs et de recalculer le résultat lorsqu'une erreur et détectée (Al-Yamani, N. Oh et

E. J. McCluskey 2001).

Ces méthodes ont cependant plusieurs limites. Leur spécificité à un algorithme permet d'obtenir des mécanismes de détection d'erreur ayant un faible surcoût mais ne pouvant pas être généralisés.

La phase de vérification de l'absence d'erreur survient après le calcul, donc le délai entre l'occurrence d'une erreur et sa détection peut être long. Lorsqu'une erreur survient, du temps et de l'énergie ont été dépensés à réaliser des calculs inutiles jusqu'à ce que celle-ci soit détectée. Parfois, comme pour la multiplication de matrice, il est possible de découper un algorithme en multiples étapes (réparties temporellement ou spatialement). Le grain de la phase de vérification (sur peu ou beaucoup de données) offre alors un compromis entre la latence de détection d'erreur et son surcoût (Z. Chen 2013).

Enfin, ces méthodes ne fonctionnent pas directement avec les arithmétiques à virgule flottante. En effet, à cause des erreurs d'arrondi, les opérations de ces arithmétiques ne respectent pas les règles usuelles comme la distributivité ou l'associativité. Par exemple, considérons ce produit de matrice :

$$\begin{bmatrix} 2^{63} & 2^{63} \\ 2^{-63} & 2^{-63} \end{bmatrix} \times \begin{bmatrix} 2^{63} & 2^{-63} \\ 2^{63} & 2^{-63} \end{bmatrix} = \begin{bmatrix} 2^{127} & 2^{1} \\ 2^{1} & 2^{-125} \end{bmatrix}$$

Le calcul de la somme de contrôle d'entrée est :

$$\begin{bmatrix} 2^{63} + 2^{-63} & 2^{63} + 2^{-63} \end{bmatrix} \times \begin{bmatrix} 2^{63} + 2^{-63} \\ 2^{63} + 2^{-63} \end{bmatrix} = 2^{127} + 2^{2} + 2^{-125}$$

La somme de contrôle nécessite ici 21 bits de précision pour être représentée exactement. Si le format choisi a une précision inférieure, le résultat ne sera pas exact. Par exemple, le résultat sera au format *binary32* (simple précision) 2^{127} (arrondi au plus près). Il est donc possible de construire une matrice résultat erronée dont la somme de contrôle de sortie sera égale, représentée dans le format choisi, à la somme de contrôle d'entrée, par exemple :

$$\begin{bmatrix} 2^{127} & 2^{1} \\ 2^{1} & 2^{103} \end{bmatrix}$$

Dans ce cas, la somme de contrôle de sortie est égale à $2^{127} + 2^{103} + 2^{2}$. Au format simple précision, le résultat sera aussi 2^{127} et l'erreur ne sera pas détectée, malgré une énorme erreur relative (4×10^{70} %).

Approche	Pertinence
codes correcteurs d'erreurs	protège les communications, pas les calculs
arithmétique modulaire	coût élevé
double échantillonnage	coût élevé et non applicable aux FPGA
tolérance aux fautes logicielle	non applicable aux FPGA
tolérance aux fautes au niveau algo.	coût faible, couverture importante

TABLE 2.1 – Comparaison de différentes approches de tolérance aux fautes pour la spéculation temporelle sur FPGA

Plusieurs approches ont cependant été proposées pour rendre ces méthodes utilisables avec les opérations à virgules flottantes. Il est possible de simplement choisir un seuil d'erreur acceptable pour l'application, ou un seuil observé expérimentalement. Cependant, un seuil d'erreur trop faible provoquera des faux positifs, tandis qu'un seuil d'erreur trop élevé provoquera des faux négatifs. Roy-Chowdhury et al. ainsi que Braun et al. proposent de calculer une borne supérieure de l'erreur d'arrondi à la volée (ROY-CHOWDHURY, BELLAS et BANERJEE 1996 ; BRAUN, HALDER et WUNDERLICH 2014).

Des méthodes d'ABFT ont été utilisées sur différentes plateformes modernes : FPGA (JACOBS, CIESLEWSKI et GEORGE 2012), GPU (BRAUN, HALDER et WUNDERLICH 2014 ; HARI et al. 2020), superordinateurs (X. LIANG et al. 2017), etc.

2.2.4.7 Synthèse

Parmi les différentes techniques de tolérance aux fautes présentées, seules certaines sont applicables au cas de la spéculation temporelle pour FPGA. La table 2.1 résume la pertinence de chaque approche présentée dans ce chapitre.

Les codes correcteurs d'erreur peuvent être utilisés pour protéger des communications et des mémoires. Ils ne sont pas utilisables pour protéger des calculs et ne conviennent donc pas à notre besoin.

La duplication ou la triplication (temporelle ou spatiale) induisent un surcoût important de 100 % en temps ou en matériel, mais aussi en énergie. Les gains espérés grâce à la spéculation temporelle étant inférieur à 100 %, utiliser la duplication réduirait les performances et l'efficacité énergétique.

La tolérance aux fautes logicielle s'applique à une architecture exécutant un programme. Cette technique est particulièrement pertinente pour les processeurs à microar-

chitectures superscalaire ou vectorielle, mais ne s'applique pas aux architectures matérielles comme les FPGA.

La redondance basée sur l'arithmétique modulaire peut en revanche être utilisée pour la détection d'erreur temporelles. Le choix de la base offre un compromis entre le coût de la détection d'erreur et sa couverture : une base contenant plus de diviseurs ou des diviseurs plus grands permet une meilleure couverte d'erreur, mais le coût des opérateurs sera plus élevé. De plus, on peut s'attendre à ce que les opérateurs sur les différents restes aient un chemin critique plus court que les opérateurs principaux. Aussi, ce mécanisme de détection d'erreur s'avère très pertinent dans le cadre de la spéculation temporelle où le temps de propagation des signaux influe sur le risque d'erreur. En effet, lorsque la fréquence d'horloge devient trop élevée pour que le circuit fonctionne correctement, les opérateurs principaux seront impactés tandis que les opérateurs redondants fonctionneront correctement. Cette propriété permettrait de ne pas avoir de faux positifs ni de faux négatifs liés aux erreurs temporelles impactant le circuit de détection d'erreur. Cependant, ce n'est pas une certitude à cause du routage complexe des outils modernes de synthèse et de l'équilibrage des différents chemins.

Un inconvénient de cette solution est qu'elle protège uniquement des erreurs temporelles se produisant dans les opérateurs du chemin de données principal. Elle ne protège pas d'erreurs temporelles survenant ailleurs, comme sur les circuits de contrôle ou la mémoire. Elle doit donc être utilisée conjointement à d'autres mécanismes pour détecter toutes les erreurs.

Si cette approche est pertinente, elle est en revanche très chère. Par exemple, pour une détection d'erreur basée sur les restes de la division par 3 combinée à la duplication des circuits non arithmétiques, le surcoût est de 25% (Campbell et al. 2015). Avec un mécanisme d'injection d'erreur simpliste, les auteurs montrent que leur approche protège de plus de 99% des erreurs. Si l'on prend en compte des erreurs plus complexes, par exemple des inversions de plusieurs bits, alors cette approche protège seulement deux tiers des erreurs des circuits arithmétiques.

Les approches basées sur le principe du double échantillonnage peuvent sembler pertinentes puisqu'elles s'appliquent à des cibles matérielles et sont spécifiquement conçues pour l'*overclocking* ou l'*undervolting*. Elles ont cependant un surcoût important pour une couverture d'erreur limitée à une partie des chemins. De plus, si le principe paraît simple, il est en réalité difficile à implémenter. Il nécessite en effet de bien connaître le comportement temporel de la cible (ce qui n'est pas le cas à cause de la variabilité) et d'implémenter

la solution à très bas niveau. Un autre problème est que certains composants des FPGA contiennent des registres internes qui ne peuvent pas être protégés par ces méthodes, dont les DSP et les blocs RAM. Les registres internes de ces composants, utilisés pour intégrer le composant à un *pipeline* ou pour toute autre fonction, ne sont pas accessibles à l'extérieur de composant ce qui interdit l'accès à leurs signaux d'entrée et de sortie. De plus, ces composants sont souvent présentés comme des boîtes noires dont on ne connaît pas le fonctionnement interne. Une solution est de se passer de ces composants et de n'utiliser que les *slices* pouvant être protégées. Cependant, ces blocs spécialisés sont fournis parce qu'ils permettent d'implémenter des fonctions qui seraient coûteuses implémentées à base de *slices*, comme les multiplications, les registres à décalage, les files d'attente (FIFO), les banques mémoires ou les *barrel shifter*. De plus, changer de composant pour implémenter une fonction a une influence sur la fréquence maximale de fonctionnement du circuit (J. M. Levine et al. 2013). Dans le cas où l'implémentation à base de *slices* est plus lente (par exemple, les multiplications), cela diminue la fréquence maximale du circuit alors que ces méthodes étaient utilisées pour l'augmenter.

La principale limite de la tolérance aux fautes au niveau algorithmique est la spécificité des méthodes utilisées. Si une méthode existe pour l'algorithme utilisé, cette technique est donc très pertinente pour protéger un accélérateur matériel des erreurs temporelles. La mise en œuvre est relativement simple : il suffit de réaliser les calculs supplémentaires parallèlement à l'exécution de l'algorithme principal, sur les mêmes données. Ces calculs supplémentaires nécessitent moins de ressources que le cœur de l'accélérateur et n'exigent donc pas d'être autant optimisés. De plus, ces méthodes permettent d'adapter facilement la détection d'erreur pour plusieurs variantes de l'accélérateur (niveau de parallélisme, format de données, plateforme, etc.) ou plusieurs générations.

Ces méthodes protègent seulement la partie algorithmique de l'accélérateur. Elles peuvent donc être utilisées conjointement avec des protections sur les mémoires ou les communications pour protéger l'ensemble du système. De plus, tous les composants non impliqués dans le calcul de l'algorithme principal (calculs mais aussi contrôle) mais pour d'autres tâches peuvent être protégés par des techniques plus classiques comme la duplication. Le surcoût relatif reste assez faible, car ceux-ci représentent généralement une petite partie des ressources utilisées : la majorité des ressources est allouée au calcul de l'algorithme.

Ces méthodes sont particulièrement adaptées aux accélérateurs matériels. En effet, ceux-ci permettent facilement de gérer plusieurs calculs en parallèle. Ainsi, les calculs

dédiés à la détection d'erreur peuvent être réalisés en parallèle de l'algorithme principal sans influencer les performances de ce dernier. Sur un processeur ou un GPU, l'impact sur les performances de l'algorithme principal ne serait pas nul, même s'il peut être faible. De plus, les accélérateurs matériels permettent d'utiliser des opérateurs de taille arbitraire (par exemple, l'addition d'un mot 7 bits avec un mot 13 bits) ainsi que des registres de taille arbitraire (par exemple, 14 bits pour le résultat de l'opération précédente). Cette possibilité, inexistante dans les architectures généralistes, permet de limiter le surcoût de la détection d'erreur en utilisant le minimum de ressources nécessaires.

2.3 Conclusion

Nous avons montré dans ce chapitre que la spéculation temporelle permettait d'améliorer les performances et l'efficacité énergétique d'un circuit, mais qu'elle devait impérativement être utilisée conjointement à un mécanisme de détection d'erreur. En effet, le risque d'erreur est important et les différentes sources de variabilité rendent impossible la sélection hors ligne d'un point de fonctionnement sûr.

Nous avons présenté quelques techniques de détection d'erreur, montré si elles sont pertinentes ou non pour détecter les erreurs temporelles sur FPGA et comparé leur couverture et leur coût. Parmi ces techniques, nous avons conclu que la tolérance aux fautes au niveau algorithmique, lorsqu'elle est applicable, est la technique ayant la meilleure couverture d'erreur pour le coût le plus faible.

Chapitre 3

Réseaux de neurones convolutifs et FPGA

Dans les chapitres précédents, nous avons montré que les architectures dédiées permettent d'obtenir de meilleures performances et une meilleure efficacité énergétique pour certaines applications. Nous avons ensuite présenté la spéculation temporelle et montré que les FPGA sont une plateforme pertinente pour celle-ci. En effet, ils permettent un contrôle fin de la fréquence et d'implémenter efficacement différents mécanismes de détection d'erreur. Nous avons pu observer empiriquement que la fréquence à laquelle un circuit fonctionne sans erreur est bien au-delà de la fréquence nominale. Cette marge de progression importante permet des gains significatifs en performance et en efficacité énergétique. La spéculation temporelle est donc une piste pour améliorer les performances et l'efficacité énergétique d'architectures dédiées sur FPGA.

Ce chapitre présente l'application sur laquelle nous nous basons pour étudier l'impact de la spéculation temporelle sur une architecture dédiée sur FPGA : les réseaux de neurones convolutifs (*Convolutional Neural Networks*, CNN). Nous commençons par présenter les CNN et différentes optimisations existantes permettant de réduire leur consommation d'énergie ou d'améliorer leurs performances. Nous exposons ensuite les principes de mise en œuvre de CNN sur FPGA. Nous présentons les travaux existants portant sur la spéculation temporelle et la détection d'erreur pour CNN. Nous exposons quelques travaux portant sur l'impact des erreurs sur la précision des CNN, puis présentons des mécanismes existants de détection d'erreur spécifiques aux réseaux de neurones.

3.1 Réseaux de neurones convolutifs

Dans le domain de l'intelligence artificielle, les réseaux de neurones convolutifs (*Convolutional Neural Networks*, CNN) sont utilisés pour des tâches complexes comme la classification d'images (Krizhevsky, Sutskever et Geoffrey E Hinton 2012), la recon-

naissance faciale (Lawrence et al. 1997), la reconnaissance de caractères (Lecun et al. 1998), etc. Ce type de réseau de neurones est composé d'une série de couches (ou *filtres*) conçus pour traiter des signaux *naturels* représentés par des matrices (images, sons, etc.) de manière analogue à un *champ de vision* et en extraire une *sémantique*. La reconnaissance de motif est basée sur l'algorithme de convolution discrète, d'où le nom (Yann LeCun, Bengio et G. Hinton 2015).

Un CNN est utilisé en deux phases : l'apprentissage (*backward pass*) et l'inférence (*forward pass*). La phase d'apprentissage permet de trouver tous les paramètres du modèle (valeurs des filtres) à partir de données annotées (apprentissage supervisé). La phase d'apprentissage est principalement basée sur la méthode de rétropropagation du gradient (Y. LeCun et al. 1989) : les paramètres du modèle sont itérativement modifiés pour minimiser l'erreur entre le résultat obtenu et le résultat attendu. La phase d'inférence utilise le modèle ainsi obtenu pour obtenir le résultat sur une nouvelle donnée.

Les modèles de réseaux sont donc typiquement entrainés une seule fois pour un nombre indéterminé d'inférences. Les efforts d'optimisations et la littérature se concentrent donc principalement sur la phase d'inférence puisque celle-ci nécessite au total plus de calculs et d'énergie, bien que la phase d'apprentissage puisse nécessiter des ressources importantes. L'optimisation des réseaux eux-mêmes (comme leur nombre de paramètres) et de leur implémentation bénéficie d'une large attention dans la littérature. Les CNN sont de plus en plus utilisés dans l'industrie et nécessitent beaucoup de calculs, de mémoire et d'énergie. Certains modèles de CNN peuvent nécessiter 40 milliards d'opérations pour une seule inférence (Canziani, Paszke et Culurciello 2016). Les optimisations utilisées peuvent donc avoir un impact important sur les coûts d'exploitations de ces techniques.

Dans le reste de cette section, nous présentons plus en détail l'architecture et les couches typiques d'un CNN ; nous expliquons les optimisations existantes sur la phase d'inférence ; nous traitons des représentations de données utilisées et nous présentons les techniques d'implémentations existantes. Cette section se base en partie sur des articles de synthèse existants par Abdelouahab et al. (ABDELOUAHAB et al. 2018) et par Lecun et al. (Yann LeCun, Bengio et G. Hinton 2015).

3.1.1 Architecture

Un CNN typique est composé d'une suite de *couches* de différents types. L'entrée du calcul est une matrice (par exemple, une matrice 1D pour un signal audio, une matrice 3D pour une image en couleur, etc.), tout comme le résultat (par exemple, une matrice 1D

pour un vecteur de vraisemblance).

Les premières couches sont composées de couches de convolution (suivies d'une couche non linéaire) et de *pooling*. Elles servent à extraire de l'information organisée sous forme de *carte de fonction* (*feature map*). Ces couches traitent des cartes de fonction (ou l'entrée elle-même) et en extraient une information d'un niveau plus *abstrait* pour donner une carte de fonction sémantiquement plus abstraite. Chaque « image » des cartes de fonction représente le résultat d'un filtre différent appliqué uniformément à l'entrée (par exemple, repérant une ligne horizontale). Cette architecture exploite le fait que les signaux naturels sont hiérarchiques. Ainsi, plusieurs étages de ces couches peuvent être utilisés en cascade pour détecter des éléments de plus en plus abstraits, organisés en hiérarchie. Par exemple, pour la reconnaissance vocale, le signal s'assemble en sons, puis en phonèmes, puis en syllabes, en mots et enfin en phrases (Yann LeCun, Bengio et G. Hinton 2015).

Le reste du CNN se compose de couches plus classiques de réseaux de neurones artificiels comme les perceptrons multicouches. Cette partie sert donc à traiter les cartes de fonction les plus abstraites (les dernières du calcul) afin d'obtenir le résultat final (comme une vraisemblance de classes, une description d'image, etc.).

Les modèles de CNN se distinguent par le nombre et le type de couches, la taille des filtres, leur profondeur, leur largeur (il peut y avoir plusieurs flots de calculs parallèles), etc. Ces choix architecturaux sont en dehors du périmètre de ce travail. La figure 3.1 donne un exemple d'un modèle de CNN.

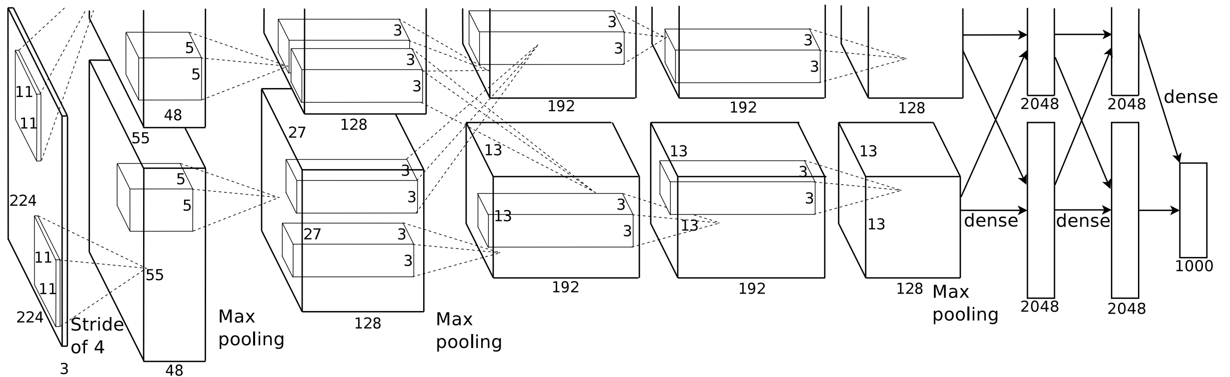


Figure 3.1 – Exemple d'architecture de CNN : AlexNet (Krizhevsky, Sutskever et Geoffrey E Hinton 2012). L'entrée est une image carrée de 224 pixels de côté à 3 couleurs. La première couche de convolution utilise des noyaux de taille 11×11 pour générer deux cartes de fonction de profondeur 48. La sortie est un vecteur de vraisemblance pour 1000 classes.

3.1.2 Types de couches

Cette section présente le fonctionnement, l'utilisation et les contraintes des couches principales des CNN.

Couche de convolution Les couches de convolution sont basées sur le produit de convolution discret [1] (DUMOULIN et VISIN 2016). Elles permettent d'appliquer un ensemble de filtres (noyaux de convolution) à leur entrée. Chacune des M cartes de fonction 2D de sortie est la somme des convolutions 2D de chacune des N cartes de fonction 2D d'entrée avec des noyaux de convolution différents. Autrement dit, $M \times N$ noyaux de convolution uniques sont utilisés, permettant de détecter plusieurs motifs (ou *fonctions*) en fonction des résultats de la couche précédente. Les valeurs de ces filtres, les *poids*, sont le résultat de l'apprentissage. Chaque couche de convolution a différents paramètres liés à l'architecture du CNN :

- N : nombre de cartes de fonction d'entrée ;
- M : nombre de cartes de fonction de sortie ;
- $R \times C$: dimension 2D des cartes de fonction de sortie ;
- S : pas de convolution ;
- K : taille des noyaux ;
- *padding* (remplissage) : type de remplissage pour les bords.

Les dimensions 2D des cartes de fonction d'entrée peuvent être dérivées de ces paramètres. Une couche de convolution (sans *padding*) qui génère une matrice de taille $M \times R \times C$ avec $M \times N$ filtres de taille $K \times K$ traite une matrice d'entrée de taille $N \times R' \times C' = N \times (S(R-1) + K) \times (S(C-1) + K)$.

Le pas de convolution permet d'appliquer le filtre à « gros grain » : la fenêtre glissante du filtre sur l'entrée est effectuée avec ce pas. Un pas supérieur à un (non unitaire) permet de réduire le nombre de calculs nécessaires ainsi que la taille des cartes de fonction de sortie par rapport à celle des cartes de fonction d'entrée (par un rapport S^2). Cela revient à réaliser un sous-échantillonage sur les cartes de fonction de sortie. Il est plutôt utilisé dans les premières couches des réseaux avec un filtre de grande taille.

Il y a plusieurs possibilités pour le *padding* pour les bords : pas de *padding* ; *padding* par des zéros (entièrement ou partiellement) ; *padding* par les valeurs des bords ; etc. Ce choix

1. En réalité, il s'agit d'une corrélation croisée ; les deux fonctions étant équivalentes avec un filtre tourné à 180°, utiliser l'une ou l'autre ne change rien tant que l'apprentissage et l'inférence restent cohérents.

permet d'appliquer, ou non, les filtres sur l'entièreté de l'entrée (y compris les bords). Au-delà de l'impact sur l'application des filtres, ce choix change la taille de la matrice d'entrée. Dans la suite, nous considérerons toujours des convolutions sans *padding* pour simplifier, mais ce choix n'a pas vraiment d'impact sur l'implémentation des convolutions ni sur nos travaux.

En notant x l'entrée de la convolution, y sa sortie et w les poids (ou noyaux), y est défini par :

$$y_{r,c,m} = \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} x_{n,Sr+i,Sc+j} \cdot w_{m,n,i,j}$$

avec :

$$\begin{aligned} 0 \leqslant r < R \\ 0 \leqslant c < C \\ 0 \leqslant m < M \end{aligned}$$

La figure 3.2 montre une représentation graphique d'une couche de convolution reprenant l'analogie du « champ visuel » : l'application du filtre (dont une instance est explicitée) permet de calculer les valeurs des cartes de fonction de sortie, chaque carte de fonction de sortie est la somme de convolutions 2D des cartes de fonction d'entrée.

Couche d'activation Les couches de convolution sont immédiatement suivies d'une couche d'activation. Ces couches servent à introduire une non-linéarité, comme pour les réseaux de neurones classiques imitant les neurones biologiques : elles permettent d'*activer* les *neurones* si la valeur de sortie dépasse, par exemple, un certain seuil. Plusieurs types de fonctions non-linéaires peuvent être utilisées comme la fonction tangente hyperbolique, la fonction sigmoïde (ABDELOUAHAB et al. 2018) ou la fonction ReLU $f(x) = \max(0, x)$ (KRIZHEVSKY, SUTSKEVER et Geoffrey E HINTON 2012).

La fonction non-linéaire est simplement appliquée à chaque valeur de la matrice opérande. Ce type de couches est donc assez simple, ne nécessite pas d'accès mémoire particulier (chaque valeur d'entrée est lue une fois, chaque valeur de sortie est écrite une fois) et ne nécessite pas beaucoup de calculs. La fonction ReLU est beaucoup utilisée pour sa simplicité (son calcul étant très rapide et peu coûteux, ne nécessitant aucune fonction mathématique poussée), aussi bien pour l'apprentissage que l'inférence (KRIZHEVSKY,

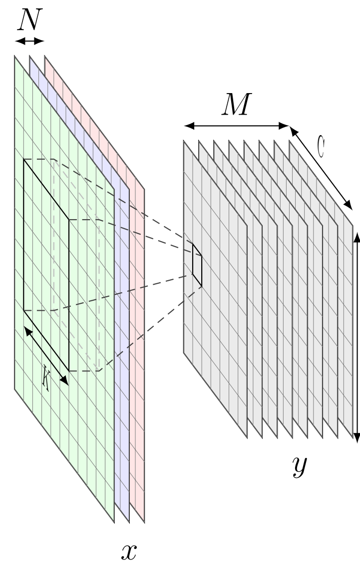


FIGURE 3.2 – Représentation graphique d'une couche de convolution et son « champ de vision ». Ses paramètres sont $R = C = 7$, $K = 5$, $N = 3$, $M = 8$ et $S = 1$.

SUTSKEVER et Geoffrey E HINTON 2012). Dans les implémentations matérielles, cette couche est souvent fusionnée avec les couches de convolution pour limiter les transferts de mémoire.

Couche de *pooling* Les couches de *pooling* permettent de « résumer » l'information contenue dans les cartes de fonction (DUMOULIN et VISIN 2016). La dimension des données est donc réduite (sous-échantillonage). Plusieurs fonctions sont utilisées, comme la moyenne ou le maximum (ABDELOUAHAB et al. 2018).

Chaque carte de fonction est traitée indépendamment et leur nombre ne change pas. Les blocs considérés par le *pooling* peuvent se chevaucher ou non (KRIZHEVSKY, SUTSKEVER et Geoffrey E HINTON 2012).

Par exemple, un choix typique est un bloc de taille 2×2 et un pas de 2 permettant de diviser la taille des données par 4, sans chevauchement. La figure 3.3 montre une représentation graphique d'une couche de *pooling*.

Couche entièrement connectée Ces couches se trouvent typiquement à la fin du réseau et servent à réaliser la classification à partir des prétraitements réalisés par les couches antérieures (donnant les fonctions). Le nom « entièrement connectée » (*fully-connected*) vient du fait que chaque valeur résultat dépend de *toutes* les valeurs du vecteur

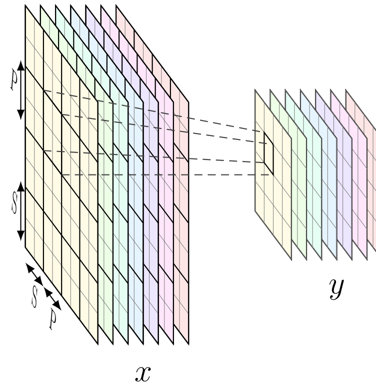


FIGURE 3.3 – Représentation graphique d'une couche de *pooling* avec un bloc de taille $P \times P = 2 \times 2$ et un pas de $S = 2$. L'entrée x a le même nombre de cartes de fonction que la sortie y mais celles-ci sont plus grandes.

d'entrée, permettant la classification, contrairement à une couche de convolution faite pour détecter des fonctions locales. Ces couches sont en réalité une simple multiplication matrice-vecteur, donc une application linéaire. Les coefficients de la matrice sont les poids (calculés lors de la phase d'apprentissage) tandis que le vecteur est l'entrée (si besoin « aplati » à partir d'une matrice, il n'y a plus besoin de conserver les caractéristiques spatiales comme pour les couches de convolution).

Une seule de ces couches permet donc de classifier des données linéairement séparables. Cependant, un réseau permet de résoudre des problèmes non linéaires si on utilise couches entièrement connectées entrelacées avec des couches d'activation (ou couches non linéaires) (RUMELHART, Geoffrey E. HINTON et WILLIAMS 1986).

Les couches entièrement connectées représentent 8 % de la charge de travail nécessaire du modèle AlexNet et moins de 1 % pour d'autres modèles (ABDELOUAHAB et al. 2018) : elles ne représentent pas la majorité des calculs nécessaires. De plus, ce type de calcul peut tirer parti de tout les travaux sur l'optimisation et l'implémentation de l'algèbre linéaire, en particulier la multiplication matrice-vecteur. Cela explique que peu de travaux de la littérature se focalise sur ces couches.

La figure 3.4 montre une représentation graphique d'une couche entièrement connectée.

Autres couches D'autres couches existent (*dropout* (KRIZHEVSKY, SUTSKEVER et Geoffrey E HINTON 2012), normalisation de lot (IOFFE et SZEGEDY 2015), etc.) permettant d'accélérer l'apprentissage ou les qualités des CNN.

Tous ces types de couches ne nécessitent pas le même nombre de calculs. Les couches

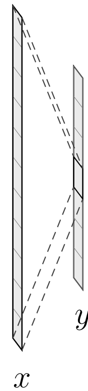


FIGURE 3.4 – Représentation graphique d'une couche entièrement connectée pour 7 classes. Chaque sortie de y dépend de toutes les entrées de x, représentées simplement sous forme de vecteur quelles que soient les dimensions des sorties des couches précédentes.

de convolution, nombreuses et requérant plus de calculs, représentent la plus grosse charge de travail. En comparant les charges de travail en multiplication-accumulation, les couches de convolution représentent pour différents modèles de réseaux (ABDELOUAHAB et al. 2018) :

- AlexNet (KRIZHEVSKY, SUTSKEVER et Geoffrey E HINTON 2012) : 91,9 % ;
- VGG-16 (SIMONYAN et ZISSERMAN 2015) : 99,2 % ;
- VGG-19 (SIMONYAN et ZISSERMAN 2015) : 99,4 % ;
- GoogLeNet (SZEGEDY et al. 2015) : plus de 99,9 % ;
- ResNet-50, ResNet-101 et ResNet-152 (HE et al. 2016) : plus de 99,9 %.

Elles représentent donc la grande majorité des calculs nécessaires à l'inférence et cette tendance est à la hausse. Les couches de convolution sont par conséquent l'objet de nombreux travaux de recherche. Dans la suite, nous présentons une sélection de ces travaux, d'abord sur les approches *algorithmiques* (changer le calcul pour nécessiter moins de ressources) puis sur la représentation des données (changer le type de données pour nécessiter moins de ressources) et enfin spécifiques aux implémentations.

3.1.3 Optimisations algorithmiques

Cette section présente quelques optimisations du calcul de convolution au niveau algorithmique.

Élagage L'élagage (*pruning*) permet de réduire le nombre d'opérations et la mémoire requises en supprimant des *connexions* et des *neurones* dans une couche (HAN, POOL et al. 2015). Cela revient à considérer que certains paramètres (poids) sont nuls et que certains résultats ne sont pas calculés. Dans l'implémentation, la mémorisation de ces poids nuls peut être optimisée et les multiplications par des poids nuls peuvent être évitées. Les résultats élagués peuvent ne pas être mémorisés et toutes leurs *connexions sortantes* peuvent ne pas être calculées. On peut donc considérer que l'apprentissage donne, en plus des valeurs des paramètres, quelles connexions doivent être conservées. Le choix des connexions à conserver peut être fait, après une première phase d'apprentissage sans élagage, selon un seuil (HAN, POOL et al. 2015) ou en modélisant leur consommation d'énergie et leur impact sur le résultat du modèle (YANG, Y.-H. CHEN et Vivienne SZE 2017). Cette méthode a permis de réduire par 9 à 13 fois le nombre de paramètres de modèles (HAN, POOL et al. 2015) ou de réduire leur consommation d'énergie par un facteur 1,6 (GoogLeNet) ou 3,7 (AlexNet) (YANG, Y.-H. CHEN et Vivienne SZE 2017). L'architecture matérielle doit évidemment être conçue pour mettre à profit la faible densité des données (LU et al. 2019 ; HAN, LIU et al. 2016).

Multiplication matrice-matrice Les convolutions discrètes sont essentiellement des produits scalaires et peuvent bénéficier des implémentations très optimisées des multiplications matrice-matrice. Cela permet d'optimiser l'optimisation et la mise en œuvre de l'algèbre linéaire (comme les implémentations de BLAS et LAPACK) aussi bien sur CPU et GPU (CHELLAPILLA, PURI et SIMARD 2006) que sur FPGA (SUDA et al. 2016), d'utiliser des algorithmes de multiplication de matrice plus efficace que $\Theta(n^3)$ et optimisés pour la hiérarchie mémoire de l'architecture utilisée. En revanche, cette méthode introduit de la redondance dans la représentation des opérandes, pouvant donc nécessiter plus de mémoire ou des accès mémoires plus complexes (V. SZE et al. 2017).

Cette manière de calculer une convolution revient simplement à réorganiser les opérandes et le résultat pour *exprimer* les produits scalaires sous cette forme. Par exemple, considérons la convolution 2D suivante, avec une entrée x de taille 4×4 convoluée par un filtre w de taille 3×3 :

$$y = x * w = \begin{bmatrix} -5 & -3 & -1 & 1 \\ 7 & -6 & -3 & 9 \\ -5 & -5 & 7 & -4 \\ 9 & 2 & 2 & 5 \end{bmatrix} * \begin{bmatrix} 2 & -4 & 0 \\ -2 & 1 & 3 \\ 4 & -1 & -3 \end{bmatrix} = \begin{bmatrix} -63 & 19 \\ 92 & -4 \end{bmatrix}$$

La première valeur du résultat (-63) est le résultat du produit scalaire du coin supérieur gauche de l'entrée (-5, -3, -1, 7, …, -5, 7) par les valeurs du filtre. Il en va de même pour les autres valeurs. On peut donc réorganiser les opérandes et le résultat pour *exprimer* les produits scalaires sous cette forme. On peut donc réécrire la convolution d'exemple sous forme de produit de matrice comme suit :

$$y = \begin{bmatrix} -5 & 7 & -5 & -3 & -6 & -5 & -1 & -3 & 7 \\ 7 & -5 & 9 & -6 & -5 & 2 & -3 & 7 & 2 \\ -3 & -6 & -5 & -1 & -3 & 7 & 1 & 9 & -4 \\ -6 & -5 & 2 & -3 & 7 & 2 & 9 & -4 & 5 \end{bmatrix} \times \begin{bmatrix} 2 \\ -2 \\ 4 \\ -4 \\ 1 \\ -1 \\ 0 \\ 3 \\ -3 \end{bmatrix} = \begin{bmatrix} -63 \\ 92 \\ 19 \\ -4 \end{bmatrix}$$

Pour cet exemple, le calcul nécessite 36 multiplications et 32 additions si la multiplication de matrice est calculée naïvement, tout comme le calcul de convolution de référence.

Étant donné que le filtre est appliqué comme une *fenêtre glissante* sur l'entrée, il y a naturellement une redondance dans la matrice gauche (sauf si $S \geqslant K$) à un taux de l'ordre de $\frac{K^2}{S^2}$. Il est possible d'obtenir des matrices de la taille voulue (afin d'obtenir de meilleures performances) en utilisant plusieurs filtres (concaténés dans la matrice droite horizontalement pour les filtres sur la dimension M, verticalement pour les filtres sur la dimension N) ou en utilisant plusieurs entrées sur la dimension N (concaténées horizontalement dans la matrice gauche). Avec les dimensions de convolution introduits ci-dessus, une couche de convolution complète peut être calculée avec un seul produit d'une matrice $RC \times NK^2$ par une matrice $NK^2 \times M$.

Transformations de Fourier Une convolution dans le domaine réel revient à une simple multiplication élément par élément dans le domaine fréquentiel. Cette propriété peut être utilisée pour calculer une convolution 2D en trois étapes, technique classique en traitement d'image :

1. Transformation de Fourier discrète (*Discrete Fourier Transform*, DFT) des entrées et des filtres ;
2. convolution dans le domaine fréquentiel, c'est-à-dire multiplication élément par

élément ;

3. transformation inverse (IDFT) du résultat.

Une transformée de Fourier rapide (*Fast Fourier Transform*, FFT), algorithme pour calculer une DFT ou une IDFT, nécessite $\Theta(n \log n)$ opérations, où n est la taille du signal (dans le cas unidimensionnel). Une FFT bidimensionnelle est la composition de deux FFT unidimensionnelles et nécessite, avec nos notations de dimensions de convolution, $\Theta(RC \log(RC))$ opérations [2]. La multiplication dans le domaine fréquentiel nécessite $4RC$ opérations parce que les valeurs sont représentées par des nombres complexes. Au total, un calcul de convolution 2D basée sur cette méthode nécessite donc $\Theta(3RC \log RC + 4RC)$ opérations plutôt que $\Theta(2RCK^2)$.

Cette technique peut être utilisée pour les convolutions de CNN (Mathieu, Henaff et Yann LeCun 2014). De plus, chaque carte de fonction de sortie étant une somme de convolutions 2D, les résultats de chaque convolution 2D peuvent être sommés dans le domaine fréquentiel parce que la transformation de Fourier est une opération linéaire. Une couche de convolution nécessite alors seulement M transformées inverses et non NM.

Cette méthode réduit le nombre d'opérations nécessaires (principalement les multiplications mais augmente les additions) et augmente les besoins en mémoire (parce que les coefficients sont complexes et à cause du *padding* nécessaire). Il est aussi possible de calculer la transformée du filtre à l'avance ou de la calculer une seule fois pour plusieurs convolutions si les mêmes filtres sont appliqués à plusieurs entrées (*batching*).

Les performances de l'approche basée sur la FFT sont limitées par l'écart entre la taille du noyau de convolution et celle de l'image [3], cette dernière étant beaucoup plus grande. Pour pallier ce problème, il est possible d'utiliser la technique du *Overlap-and-Add* (OaA) qui consiste à morceler l'image et convoluer chaque bloc avec le filtre via une transformation de Fourier (en utilisant du *padding* pour éviter les effets de bord). Ainsi, la taille effective de l'image peut être plus finement choisie et permet de réduire le nombre d'opérations à $\Theta(RC \log_2 K)$ par convolution 2D (Chi Zhang et Prasanna 2017).

De plus, les avantages d'un éventuel élagage sont annulés par cette méthode (V. Sze et al. 2017).

2. Les FFT sur l'image et le filtre doivent être de la même dimension parce que les deux signaux doivent avoir la même dimension. On considère ici que $K^2 \leqslant RC$, ce qui est très généralement le cas.

3. K^2 est souvent du même ordre de grandeur que $\log_2 RC$, par exemple pour les trois dernières couches de convolution d'AlexNet : $K^2 = 9$ et $log_2 RC \simeq 7.4$.

Algorithmes minimaux de filtrage de Winograd La transformation de Winograd peut être utilisée pour calculer les sommes de convolutions 2D (Lavin et Gray 2016). Elle minimise le nombre de multiplications requises au prix d'un nombre d'additions plus élevé, ce qui est intéressant parce que la multiplication coûte généralement beaucoup plus cher que l'addition.

Cette approche est basée sur les algorithmes minimaux de filtrage (*minimal filtering algorithms*) de Winograd (Winograd 1980), des algorithmes pour calculer des convolutions. Winograd a proposé de noter $F(m,n)$ l'algorithme minimal calculant m sorties pour un filtre 1D de n valeurs. (Il est possible d'imbriquer ces algorithmes pour calculer des convolutions multidimensionnelles.) Il a prouvé que $F(m,n)$ nécessite exactement $m+n-1$ multiplications. Par exemple, pour $F(2,3)$, Winograd a proposé de calculer la convolution *via* des valeurs nommées m_1, m_2, m_3 et m_4 telles que [4] :

$$y = \begin{bmatrix} x_0 & x_1 & x_2 \\ x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

avec :

$$m_1 = (x_0 - x_2)w_0 \qquad m_2 = (x_1 + x_2)\frac{w_0 + w_1 + w_2}{2}$$

$$m_4 = (x_1 - x_3)w_2 \qquad m_3 = (x_2 - x_1)\frac{w_0 - w_1 + w_2}{2}$$

Ou, écrit sous forme matricielle [5] (Lavin et Gray 2016) :

$$y = A^\intercal \left[(Gw) \odot (B^\intercal x)\right]$$

où :

$$A^\intercal = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \qquad B^\intercal = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \qquad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

4. On reconnaît l'écriture de la convolution de x et w sous forme de multiplication de matrice.
5. L'opérateur binaire $\odot$ désigne la multiplication élément par élément (produit de Hadamard) et $A^\intercal$ la transposée de A.

Ce calcul nécessite bien $3+2-1=4$ multiplications (ainsi que 12 additions au total) à la place des 6 multiplications utilisées dans la convolution naïve : ce produit de matrice est donc minimal. De plus, comme pour la transformation de Fourier, les opérations sur les filtres (multiplicandes droites des définitions de m) peuvent être précalculées ou réutilisées pour plusieurs convolutions. D'autres algorithmes minimaux de filtrage existent pour d'autres valeurs de m et n. Il est aussi possible de calculer la convolution bloc par bloc afin de choisir la valeur de m.

Pour une convolution 2D, il faut « imbriquer » cet algorithme deux fois, ce qu'on note $F(2\times 2, 3\times 3)$. On obtient alors :

$$y = A^{\intercal}\left[(GwG^{\intercal}) \odot (B^{\intercal}xB)\right]A$$

Cette technique utilise 16 multiplications plutôt que 36, soit un gain de facteur 2,25.

L'efficacité de cette approche dépend de la taille des filtres K et de la taille de *bloc* choisie, c'est-à-dire de $F(m,n)$. Une taille de bloc plus grande réduit les multiplications nécessaires pour un coût de transformation plus important (V. Sze et al. 2017). Cette approche est particulièrement efficace quand la K est petit (ABDELOUAHAB et al. 2018), ce qui est souvent le cas pour les modèles de CNN modernes. Cependant, cette transformation ne peut être utilisée que si le pas est unitaire (ABDELOUAHAB et al. 2018).

3.1.4 Représentation des données

Lorsque l'apprentissage ou l'inférence d'un CNN ont lieu sur processeur ou GPU, elles utilisent très souvent le format à virgule flottante simple précision (*binary32*). En effet, ce format est quasi universel et est largement supporté par ces plateformes. Ce choix est pertinent pour une implémentation logicielle grâce aux bonnes performances des opérateurs flottants sur ces plateformes et aux propriétés de ce format (amplitude et précision suffisantes). De plus, utiliser le même format pour la phase d'entraînement et la phase d'inférence ne nécessite pas de traitement supplémentaire sur les paramètres du réseau entre les deux phases.

Cependant, ce format à virgule flottante n'est pas le plus efficace (en performance et en efficacité énergétique) sur d'autres plateformes pouvant être utilisées pour l'inférence (FPGA, ASIC, CPU sans FPU, co-processeur, etc.). Les formats utilisés sont plutôt des formats entiers (ou à virgule fixe) ou des formats à virgule flottante plus petits. Une telle

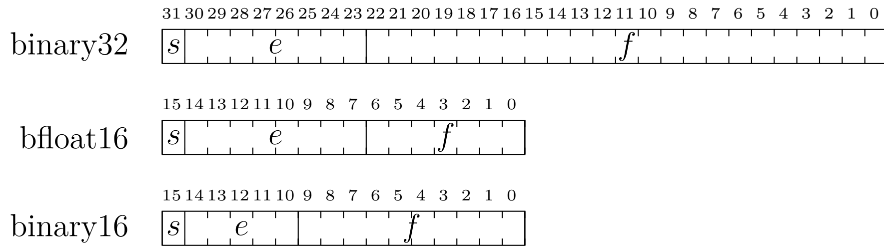


FIGURE 3.5 – Comparaison de la représentation en mémoire d'un nombre flottant au format IEEE-754 simple précision (*binary32*), au format demi précision (*binary16*) et au format non standard *bfloat16*. Tous ces formats ont la même interprétation : le format *bfloat16* a donc la même amplitude (autant de bits d'exposant) que le format *binary32* et est moins précis que le format *binary16* (moins de bits significatifs). Le format *bfloat16* utilise deux fois moins de mémoire que le format *binary16* et ses opérateurs sont moins coûteux.

transformation d'un format numérique à un autre provocant une perte d'information est appelée *quantification*. Dans le cas des couches de convolution, les entrées, les poids et les résultats intermédiaires peuvent être quantifiés ou non. Ces quantifications présentent plusieurs avantages (SANKARADAS et al. 2009 ; CAI et al. 2020 ; V. SZE et al. 2017) :

— réduction de la bande passante nécessaire (ou augmentation du débit) et réduction de l'énergie requise par les déplacements de mémoire ;
— réduction de la mémoire nécessaire ;
— réduction du matériel nécessaire parce que les opérations arithmétiques sont plus simples et réduction de l'énergie requise ;
— réduction de la latence de l'inférence.

Le détail des techniques de quantification est en dehors du périmètre de ce travail.

Il est possible de conserver un format à virgule flottante nécessitant moins de bits et moins de ressources que les formats classiques. Par exemple, le format *bfloat16* est un format à virgule flottante sur 16 bits ayant la même amplitude que le format *binary32* (voir figure 3.5), très utilisé en apprentissage machine. D'autres travaux mentionnent des formats plus exotiques, par exemple une représentation logarithmique sur 8 bits (JOHNSON 2018).

Il est aussi possible d'utiliser des formats entiers. Des travaux ont montré que des quantifications jusqu'à des formats entiers 8 bits, 4 bits ou même 1 bit peuvent être utilisées sans impacter fortement les résultats des réseaux de neurones. Certaines techniques ajoutent cependant une seconde phase d'apprentissage pour adapter le réseau au nouveau format de données utilisé et améliorer la qualité des résultats (QIU et al. 2016 ;

interprétation			représentation binaire		
x	w	$x \times w$	x	w	$\overline{x \oplus w}$
-1	-1	$+1$	0	0	1
-1	$+1$	-1	0	1	0
$+1$	-1	-1	1	0	0
$+1$	$+1$	$+1$	1	1	1
y_0	y_1	$y_0 + y_1$	y_0	y_1	popcount(y)
-1	-1	-2	0	0	00
-1	$+1$	0	0	1	01
$+1$	-1	0	1	0	01
$+1$	$+1$	$+2$	1	1	10

TABLE 3.1 – Correspondance entre l'interprétation des valeurs et leur représentation binaire pour la multiplication et l'addition dans un réseau de neurones convolutif binaire (l'addition peut être généralisée à plus de deux opérandes)

RASTEGARI et al. 2016).

À l'extrême, les poids et les entrées peuvent être binaires (représentant les valeurs -1 et 1, parfois interprétées telles quelles, parfois multipliées par un facteur). Cette approche a l'avantage de réduire considérablement la complexité des opérateurs : le produit scalaire est réalisé par un simple opérateur ou-exclusif pour la multiplication et par un comptage de bit (*popcount*) pour l'addition [6] (COURBARIAUX et al. 2016) (voir la table 3.1). Elles permettent, par exemple, une réduction du temps de calcul des convolutions par un facteur 58 sur CPU (RASTEGARI et al. 2016) sans perte de niveau de précision sur les résultats du réseau. Ces opérations bit à bit permettent aussi des implémentations efficaces sur GPU (COURBARIAUX et al. 2016) et FPGA (UMUROGLU et al. 2017).

Pour limiter l'impact d'une quantification trop importante, il est possible d'utiliser des données ternaires (représentant les valeurs $-1, 0, 1$). Cette approche a aussi l'avantage de permettre l'élagage. Les multiplications par 0, 1 ou -1 peuvent être effectuées par un additionneur de la taille de l'entrée (permettant de calculer l'opposé de l'entrée le cas échéant). Les valeurs ternaires peuvent être représentées sur deux bits en utilisant une représentation en complément à deux classique (11, 00, 01) ou peuvent être compressées pour économiser de la mémoire et de la bande passante au prix de logique supplémen-

6. Les sommes des résultats des convolutions 2D doivent cependant avoir recours à des opérateurs classiques parce que les opérandes (les résultats des produits scalaires) sont sur plusieurs bits.

taire [7] (Prost-Boucle, Bourge et Pétrot 2018). Alemdar et al. utilisent deux réseaux de neurones pour l'entraînement : un classique et un dont les poids (et valeurs d'activations) sont contraints aux valeurs ternaires (Alemdar et al. 2017). Lors de l'utilisation du réseau final (phase d'inférence), les entrées, les poids et les valeurs d'activations sont toutes ternaires.

3.1.5 Mises en œuvre sur FPGA

Comme présenté dans la section 1.4.3, les architectures dédiées ont un vaste espace de conception dû aux nombreux choix possibles. Nous présentons ici les principaux choix pour un accélérateur de convolution.

Tailles des tuiles, parallélisme et réutilisation des données Les données nécessaires pour calculer une couche de convolution, entrées et poids, ne peuvent souvent pas être stockées entièrement dans la mémoire interne du FPGA. Les données sont alors découpées en blocs, appelées *tuiles* (voir figure 3.6). Une implémentation typique s'organise alors autour de trois opérations majeures parallèles, en numérotant les tuiles par ordre chronologique de calcul :

- copie de la tuile d'opérandes (données et coefficients) n° n de la mémoire externe vers la mémoire interne ;
- calcul de la convolution sur la tuile n° $n-1$;
- copie de la tuile résultat n° $n-2$ de la mémoire interne vers la mémoire externe.

Ce *macro-pipeline* doit être équilibré afin de maximiser les performances. Les dimensions des tuiles sont choisies selon les contraintes de mémoire de la plateforme. Cette organisation est par exemple utilisée par Zhang et al. (Chen Zhang et al. 2015).

La convolution offre un haut niveau de parallélisme : toutes les MRC valeurs de sortie sont indépendantes et peuvent donc être calculées en parallèle. De plus, chacune de ces valeurs nécessite NK^2 multiplications-accumulations (MAC) qui peuvent également être en partie parallélisées. Le *niveau de parallélisme* doit donc être choisi en fonction des capacités de calculs de la plateforme, notamment des capacités de stockage interne.

En revanche, le choix des dimensions à paralléliser (ou, de manière équivalente, l'ordonnancement des calculs) influence les copies de données nécessaires. En effet, une convo-

7. La limite théorique est de $\log_2(3) \approx 1.585$ bits par donnée ternaire, soit un taux de compression de 20,75 %. En compressant par exemple 5 données ternaires ($3^5 = 243$ valeurs possibles) en 8 bits ($2^8 = 256$ valeurs possibles), le taux est de 20 %.

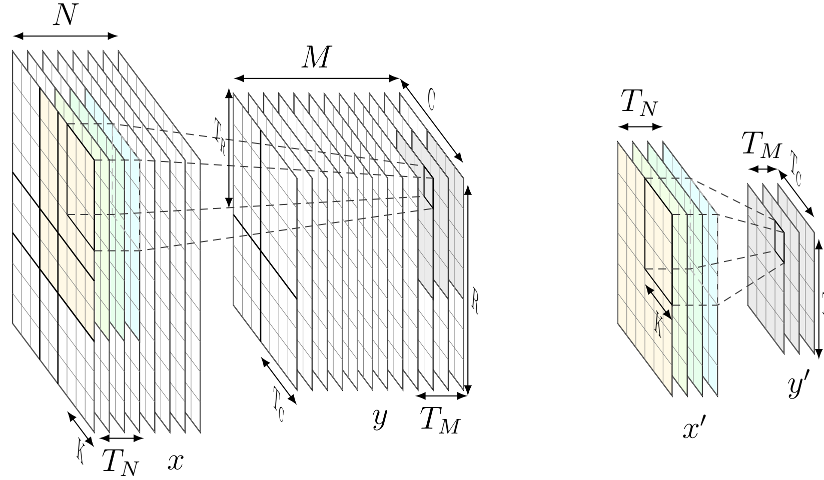


FIGURE 3.6 – Représentation graphique d'une tuile au sein une couche de convolution à gauche. À droite, représentation de la tuile isolée, correspondant à une convolution plus petite. Ses paramètres sont $T_R = T_C = 4$, $T_N = 4$, $T_M = 3$. Les tuiles d'entrées de x se chevauchent. La convolution complète nécessite le calcul de 32 tuiles, chacune des 16 tuiles des sorties nécessite 2 tuiles de filtres (non représentées) et 2 tuiles d'entrées.

lution expose les réutilisations de données suivantes :

- les cartes de fonction d'entrées sont réutilisées M fois ;
- chaque valeur des cartes de fonction d'entrée est réutilisée K^2/S^2 fois ;
- chaque valeur de poids est réutilisé RC fois ;
- les convolutions peuvent être calculées en lot (*batch*) afin de réutiliser les filtres.

L'ordonnancement des calculs (tuiles, parallélisation) a donc intérêt à permettre la réutilisation des données pour éviter les copies depuis la mémoire externe. Chen et al. proposent une taxonomie des différents flots de données possibles et montrent leurs performances ainsi que leur influence sur l'efficacité énergétique des solutions (Y.-H. CHEN, EMER et Vivienne SZE 2016).

Éléments de mise en œuvre La mise en œuvre sur FPGA se base sur deux éléments principaux : les mémoires et les *processing elements* (PE), éléments de calculs (Chen ZHANG et al. 2015 ; Y.-H. CHEN, EMER et Vivienne SZE 2016). Pour la convolution, les PE permettent de calculer des opérations MAC selon un certain degré de parallélisme. En plus de ses opérateurs, organisés en *pipeline*, chaque PE utilise un registre pour stocker son résultat courant et des mémoires locales pour ses opérandes. La mémoire doit donc être correctement organisée et exposer suffisamment de ports. Les données des tuiles sont

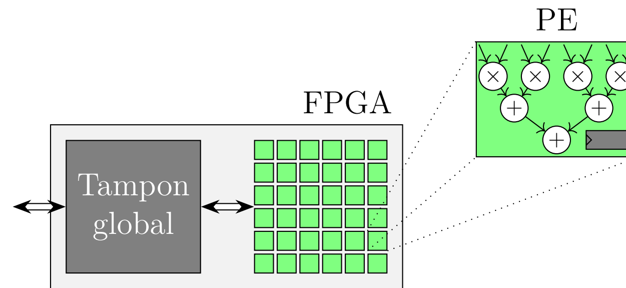


FIGURE 3.7 – Mise en œuvre sur FPGA à base de *Processing Elements*

donc réparties dans les blocs mémoires du FPGA. Selon l'organisation des calculs, les données peuvent transiter entre les PE ou via un tampon global. Un contrôleur gère l'ordonnancement des opérations : comptage des itérations, copies des données, etc. La figure 3.7 illustre cette architecture avec 36 PE calculant 4 MAC par cycle.

Architectures en *streaming* et moteur de calcul unique Les éléments de mise en œuvre présentés ci-dessus permettent de calculer une unique couche de convolution et non une inférence complète. Celle-ci nécessite de calculer toutes les couches du réseau de neurones. Il existe plusieurs manières d'organiser les calculs d'un CNN complet. Venieris et al. proposent une distinction entre deux types principaux : l'architecture fonctionnant en *streaming* et le moteur de calcul unique (VENIERIS, KOURIS et BOUGANIS 2018).

Le premier type correspond à un calcul chaîné des différentes couches (convolutions, *pooling*, etc.) sur du matériel distinct. Chaque bloc peut donc être optimisé pour la couche concernée (dimensions, niveau de parallélisme, quantification, etc.). Les blocs forment un *pipeline*, les différentes couches sont donc calculées en parallèles. La figure 3.8 présente une telle architecture.

Le second type se base sur du matériel générique étant utilisé pour calculer toutes les couches. Le même matériel est utilisé séquentiellement pour calculer les différentes couches. La configuration du matériel, pilotée par logiciel, permet de choisir les opérations à effectuer mais aussi de fournir les poids. Cette approche a l'avantage d'être plus souple (plusieurs modèles de CNN sont supportés sans changer le *bitstream*) mais peut être moins efficace à cause de la généricité du l'implémentation. De plus, la configuration, se rapprochant parfois d'instructions, rajoute un surcoût. La figure 3.9 présente une telle architecture.

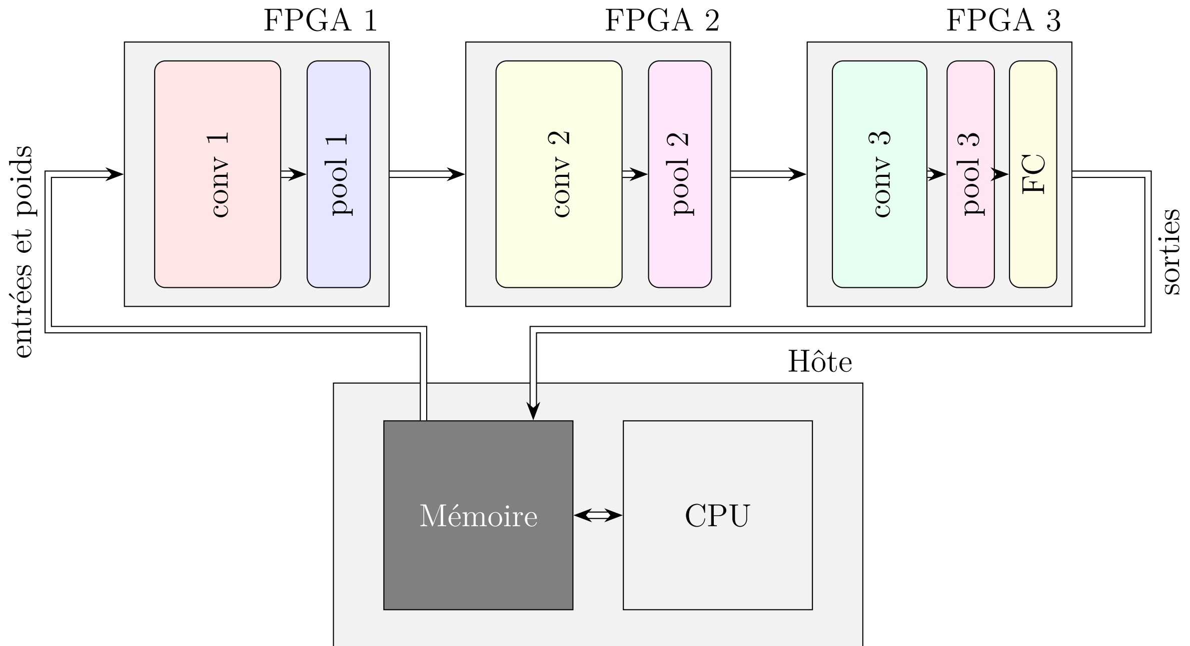


FIGURE 3.8 – Exemple d'architecture fonctionnant en *streaming* pour un CNN à trois couches de convolution, trois couches de *pooling* et une couche entièrement connectée, déployé sur trois FPGA

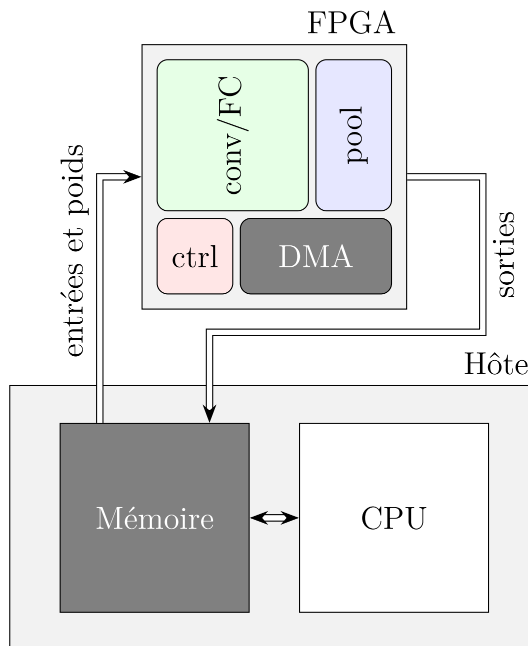


FIGURE 3.9 – Exemple de moteur de calcul unique avec un seul FPGA : les couches de convolution, de *pooling* et complètement connectées sont exécutées *tour à tour* sur le même matériel

3.2 Spéculation temporelle pour accélérateurs de convolution

Comme montré section 2.1, la spéculation temporelle peut permettre d'améliorer les performances et l'efficacité énergétique d'accélérateurs matériels. Cette section présente quelques travaux existants portant sur la spéculation temporelle d'accélérateurs de CNN.

Salami et al. proposent d'utiliser de l'*undervolting* pour un accélérateur de convolution sur FPGA pour améliorer son efficacité énergétique (Salami et al. 2020). Ils mesurent expérimentalement que la tension nominale des FPGA peut être réduite de 33 % en moyenne sans observer de perte de précision (mais possiblement des erreurs temporelles). La réduction de la tension permet un gain en efficacité énergétique de 160 %. Cependant, ils n'utilisent pas de mécanisme de détection ou de correction d'erreur. Ces travaux permettent donc de mesurer les marges possibles d'*undervolting* sur FPGA mais ne sont pas utilisables tels quels en production.

Zhang et al. montrent également des résultats d'*undervolting* sur une architecture de type TPU, avec détection d'erreur (J. Zhang et al. 2019). La détection d'erreur se base sur des bascules Razor (voir section 2.2.4.4). Seuls les opérateurs MAC sont protégés et seules 14 des 40 bascules de chaque opérateur sont protégées, les autres ne devant pas provoquer d'erreur à la tension minimale considérée d'après les simulations. Les auteurs estiment que cette protection nécessiterait 10 % de puissance supplémentaire. Lorsqu'une erreur temporelle est détectée par une bascule Razor, l'opérateur MAC concerné est réutilisé au cycle suivant pour finir correctement le calcul (permettant aux signaux trop lents de se propager). Le calcul qui aurait dû s'exécuter sur cet opérateur à ce cycle est simplement annulé, ce qui revient à injecter une autre erreur. Cependant, certains modèles sont entraînés en annulant aléatoirement certains calculs (*dropout*) et peuvent ainsi être considérés résilients à ce type d'erreur. Ce choix permet aux erreurs de n'avoir aucun impact négatif sur les performances et l'efficacité énergétique [8]. Cette méthode présente donc un compromis entre le gain en énergie et la précision des résultats (*accuracy*). Cependant, cette méthode utilise la méthode Razor, imparfait et surtout non applicable aux FPGA à cause des registres internes de certains composants comme les DSP et les BRAM.

8. Les auteurs justifient ce choix en montrant l'impact du taux d'erreur sur l'efficacité énergétique si les calculs erronés devaient être réexécutés : un taux d'erreur, même relativement faible, annulerait les gains dus à l'*undervolting*. Cependant, le taux d'erreur minimum (non nul) qu'ils considèrent est déjà assez élevé (0,1 %). En asservissant la tension au taux d'erreur, il est possible d'obtenir un taux plus faible et donc de limiter l'impact (voir la section 4.3.1 pour plus de détails).

Nunez-Yanez propose un mécanisme de DVFS pour réseaux de neurones binaires sur FPGA (J. L. NUNEZ-YANEZ 2018). La détection d'erreur se base sur le mécanisme Elongate (présenté section 2.2.4.4). Cette approche permet d'augmenter les performances (images traitées par seconde) de 86,8 % et l'efficacité énergétique (images traitées par watt) de 86,3 %. Nunez-Yanez propose aussi de tolérer quelques erreurs afin de proposer un compromis entre performances et précision. L'approche se base sur un simple compteur d'erreur, la fréquence ou la tension étant ajustée à partir d'un certain seuil (l'amplitude des erreurs n'est donc pas connue, seulement leur nombre). Il a montré empiriquement que les erreurs n'impactaient pas la précision jusqu'à un seuil de 45 000. En acceptant une perte de précision de 1 %, il est possible d'augmenter encore les performances entre 5 et 23 %. Cependant, ces différents seuils sont imprévisibles à causes des multiples sources de variabilité. Ces observations empiriques se basent sur des mesures de référence (*benchmarks*) : il n'est pas possible de connaître l'impact sur la précision dans le cas général. De plus, cette méthode n'a été testée que sur des CNN binaires.

3.3 Impact des erreurs sur la précision

Les réseaux de neurones artificiels utilisent un ensemble de « neurones » et de « synapses » pour calculer une fonction. Comme le nombre de neurones et de synapses d'un modèle n'est en général pas le nombre minimal requis pour réaliser une tâche donnée, les réseaux de neurones ont une résilience intrinsèque aux erreurs (TORRES-HUITZIL et GIRAU 2017). De plus, leur entrée est typiquement un signal (image, son, texte, etc.) et peut donc être bruitée. Grâce à l'apprentissage sur un corpus de données lui-même bruité, un réseau de neurones peut être tolérant au bruit. Une erreur de calcul n'implique donc pas forcément une erreur au niveau applicatif. Par exemple, pour une tâche de classification, la classe inférée par le réseau peut rester inchangée malgré des erreurs de calculs.

Cette propriété permet par exemple la quantification des données traitées et des paramètres du réseau : la quantification revient à injecter une erreur systématique sur toutes les données en supprimant leurs poids faibles, ce qui correspond à un bruit (voir section 3.1.4). Par exemple, Qiu et al. montrent que réduire la taille des données et des poids à un format à virgule fixe 8 bits (plutôt qu'un format à virgule flottante 32 bits) réduit la précision d'un réseau de seulement 0,57 points (QIU et al. 2016). Cette résilience peut aussi être utilisée pour tolérer des erreurs temporelles (J. ZHANG et al. 2019 ; J. L. NUNEZ-YANEZ 2018) (voir sous-section précédente). Libano et al. montrent que la quantification

à 1 bit d'un réseau de neurones implémenté sur FPGA augmente le risque qu'une erreur de calcul (due à un SEU) provoque une erreur de classification du réseau (Libano et al. 2020). Ruospo et al. montrent également que les réseaux de neurones deviennent moins résilients aux fautes de type *stuck-at* quand la largeur de mots des poids est réduite, pour des données représentées en virgule flottante ou en virgule fixe (Ruospo et al. 2021). Par conséquence, ne pas quantifier augmente bien la résilience du réseau de neurones, au prix d'un niveau de performance et d'une efficacité énergétique possiblement dégradées.

Des erreurs au niveau applicatif peuvent survenir soit parce que le modèle lui-même est peu résilient, soit parce que les erreurs de calculs sont nombreuses ou importantes. Les erreurs de quantification sont systématiques, prévisibles et bornées tandis que les erreurs temporelles sont imprévisibles et peuvent avoir une très grande amplitude (par exemple avec une inversion du bit de poids fort).

Jiao et al. ont analysé l'impact d'erreurs temporelles sur un réseau de neurones (Jiao et al. 2017). Ils ont collecté les erreurs temporelles observées par simulation d'additionneurs et de multiplieurs sous différentes conditions d'*undervolting* en faisant varier la tension et la température. Ils ont ensuite injecté ces erreurs à différents taux dans des calculs d'inférence d'un CNN classifiant des images en dix classes (MNIST). À un taux d'erreur de 0,001 %, la précision du réseau est déjà légèrement inférieure à la précision obtenue sans injection d'erreur. À un taux dix fois supérieur et cent fois supérieur, la précision baisse respectivement à 90 % et à 60 %. À partir de 1 % d'erreur, la précision tombe à 10 %, ce qui revient pour cette tâche à un tirage aléatoire. De plus, Li et al. montrent que les réseaux réalisant une tâche plus complexe (classification dans 1000 classes plutôt que 10) sont plus sensibles aux erreurs (Li et al. 2017).

Deng et al. ont analysé les résultats de réseaux de neurones artificiels affectés par des erreurs temporelles en utilisant des simulations au niveau porte logique (Deng et al. 2015). Pour simuler l'effet de l'*overclocking* ou de l'*underclocking*, ils modifient le délai de propagation de toutes les portes logiques du circuit avec un même délai de 10 % à 40 %. Ils montrent que la précision moyenne des réseaux de neurones baisse dès le taux minimum qu'ils utilisent, 10 %. À 30 % d'*overclocking*, la précision moyenne chute à 44 % (par rapport à des exécutions sans erreur). Ils montrent également qu'en utilisant les sorties erronées des réseaux de neurones dans le processus d'apprentissage, les réseaux peuvent être ré-entraînés afin de les rendre plus résilients à l'*overclocking*. Cette technique permet d'augmenter la précision moyenne à 72 % pour 30 % d'*overclocking*. (Il reste cependant des erreurs au niveau application et il ne s'agit pas d'un moyen de détecter les erreurs

temporelles.)

Ces résultats montrent que les erreurs temporelles peuvent détériorer les résultats d'un CNN, même à un faible taux d'erreur. Même dans le cas d'un CNN dit résilient (par exemple, sans quantification), le risque qu'une faute ait un impact au niveau applicatif n'est pas nul.

3.4 Détection d'erreur pour CNN

En plus des techniques présentées section 2.2.4, certaines approches spécifiques aux CNN peuvent être utilisées pour détecter les erreurs de calcul.

Par exemple, Lin et al. utilisent une technique nommée tolérance au bruit algorithmique (*Algorithmic Noise-Tolerance*, ANT) afin de corriger les erreurs temporelles dans le calcul de convolution d'un CNN (Lin, S. Zhang et Shanbhag 2016). Les erreurs qu'ils détectent sont dues à la variabilité importante ($\sigma/\mu = 39\%$) qui vient de l'utilisation d'une tension d'alimentation faible. La technique est basée sur la duplication du calcul principal, mais le calcul redondant est une estimation statistique du résultat exact. S'il y a une erreur trop importante entre les deux résultats (dépassant un seuil), on considère qu'une erreur s'est produite et on utilise l'estimation à la place du résultat erroné (qui a une erreur systématique mais contrôlée, à l'instar de la quantification). Le surcoût de la technique est directement lié au coût du calcul de l'estimateur. Selon les paramètres de la couche de convolution, les auteurs donnent un surcoût (calculé en nombre d'additionneurs complets) de quelques pourcents. L'inconvénient principal de cette méthode est qu'elle se base sur l'hypothèse que les erreurs du calcul principal auront une grande amplitude (les bits de poids forts sont erronés) alors que les erreurs de l'estimateur auront une faible amplitude correspondant à du bruit dû à la quantification (les bits de poids faible sont erronés), or nous avons observé que ce n'était pas le cas (voir section 4.3.8).

3.5 Conclusion

Nous avons présenté dans ce chapitre le fonctionnement des réseaux de neurones convolutifs et le fait que l'opération de convolution représente la grande majorité des calculs nécessaire à une inférence. La convolution des CNN est donc naturellement un traitement bénéficiant d'un large éventail de recherches dans le but d'augmenter ses performances et son efficacité énergétique. Cet algorithme est adapté aux architectures hautement pa-

rallèles, comme les FPGA, ce qui permet d'atteindre un bon niveau de performance. La spéculation temporelle permet d'améliorer les performances et l'efficacité énergétiques : les travaux existants sur FPGA montrent des gains potentiels de l'ordre de 30 % à 80 %. La spéculation temporelle implique un risque d'erreur, nous avons donc évalué ce que représentait ce risque au niveau applicatif (précision du réseau) et concluons que les erreurs temporelles ne peuvent pas, en général, être tolérées sans réduire la précision, la qualité du réseau de neurones. Nous avons donc présenté quelques mécanismes existants de détection d'erreur spécifiques aux CNN, mais ces mécanismes sont soit inadaptés à la détection d'erreur temporelles soit trop coûteux.

Chapitre 4

Spéculation temporelle sûre pour réseaux de neurones convolutifs

Nous avons montré dans les chapitres précédents que les accélérateurs matériels, parmi les différentes architectures de calcul, peuvent offrir un très bon compromis entre les performances d'une application et sa puissance électrique. Nous avons présenté la spéculation temporelle, un moyen d'améliorer les performances et l'énergie requise pour réaliser un calcul et nous avons montré que celle-ci doit être accompagnée d'un moyen de garantir la correction du calcul. Enfin, nous avons montré que les convolutions utilisées dans les CNN sont un bon candidat pour une implémentation pour accélérateur matériel.

Ce chapitre présente la contribution principale de ce travail. Nous commençons par présenter le mécanisme de détection d'erreur basé sur l'algorithme appliqué aux convolutions des réseaux de neurones convolutifs. Nous exposons les différents défis de mise en œuvre pour réaliser un accélérateur matériel utilisant cette technique. Nous présentons ensuite une évaluation de notre approche : quels gains nous avons pu observer sur notre prototype et les limites de validité. Enfin, nous entamons une discussion sur le potentiel et les limites de celle-ci.

4.1 ABFT pour les CNN

Dans cette section, nous décrivons notre nouveau mécanisme de détection d'erreurs pour les convolutions.

4.1.1 Vue d'ensemble

Notre approche s'appuie sur un mécanisme de détection des erreurs basé sur des propriétés algorithmiques. Elle partage certaines idées avec des travaux antérieurs sur la tolérance aux fautes au niveau algorithmique (*Algorithm Based Fault Tolerance*, ABFT)

Calcul	Additions	Multiplications
convolution	$\Theta(MNRCK^2)$	$\Theta(MNRCK^2)$
somme de contrôle de sortie	$\Theta(MRC)$	0
somme de contrôle d'entrée	$\Theta\left(NK^2(M+C)+NRCS^2\right)$	$\Theta(NK^2)$

TABLE 4.1 – Complexités des calculs de la convolution et des sommes de contrôle en nombre d'opérations d'additions et de multiplications

pour les opérations matricielles (HUANG et ABRAHAM 1984).

La détection d'erreur exploite un invariant du calcul de convolution. L'invariant est vérifié et, s'il est invalide, on conclut que le calcul est lui-même erroné. L'invariant est vérifié grâce à deux sommes de contrôle : une calculée sur le résultat (la sortie) de la convolution, l'autre calculée directement sur les opérandes (l'entrée). Le calcul est considéré comme erroné lorsque ces deux sommes de contrôle ne sont pas égales.

Ce mécanisme permet seulement de détecter les erreurs. Lorsqu'une erreur survient, il est nécessaire de recalculer entièrement la tuile erronée. La taille de la tuile utilisée pour la détection d'erreur permet un compromis entre le coût de la détection et le coût de la correction : plus une tuile est grande et plus le surcoût relatif de la détection est faible (grâce aux factorisations sur les dimensions tuilées R, C et M), mais plus le coût d'une correction est élevé (il y a plus de calculs à exécuter à nouveau, même pour une seule erreur). La tuile pour la détection d'erreur peut être différente de la tuile utilisée pour l'organisation des calculs sur le FPGA (par exemple, deux tuiles pour une, une tuile pour deux, etc.), mais par souci de simplification nous considérons que les deux types de tuiles sont identiques.

La somme de contrôle d'entrée est basée sur l'expression de la somme de contrôle de sortie dans lequel on substitue les sorties par leur expression en fonction des entrées. Cette expression, qui ne dépend donc plus des sorties, peut ensuite être simplifiée autant que possible. Ces simplifications permettent au calcul de la somme de contrôle d'entrée d'avoir une complexité inférieure à l'ensemble du calcul de convolution et du calcul de somme de contrôle de sortie. Sans elles, cela reviendrait à une simple redondance. La table 4.1 présente les complexités obtenues en appliquant les simplifications présentées dans cette section. Les dimensions sont celles introduites en section 3.1.2. La figure 4.1 donne le compte réel d'opérations pour les couches de convolution d'un modèle de CNN.

Nous expliquons d'abord le mécanisme de détection d'erreur sur les convolutions 1D et 2D afin de faciliter la compréhension. Nous poursuivons par l'application aux sommes

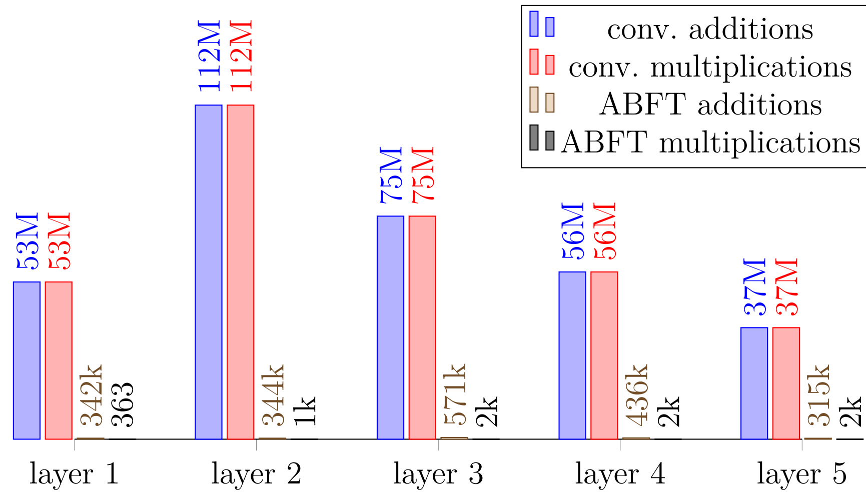


FIGURE 4.1 – Nombre d'opérations d'additions et de multiplications nécessaires pour la convolution et pour les sommes de contrôle pour les cinq couches de convolution d'AlexNet (KRIZHEVSKY, SUTSKEVER et Geoffrey E HINTON 2012). Toutes les couches ont un pas unitaire sauf la première qui a un pas de 4.

de convolutions 2D, c'est-à-dire les couches de convolution utilisées dans les CNN. Pour faciliter la lecture, nous considérons d'abord que le pas est unitaire ($S = 1$) avant d'étendre aux pas non unitaires.

4.1.2 ABFT pour convolution 1D

La convolution 1D est définie par l'équation suivante :

$$y_n = \sum_{k=0}^{K-1} x_{n+k} \cdot w_k$$

où y est le vecteur de sortie de taille N, w est le vecteur de poids de taille K et x est le vecteur d'entrée de taille $N + K - 1$.

La somme de contrôle de sortie σ est simplement la somme de tous les éléments du vecteur y :

$$\sigma = \sum_{n=0}^{N-1} y_n$$

Ce calcul nécessite $N - 1$ additions.

Comme expliqué précédemment, la somme de contrôle d'entrée ρ est basée sur l'ex-

pression de la somme de contrôle de sortie dans laquelle on substitue la définition de y :

$$\rho = \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} x_{n+k} \cdot w_k$$

Cette expression peut être simplifiée de plusieurs manières, expliquées ci-dessous, afin de réduire le coût de son calcul.

Factorisation La somme sur la dimension N peut être factorisée par les poids w :

$$\rho = \sum_{k=0}^{K-1} \left(w_k \cdot \sum_{n=0}^{N-1} x_{n+k} \right)$$

La factorisation permet d'effectuer le calcul avec seulement K multiplications plutôt que NK.

Réutilisation des sommes Les différentes sommes de x se chevauchent : elles ont des éléments en commun. On peut donc calculer l'une à partir de l'autre à partir de la différence entre les deux pour réduire le nombre de sommes nécessaires.

Soit X_k la somme multipliée par le poids w_k :

$$X_k = \sum_{n=0}^{N-1} x_{n+k}$$

Chaque somme nécessite $N - 1$ si on les calcule naïvement. Néanmoins, on remarque que chaque X est une somme sur un sous-ensemble légèrement différent de x en raison des décalages induits par k. Une fois qu'une valeur de X pour une instance spécifique de k est calculée, les instances restantes peuvent donc être calculées par différence. Le calcul de chacune des instances, sauf une, nécessitent donc seulement 2 sommes. Il existe de multiples façons de calculer X pour tirer parti de cette réutilisation, par exemple :

$$X_k = \begin{cases} \sum_{n=0}^{N-1} x_{n+k} & \text{si } k = 0 \\ X_{k-1} - x_{k-1} + x_{N-1+k} & \text{si } k > 0 \end{cases}$$

Le calcul de X_0 est effectué en premier, nécessitant $N - 1$ sommes. Ensuite, les $K - 1$ instances restantes de X sont calculées par addition et soustraction d'un élément de x.

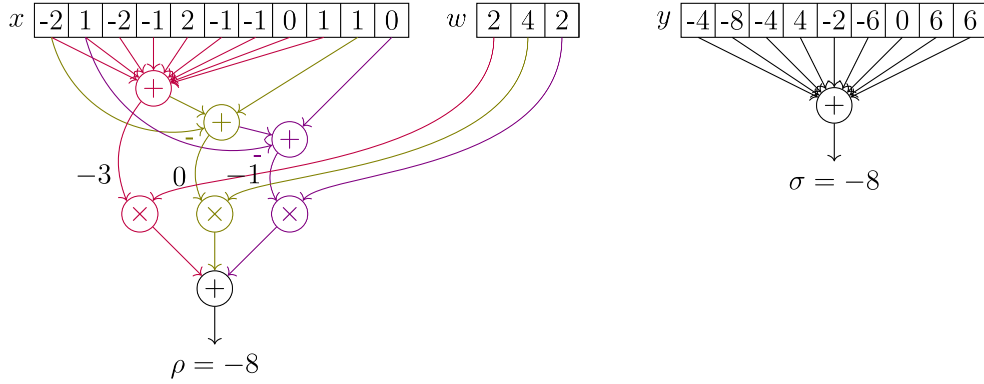


FIGURE 4.2 – Exemple de calcul des sommes de contrôle sur une convolution 1D. Les trois valeurs -3, 0 et -1 correspondent aux trois instances de X.

Ces simplifications permettent de réduire le nombre d'opérations nécessaires pour le calcul de la somme de contrôle d'entrée de $\Theta(NK)$ à $\Theta(N+K)$ additions et de NK à K multiplications. La figure 4.2 donne un exemple de calculs de sommes de contrôle. Pour cet exemple (avec $N = 9$ et $K = 3$), la convolution nécessite 18 opérations d'addition et 27 opérations de multiplication ; la somme de contrôle de sortie 8 additions et la somme de contrôle d'entrée 14 additions et 3 multiplications.

4.1.3 ABFT pour convolution 2D

Dans le cas 2D, la convolution est définie par l'équation suivante :

$$y_{r,c} = \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} x_{r+i,c+j} \cdot w_{i,j}$$

où y est la matrice sortie de taille $R \times C$, w est la matrice de poids de taille $K \times K$ et x est la matrice d'entrée de taille $(R - 1 + K) \times (C - 1 + K)$.

La somme de contrôle de sortie est toujours la somme de tous les éléments de la matrice de sortie :

$$\sigma = \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} y_{r,c}$$

Ce calcul nécessite $RC - 1$ additions.

En substituant y par sa définition, on obtient l'expression de la somme de contrôle

d'entrée à partir des opérandes :

$$\rho = \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} \left(\sum_{i=0}^{K-1} \sum_{j=0}^{K-1} x_{r+i,c+j} \cdot w_{i,j} \right)$$

Comme pour la convolution 1D, cette expression peut être simplifiée afin de réduire le coût du calcul : les mêmes simplifications que dans le cas 1D peuvent être appliquées mais sur les deux dimensions R et C.

Factorisation La somme sur les dimensions R et C peut être factorisée par les poids w :

$$\rho = \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \left(w_{i,j} \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} x_{r+i,c+j} \right)$$

Cela réduit le nombre de multiplications de RCK^2 à K^2.

Réutilisation des sommes Les ensembles d'éléments sommés pour être multipliés par les poids w se chevauchent sur deux dimensions. Soit $X_{i,j}$ la somme multipliée par le poids $w_{i,j}$:

$$X_{i,j} = \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} x_{r+i,c+j}$$

Chaque somme nécessite $RC-1$ si on les calcule naïvement. Chaque X est une somme sur un sous-ensemble légèrement différent de x en raison des décalages induits par i et j (voir figure 4.3a). Une fois qu'une valeur de X pour une instance spécifique de i, j est calculée, les instances restantes peuvent donc être calculées par différences comme dans le cas 1D, mais sur deux dimensions. Le calcul de chacune des instances, sauf une, nécessitent donc seulement $2C$ ou $2R$ sommes.

Une façon de tirer parti de cette réutilisation pour calculer les valeurs de X est la suivante :

$$X_{i,j} = \begin{cases} \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} x_{r,c} & \text{si } i = 0 \text{ et } j = 0 \\ X_{i,j-1} - \sum_{r=0}^{R-1} x_{r+i,j-1} + \sum_{r=0}^{R-1} x_{r+i,C-1+j} & \text{si } i = 0 \text{ et } j > 0 \\ X_{i-1,j} - \sum_{c=0}^{C-1} x_{i-1,c+j} + \sum_{c=0}^{C-1} x_{R-1+i,c+j} & \text{si } i > 0 \end{cases}$$

(a) Superposition des sous-ensembles multipliés par les poids (seuls 3 sous-ensembles sur 9 sont représentés)

(b) Réutilisation des sommes sur une dimension entre $X_{0,0}$ (en bleu) et $X_{0,1}$ (en vert)

FIGURE 4.3 – Superposition et réutilisation des sommes pour une convolution 2D

Le calcul de $X_{0,0}$ est effectué en premier, nécessitant $RC - 1$ sommes. Ensuite, les $K - 1$ autres instances de X sur la première ligne sont calculées par addition et soustraction d'une colonne (voir figure 4.3b). Enfin, les $K^2 - K$ instances restantes de X sont calculées par addition et soustraction d'une ligne. Cela correspond à l'optimisation utilisée dans le cas 1D appliquée dans la dimension C, puis dans la dimension R.

La combinaison des deux simplifications décrites ci-dessus réduit la complexité du calcul de la somme de contrôle d'entrée de $\Theta(RCK^2)$ produits et sommes à K^2 produits et $\Theta(RC)$ sommes.

4.1.4 ABFT pour sommes de convolution 2D (CNN)

Les convolutions utilisées dans les CNN, parfois appelées « convolutions 3D », sont en réalité des sommes de convolutions 2D, définies par l'équation suivante :

$$y_{m,r,c} = \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} x_{n,r+i,c+j} \cdot w_{m,n,i,j}$$

L'entrée x, tout comme la sortie y, a trois dimensions. La matrice de poids a quatre dimensions : chaque matrice 2D est unique pour une combinaison d'entrée et de sortie, les poids ne sont pas réutilisés.

La somme de contrôle de sortie est la somme des éléments de la matrice y :

$$\sigma = \sum_{m=0}^{M-1} \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} y_{m,r,c}$$

En substituant y par sa définition, on obtient l'expression de la somme de contrôle d'entrée à partir des opérandes :

$$\rho = \sum_{m=0}^{M-1} \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} \left(\sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} x_{n,r+i,c+j} \cdot w_{m,n,i,j} \right)$$

En plus des simplifications présentés pour le cas 2D, la somme sur la dimension M peut aussi être factorisée puisque m est invariant à l'expression de x :

$$\rho = \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \left(\left(\sum_{r=0}^{R-1} \sum_{c=0}^{C-1} x_{n,r+i,c+j} \right) \cdot \sum_{m=0}^{M-1} w_{m,n,i,j} \right)$$

Les autres simplifications sont similaires à une convolution 2D avec la factorisation de la somme sur les dimensions R et C ainsi que la réutilisation des sommes de x :

$$\rho = \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \left(X_{n,i,j} \cdot \sum_{m=0}^{M-1} w_{m,n,i,j} \right)$$

où X prend la même forme que le cas 2D, à l'exception de la dimension supplémentaire n, qui n'offre aucune réutilisation :

$$X_{n,i,j} = \begin{cases} \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} x_{n,r,c} & \text{si } i = 0 \text{ et } j = 0 \\ X_{n,i,j-1} - \sum_{r=0}^{R-1} x_{n,r+i,j-1} + \sum_{r=0}^{R-1} x_{n,r+i,C-1+j} & \text{si } i = 0 \text{ et } j > 0 \\ X_{n,i-1,j} - \sum_{c=0}^{C-1} x_{n,i-1,c+j} + \sum_{c=0}^{C-1} x_{n,R-1+i,c+j} & \text{si } i > 0 \end{cases}$$

Les trois simplifications décrites ci-dessus réduisent le coût du calcul de la somme de contrôle d'entrée de $\Theta(MNRCK^2)$ produits et sommes à NK^2 produits et $\Theta(NK^2(M + C) + NRC)$ sommes.

4.1.5 ABFT pour convolution avec pas non unitaire

Certains CNN utilisent des convolutions avec des pas non unitaires, en particulier pour leurs premières couches et les gros noyaux de convolution. Le pas est généralement proportionnel à la taille du noyau. Par exemple, le pas est typiquement de 2 pour des noyaux de taille 5×5, ou de 4 pour des noyaux de taille 11×11. Nous montrons comment

les sommes de contrôle peuvent être calculées avec un pas non unitaire, au prix d'une augmentation légère de la complexité.

Le pas, noté S, modifie le comportement de la convolution : le filtre est maintenant appliqué sur une fenêtre glissante sur l'entrée avec un pas de S dans les deux directions. Cela réduit la taille des sorties et le nombre d'opérations de multiplication-accumulation par un facteur de S^2.

Les pas non unitaires changent la manière dont chaque entrée est multipliée par les poids. Tous les poids ne sont plus utilisés pour chaque valeur d'entrée, mais seulement un poids sur S, sur les deux dimensions. On peut donc considérer une convolution à pas non unitaire comme un ensemble indépendant de convolutions à pas unitaire, chacune opérant sur une partition de x et w en prenant un élément sur S dans les deux dimensions. Pour une taille de matrice de sortie fixe $M \times R \times C$ les dimensions de la matrice x sont plus importantes : $N \times (S(R-1)+K) \times (S(C-1)+K)$.

La factorisation de la somme sur la dimension M est toujours possible, puisque m est toujours invariant à l'expression de x, affectée par le pas non unitaire.

La réutilisation dans les sommes sur les dimensions R et C est possible, si un poids est multiplié par plusieurs entrées, c'est-à-dire lorsque $S < K$ (ce qui est le cas normal) :

$$X_{n,i,j} = \begin{cases} \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} x_{n,Sr,Sc} & \text{si } i < S \text{ et } j < S \\ X_{n,i,j-S} - \sum_{r=0}^{R-1} x_{n,Sr+i,j-S} + \sum_{r=0}^{R-1} x_{n,Sr+i,S(C-1)+j} & \text{si } i < S \text{ et } j \geqslant S \\ X_{n,i-S,j} - \sum_{c=0}^{C-1} x_{n,i-S,Sc+j} + \sum_{c=0}^{C-1} x_{n,S(R-1)+i,Sc+j} & \text{si } i \geqslant S \end{cases}$$

Il y a deux différences par rapport au cas avec pas unitaire : il faut calculer complètement les $RC-1$ sommes pour S^2 instances de X (parce que elles ne partagent pas d'entrées) ; le calcul par différence est effectué avec un pas de S. Le cas $S=1$ est simplement un cas particulier.

Le calcul de la somme de contrôle de sortie est le même qu'avec un pas unitaire. Le pas ne change pas les dimensions de la matrice de sortie : la somme des éléments de la matrice de sortie nécessite le même nombre d'opérations.

Les trois simplifications décrites ci-dessus réduisent le coût du calcul de la somme de contrôle d'entrée de $\Theta(MNRCK^2)$ produits et sommes à NK^2 produits et $\Theta(NK^2(M+C)+NRCS^2)$ sommes.

4.1.6 Autres couches de CNN

Comme montré dans la section 3.1.2, les couches de convolution des CNN représentent plus de 99 % des calculs de différents modèles de CNN. Par conséquent, améliorer les performances des autres couches n'a pas beaucoup d'impact sur les performances globales. En outre, il est possible de les exécuter de manière sûre, sans *overclocking*, sans subir une trop grosse perte de performance. Il est aussi possible de protéger ces couches avec des méthodes génériques (DMR, TMR). Le surcoût relatif de ces méthodes est important mais, ces couches nécessitant peu d'opérations, le surcoût de la tolérance aux fautes pourrait être acceptable. L'évaluation des gains possibles grâce à la protection de ces couches est en dehors du périmètre de ce travail.

Une autre possibilité est de ne pas calculer ces couches sur FPGA, mais en logiciel. En effet, celles-ci nécessitant moins de calculs, elles peuvent être exécutées sur un processeur avec moins de parallélisme sans impacter les performances globales.

4.2 Accélérateur de convolution tolérant aux fautes

Dans cette section, nous décrivons plus en détails comment est conçu l'accélérateur de convolution et donnons un aperçu de la manière dont les erreurs et la fréquence peuvent être gérées.

4.2.1 Accélérateur de référence

L'accélérateur de convolution suit la stratégie de mise en œuvre proposée par Zhang et al. (Chen Zhang et al. 2015). Cette mise en œuvre a été choisie puisqu'elle correspond à nos besoins : elle cible les FPGA, est décrite et est réalisable avec des outils de synthèse de haut niveau et est relativement simple. Elle est organisée selon un *macro-pipeline* pour recouvrir les copies de données et les calculs (voir section 3.1.5). Les données sont organisées en tuiles : chaque tuile de sortie de taille $T_M \times T_R \times T_C$ nécessite $\left\lceil \frac{N}{T_N} \right\rceil$ couples de tuiles d'entrée de taille $T_N \times T_{R'} \times T_{C'}$ et de tuiles de poids de taille $T_N \times K \times K$ (voir section 3.1.2 pour les définitions des dimensions). Pour les calculs, l'accélérateur utilise des opérateurs de multiplications et des arbres d'additionneurs entiers en *pipeline*. Il contient U_M arbres effectuant U_N opération multiplication-accumulation par cycle. L'accélérateur diffère de la mise en œuvre de Zhang et al. principalement sur un point : l'ordre d'exécution des multiplications-accumulations est modifié (les boucles sont interchangées)

```
// itère sur la tuile de sortie
for(im = 0; im < Tm; im += Um) {
 for(ir = 0; ir < Tr; ir++) {
  for(ic = 0; ic < Tc; ic++) {
   // itère sur le filtre
   for(i = 0; i < K; i++) {
    for(j = 0; j < K; j++) {
     // itère sur la dimension libre de la
         tuile d'entrée
     for(in = 0; in < Tn; in += Un) {
     #pragma HLS PIPELINE
      // Um PE executant Un MAC par cycle
      for(um = 0; um < Um; um++) {
       for(un = 0; un < Un; un++) {
        pe[um] +=
         w[in + un][im + um][i][j] *
         x[in + un][S*ir+i][S*ic+j];
} } } } } } } }
```

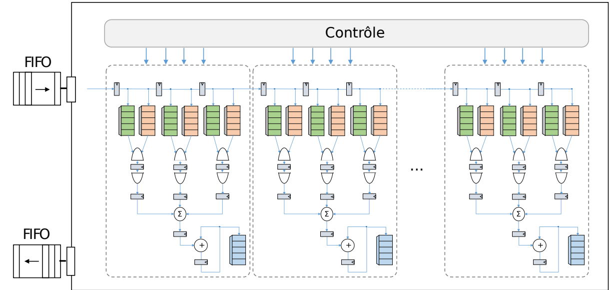


FIGURE 4.4 – Pseudo-code du cœur de calcul de l'accélérateur de convolution et vue d'ensemble de l'architecture

afin de simplifier le calcul de la somme de contrôle d'entrée (voir section 4.2.3.2). Cette architecture est illustrée figure 4.4.

Le choix de la taille des tuiles dépend de la mémoire disponible sur le FPGA. Le choix des paramètres U_M et U_N dépend des ressources de calcul du FPGA. Le débit de copies et de calculs doivent être équilibrées afin d'obtenir les meilleures performances possibles pour les ressources utilisées. En particulier, les copies de données ne doivent pas être le facteur limitant les performances parce que cela rendrait l'*overclocking* inefficace.

4.2.2 Détection d'erreur pour un accélérateur de convolution

En plus du calcul de convolution présenté dans la section précédente, le système doit réaliser les calculs de sommes de contrôle afin de pouvoir détecter les erreurs temporelles.

Ces calculs peuvent être réalisés par l'accélérateur de convolution (sur FPGA) ou ailleurs, par exemple sur le processeur d'un système hôte. Comme ces calculs nécessitent peu de ressources, une exécution de ces calculs sur FPGA contraint peu l'accélérateur complet et une mise en œuvre sur processeur est possible sous réserve de performances suffisamment élevées. Nous présentons ici la solution avec tous les calculs dans un même circuit, solution choisie pour ses gains en efficacité énergétique.

L'architecture complète est illustrée par la figure 4.5. Afin de préserver les performances de l'accélérateur, les modules de calculs de somme de contrôle d'entrée et de sortie sont conçus de manière à pouvoir maintenir le même débit d'entrée/sortie que

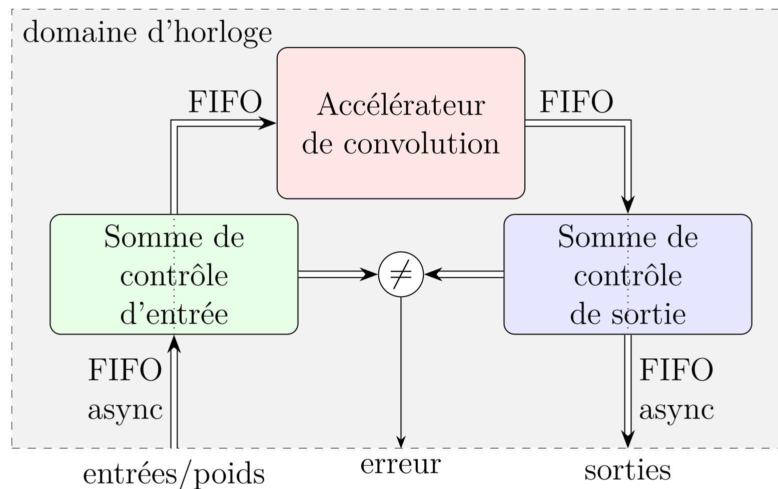


FIGURE 4.5 – Vue d'ensemble de l'accélérateur avec détection d'erreur

l'accélérateur de convolution lui-même. Les données transitent entre les trois modules de calcul par des mémoires tampon FIFO. Les modules de calcul de sommes de contrôle consomment les données d'une FIFO et les fournissent immédiatement au module suivant. La latence de calcul d'une tuile n'est donc pas impactée (ou très peu, quelques cycles) par l'ajout des calculs de sommes de contrôle. L'accélérateur se trouve dans son propre domaine d'horloge dont la fréquence est contrôlée par logiciel [1]. Les FIFO asynchrones permettent la communication entre le domaine d'horloge des composants principaux et le domaine d'horloge des composants de copie.

Dans notre mise en œuvre, l'accélérateur entier fonctionne dans le domaine d'horloge utilisé pour l'*overclocking*. Ce choix est fait pour simplifier la conception des composants et légèrement réduire le coût : davantage de ressources auraient été nécessaires pour compenser l'écart des horloges si deux domaines d'horloge avaient été utilisés. Ce choix comporte le risque que des erreurs temporelles surviennent dans les calculs de somme de contrôle aussi, mais ce risque est faible comme montré dans la section 4.3.2.

Un processeur contrôle l'accélérateur, lui fournit les tuiles à calculer et récupère les tuiles résultats, gère la gestion d'erreur (signalées par l'accélérateur) et pilote sa fréquence d'horloge.

1. Implémenté sur Xilinx *7 Series* grâce à la reconfiguration dynamique du module *Mixed-Mode Clock Manager* (MMCM).

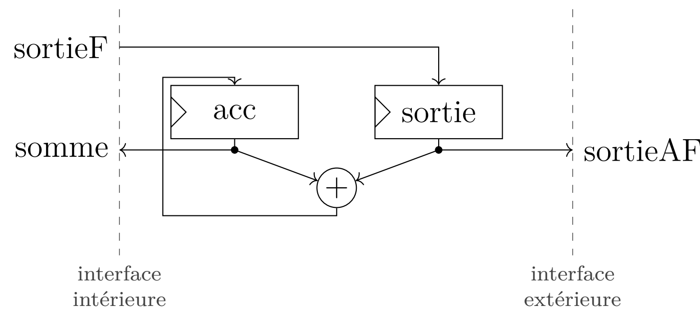


Figure 4.6 – Chemin de données du calcul de somme de contrôle de sortie

4.2.3 Mise en œuvre des calculs de sommes de contrôle

Comme montré dans la section 4.1, les calculs des sommes de contrôle sont moins complexes que le calcul de convolution. La principale contrainte de performance est de s'assurer que la cadence de traitement des données correspond à celui du calcul de convolution, afin de ne pas détériorer le débit de ce dernier. Ils doivent donc simplement avoir un niveau de parallélisme suffisant pour suivre les cadences d'entrée et de sortie, mais celui-ci doit être minimal afin de réduire au maximum le coût en ressources matérielles.

4.2.3.1 Somme de contrôle de sortie

Parce que le module de convolution produit ses résultats comme un flux de valeurs, la somme de contrôle de sortie peut être implémentée comme un simple accumulateur sur les sorties du calcul de convolution. Elle a donc un très faible surcoût en ressources.

L'accumulation peut trivialement être parallélisée au même niveau que le nombre de valeur copiées par cycle avec autant d'accumulateurs que nécessaires. Les valeurs des différents accumulateurs doivent simplement être sommées ensemble à la fin (sauf s'il n'y a qu'un accumulateur) puis réinitialisées.

La figure 4.6 représente une version simplifiée du chemin de données que nous utilisons pour calculer la somme de contrôle de sortie. Comme indiqué ci-dessus, le calcul s'effectue au niveau de l'interface entre le calcul de convolution via la FIFO `sortieF` et l'interface asynchrone avec l'extérieur via la FIFO asynchrone `sortieAF`. Les signaux de contrôle (machine à état qui contrôle la remise à zéro du registre `acc`, gestion des FIFO, etc.) ne sont pas représentés.

4.2.3.2 Somme de contrôle d'entrée

Le module de somme de contrôle d'entrée est plus complexe et coûteux, car il nécessite un multiplieur, de la mémoire pour les sommes partielles et une logique de contrôle non triviale. Pour rappel (voir section 4.1 pour les détails), le calcul est basé sur l'expression :

$$\rho = \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \left(X_{n,i,j} \cdot \sum_{m=0}^{M-1} w_{m,n,i,j} \right)$$

où :

$$X_{n,i,j} = \begin{cases} \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} x_{n,r,c} & \text{si } i = 0 \text{ et } j = 0 \\ X_{n,i,j-1} - \sum_{r=0}^{R-1} x_{n,r+i,j-1} + \sum_{r=0}^{R-1} x_{n,r+i,C-1+j} & \text{si } i = 0 \text{ et } j > 0 \\ X_{n,i-1,j} - \sum_{c=0}^{C-1} x_{n,i-1,c+j} + \sum_{c=0}^{C-1} x_{n,R-1+i,c+j} & \text{si } i > 0 \end{cases}$$

Le calcul est séparé en trois parties :

1. Une accumulation des coefficients des noyaux de convolution sur la dimension M au moment de leur copie sur la mémoire du FPGA (ce calcul correspond à la somme des w). $T_N K^2$ valeurs de $W + \lceil \log_2(T_M) \rceil$ bits doivent être mémorisées ;
2. Une accumulation des entrées en $(2K-1) \times (2K-1)$ groupes qui permettront de calculer les valeurs X. Comme pour les poids, les valeurs sont sommées au moment de leur copie. $T_N(2K-1)^2$ valeurs de $D + \lceil \log_2(T_R T_C) \rceil$ bits doivent être mémorisées ;
3. En parallèle du calcul de convolution, les valeurs de X sont calculées à partir des sommes de valeurs d'entrées. Ces valeurs sont ensuite multipliées par les sommes de poids correspondant et la somme de ces produits donne la valeur de la somme de contrôle d'entrée. K^2 valeurs de $W + \lceil \log_2(T_R T_C K^2) \rceil$ bits sont nécessaires pour mémoriser les valeurs de X.

L'étape 1 pourrait être faite hors-ligne parce que les coefficients sont connus d'avance. Cependant, les noyaux de convolution représentent généralement les plus gros transferts de données pour la convolution (comparés aux entrées et aux sorties). Un accélérateur est souvent utilisé sur des calculs par lots (*batch*) afin de réduire le poids relatif du transfert des noyaux. Or, précalculer ces sommes nécessiterait d'augmenter encore ce volume de transfert. Par conséquent, nous préférons calculer ces sommes à la volée, au prix de

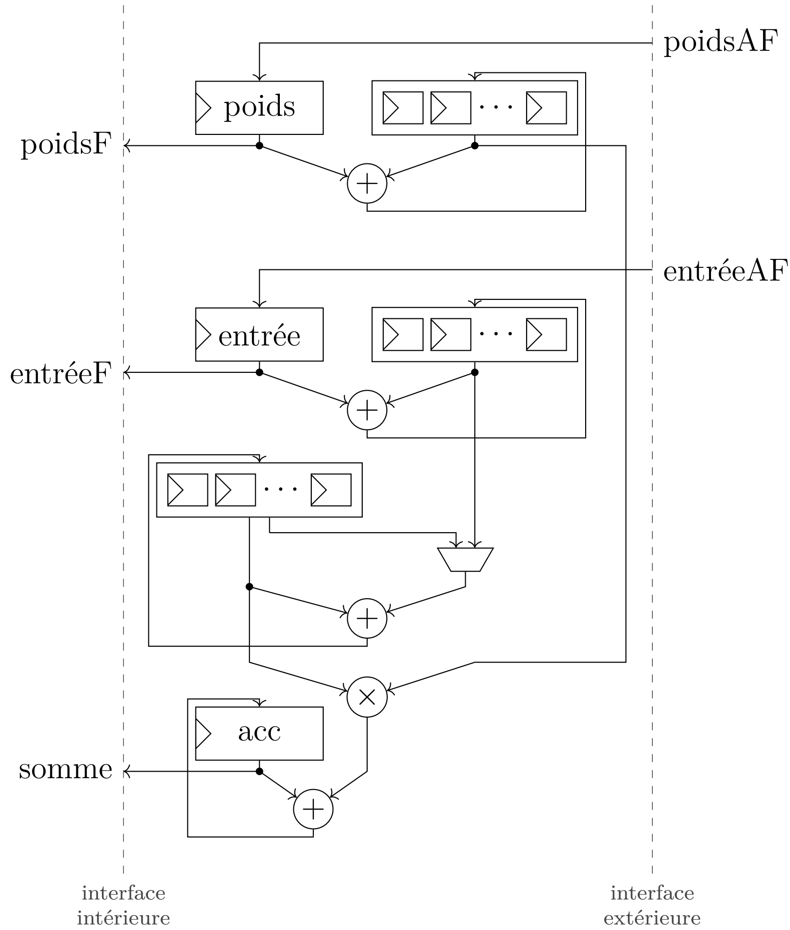


FIGURE 4.7 – Chemin de données du calcul de somme de contrôle d'entrée

quelques ressources matérielles supplémentaires.

Si besoin, cette étape peut être parallélisée facilement jusqu'à un facteur $T_N K^2$ en répliquant le chemin de données. Pour que ce soit possible, il faut cependant que deux données appartenant à la même somme (c'est-à-dire avec le même indice m) ne soient pas transmises simultanément. L'étape nécessite autant d'accumulateurs que le niveau de parallélisme choisi.

L'étape 2 peut facilement être parallélisée de la même manière jusqu'à un facteur T_N pour suivre la cadence de copie des entrées. Il est possible de paralléliser jusqu'à un facteur $T_N(2K-1)^2$, mais la somme des valeurs dans les groupes n'est pas régulière et cela nécessiterait plusieurs accumulateurs pour certains groupes.

Le chemin de données de la dernière étape peut trivialement être répliqué jusqu'à un facteur T_N, puisqu'aucune réutilisation ni factorisation n'a lieu sur la dimension N. Il est

possible de calculer certaines valeurs de X en parallèle ainsi que les produits.

La figure 4.7 illustre une architecture de calcul de somme de contrôle d'entrée. Les sommes sur les entrées et les poids sont, comme pour l'architecture de calcul de somme de contrôle de sortie, entre les interfaces synchrones et asynchrones. Cependant, comme expliqué ci-dessus, plusieurs valeurs doivent être mémorisées : l'architecture utilise donc de petits bancs de registres. Le calcul final de la somme de contrôle (étape 3 : calcul des valeurs de X, produits et somme finale) est effectué une fois que la tuile est copiée, en parallèle du calcul de convolution.

Bien entendu, l'exécution de tous ces calculs fonctionne en *macro-pipeline* de manière à suivre la cadence de copie minimale d'une donnée par cycle. Ces calculs sont parallélisables si besoin, mais il est nécessaire de les paralléliser uniquement à partir d'un haut niveau de parallélisation du calcul de convolution. En effet, ce dernier nécessitant de loin le plus de calculs, les performances peuvent ne pas être impactées même avec aucun parallélisme dans le calcul des sommes de contrôle.

4.2.4 Intégration complète

Comme mentionné dans la section 3.1, l'accélérateur du noyau de convolution n'est qu'un composant des accélérateurs de CNN. Dans cette section, nous présentons des éléments de réflexion sur l'intégration complète d'un CNN, c'est-à-dire comment obtenir une architecture de CNN complète. Nous présentons comment gérer la fréquence de fonctionnement des accélérateurs de convolution, comment de tels accélérateurs peuvent être intégrés dans différents types d'architecture de CNN.

4.2.4.1 Gestion de la fréquence

Un accélérateur avec détection d'erreur peut fonctionner à une fréquence supérieure à la fréquence nominale du circuit (*overclocking*) parce que la garantie de bon fonctionnement est apportée par la détection d'erreur et non plus par l'analyse statique déterminant la fréquence nominale. La fréquence doit être sélectionnée la plus élevée possible (afin d'augmenter au maximum les performances) tout en maintenant un taux d'erreur le plus faible possible [2] (afin d'éviter d'avoir à recalculer les tuiles erronées). Le choix de la fréquence de fonctionnement est donc réalisé à l'exécution et évolue avec le temps.

2. Le compromis entre les gains dus à l'*overclocking* et les surcoûts dus aux erreurs est donc subtil et est approfondi en section 4.3.1.

La fréquence doit donc être gérée en fonction du taux de tuiles erronées via une boucle de rétroaction. L'adaptation de la fréquence en temps réel est primordiale puisque la fréquence maximale à laquelle un accélérateur fonctionne sans erreur varie au cours du temps. L'étude approfondie (par exemple par la théorie du contrôle) de la gestion optimale de la fréquence va au-delà de la portée de ce travail. Nous présentons néanmoins quelques éléments de réflexion. Il faut toute fois noter que les gains d'une gestion optimale par rapport aux éléments présentés ne devraient pas être très importants parce que ces méthodes simples permettent déjà d'observer des taux d'erreurs très faibles pour des fréquences poussées proches de la limite.

De manière générale, la fréquence *doit* être diminuée si une tuile est erronée et *peut* être augmentée si une tuile n'est pas erronée. Il est possible d'imaginer plusieurs stratégies de gestion en fonction de plusieurs objectifs interdépendants (maximiser le débit ; minimiser la consommation globale d'énergie ; minimiser la latence moyenne soit le taux d'erreur ; etc.) :

— la fréquence peut être régulièrement augmentée si aucune erreur n'a été observée pendant un temps donné. En cas d'erreur, la fréquence peut immédiatement être réduite plus ou moins radicalement ;

— afin de trouver la fréquence optimale (à un instant t) plus rapidement, il est possible d'effectuer une recherche dichotomique (en invalidant les résultats des calculs ou en utilisant des données factices). Cette recherche peut être répétée régulièrement afin d'adapter la fréquence.

— après une tuile erronée, la fréquence peut être réduite ponctuellement uniquement pour le second calcul de cette tuile. En effet, les données jouant un rôle sur la fréquence maximale, une tuile particulière peut être erronée à cause d'une fréquence trop élevée sans qu'une fréquence plus faible soit nécessaire pour les autres tuiles ;

— etc.

Par ailleurs, un taux d'erreur non nul peut permettre de gagner en performance si les gains dus à l'augmentation de la fréquence surcompensent les pertes dues aux erreurs.

Une architecture avec plusieurs accélérateurs (figure 3.8) fonctionnant à différentes fréquences d'horloge ajoute une dimension au problème. En effet, les différentes couches (dont les temps de calcul sont normalement équilibrés) risquent de devenir déséquilibrées si les gains en fréquence ne sont pas similaires. Ce déséquilibre, s'il n'est pas correctement géré, détériorera les performances du système parce que certains accélérateurs seront bloqués. L'efficacité énergétique sera aussi impactée, en fonction de la prévalence de la puissance

statique sur la puissance dynamique du système (voir section 1.1). Si l'impact est important (dans le cas où la puissance statique prévaut, comme pour un FPGA), il est possible de l'atténuer en réduisant les fréquences afin de rééquilibrer les couches.

Pour une architecture réutilisant le même matériel pour plusieurs couches de convolution *logiques* (figure 3.9), il peut être intéressant de gérer plusieurs « profils » de fréquences pour chaque couche. En effet, les données (poids et entrées) des couches peuvent être différentes et impacter la fréquence maximale à laquelle le circuit peut fonctionner sans erreur. Il devient alors intéressant de changer de « profil » lorsque la couche logique de convolution change afin d'éviter d'être limité par la couche ayant les données les plus défavorables.

4.2.4.2 Architectures en *streaming* et moteur de calcul unique

Notre approche fonctionne dans les deux architectures présentées en section 3.1.5, le moteur de calcul unique et l'architecture fonctionnant en *streaming*.

L'utilisation de notre approche est simple dans le cas d'un moteur de calcul unique. En effet, l'accélérateur de convolution peut alors fonctionner dans son propre domaine d'horloge et des FIFO asynchrones doivent être utilisées pour ses interfaces.

Dans le cas d'une architecture fonctionnant en *streaming*, les couches de convolution peuvent être réparties sur différents matériels et peuvent donc fonctionner à différentes fréquences, en fonction du nombre d'horloges disponibles sur les cibles. Si le nombre d'horloges est insuffisant, il est possible de regrouper plusieurs accélérateurs de convolution dans un même domaine d'horloge. Dans ce cas, ce sera l'accélérateur le moins performant (provoquant des erreurs à la plus faible fréquence) qui contraindra la fréquence d'horloge utilisée. La fréquence de chaque domaine d'horloge peut être gérée indépendamment. Cependant, avec cette architecture, les différents taux d'*overclocking* peuvent mener à un déséquilibre du temps de calcul des différentes couches. Dans ce cas, les performances globales sont limitées par la fréquence la plus faible (voir section 4.2.4.1).

4.3 Évaluation

Cette section évalue les gains et la validité de notre approche. L'objectif est d'augmenter les performances et l'efficacité énergétique d'un accélérateur matériel et notre approche a des impacts à la fois négatifs et positifs sur les performances et la consommation énergétique. Il faut donc s'assurer que la méthode puisse être valide et définir dans quel cas

elle est valide.

Nous commençons par proposer un modèle de gain en performance d'un accélérateur utilisant notre approche. Nous traitons ensuite de la correction de la détection d'erreur, puis nous comparons notre approche avec une alternative utilisant l'ABFT pour produit de matrice. Nous présentons ensuite la plateforme expérimentale ayant servi au reste de l'évaluation, puis nous présentons nos résultats expérimentaux sur l'amélioration des performances et de l'efficacité énergétique. Nous traitons de la validité de l'approche du point de vue des ressources matérielles utilisées. Enfin, nous présentons nos observations sur les erreurs et le risque de faux négatifs.

4.3.1 Gains en performance et pénalités dues aux erreurs

L'approche proposée est valide pour augmenter les performances si le gain « brut » dû à l'*overclocking* dépasse les pénalités dues aux erreurs. Le gain est égal au taux d'*overclocking* (si ce sont bien les calculs qui limitent la performance, voir section 4.2.1). Chaque erreur [3] implique une pénalité sur les performances parce qu'il faut recalculer la tuile erronée. La pénalité d'une mauvaise spéculation dépend de plusieurs facteurs, dont :

— le taux d'*overclocking* (fréquence moyenne d'*overclocking* normalisée à la fréquence nominale du circuit), noté θ ;
— le taux d'erreur (probabilité qu'une tuile soit erronée), noté e ;
— l'architecture choisie (moteur de calcul unique ou architecture fonctionnant en *streaming*, voir section 3.1.5).

Cette section propose des modèles analytiques simples du gain « net » en débit, noté G, en fonction de ces facteurs et exprimé par rapport au débit obtenu sans *overclocking*. Cette métrique de performance est choisie parce qu'elle correspond au gain obtenu pour plusieurs utilisations concrètes : gain en temps de calcul pour l'inférence sur corpus de données, gains en ressources matérielles économisées pour un centre de données, inverse de la latence moyenne du calcul d'une tuile, etc. Dans certains contextes, d'autres métriques peuvent être importantes ; par exemple la latence maximale du calcul d'une tuile dans un contexte temps réel. Cependant, pour cet exemple, cette métrique est peu pertinente parce que cette latence peut être bornée à deux fois la latence sans *overclocking* (voir ci-dessous).

Ces modèles permettent de déterminer dans quelles conditions l'approche est valide

3. On considère ici les erreurs niveau tuile, le niveau de granularité de la détection d'erreur, car plusieurs erreurs dans la même tuile n'impliquent qu'une seule pénalité.

pour augmenter les performances quand le gain dépasse le surcoût. On fait par ailleurs les hypothèses suivantes :

— le second calcul des tuiles erronées est effectué à la fréquence nominale du circuit afin d'éviter les erreurs (voir section 4.2.4.1 pour les autres possibilités), chaque calcul est donc exécuté au maximum deux fois ;

— le délai pour changer la fréquence de l'accélérateur est négligeable.

Pour un moteur de calcul unique, lorsqu'une tuile résultat est erronée, les tuiles opérandes correspondantes sont à nouveau soumises à l'accélérateur pour que la tuile soit exécutée à nouveau. Les tuiles peuvent être exécutées dans n'importe quel ordre parce que toutes les tuiles sont indépendantes. Le *macro-pipeline* n'a donc pas besoin d'être bloqué. Les tuiles opérandes peuvent donc simplement être replacées dans la file d'attente de calculs à effectuer. Le débit *utile* est réduit puisqu'une tuile aura été exécutée deux fois dont une à une fréquence plus faible. Cependant, il n'y a pas de perte de temps supplémentaire : l'accélérateur est toujours affecté à une tâche, il n'y a pas de « bulle ». Le gain est dans ce cas égal à : $G_{sce}(\theta, e) = \frac{\theta}{1+e\theta}$ et l'approche est valide si $e \leqslant \frac{\theta-1}{\theta}$.

Pour une architecture fonctionnant en *streaming*, le surcoût dépend des caractéristiques de l'architecture. S'il existe des mémoires tampons entre les couches, par exemple sur les interfaces entre les FPGA, le blocage complet du pipeline peut être évité. En effet, les tampons fournissent des données à calculer à la place de la tuile supprimée. Le *pipeline* continue alors les calculs sur les données valides. Dans le pire cas, le *pipeline* complet doit être redémarré, en éliminant tous les résultats intermédiaires. En notant s le nombre d'étages de l'architecture (par exemple, $s = 7$ pour l'architecture de la figure 3.8 page 89), le gain est égal à [4] : $G_{sa}(\theta, e, s) = \frac{\theta}{1+se\theta}$ et l'approche est valide si $e \leqslant \frac{\theta-1}{s\theta}$.

La figure 4.8 illustre le seuil de validité entre le gain dû à l'*overclocking* et les pénalités dues aux erreurs pour plusieurs valeurs de s selon ce modèle. Les points de fonctionnement *au-dessus* de la courbe ne sont pas valides, tandis que ceux *en dessous* le sont : les pertes sont plus que compensées par les gains. Par exemple, avec un taux d'*overclocking* de 25 % sur une architecture fonctionnant en *streaming* à 5 étages, il faut un taux d'erreur de 4 % pour annuler le gain. Pour ce taux d'*overclocking*, l'approche est donc valide pour augmenter les performances si le taux d'erreur est inférieur à 4 %. En outre, plus le taux d'erreur est faible, plus le gain est important.

On observe que, selon ce modèle, le taux d'erreur doit être relativement élevé pour annuler les gains de l'*overclocking*. Or, le taux d'erreur dépend de la stratégie de gestion

4. G_{sce} est un cas particulier de G_{sa} avec $s = 1$.

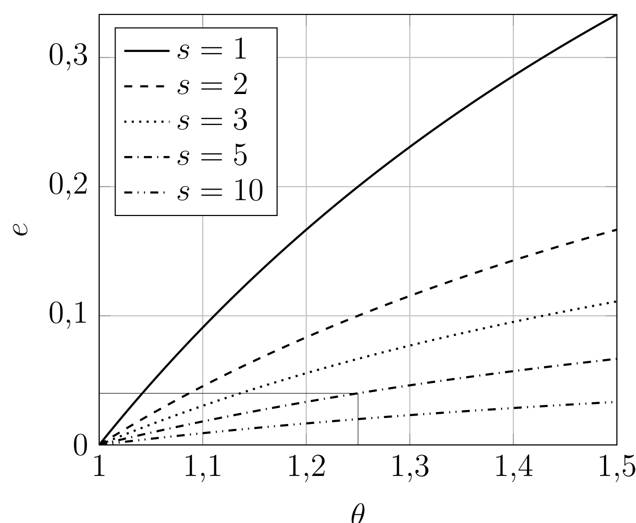


FIGURE 4.8 – Seuils de validité entre le gain dû à l'*overclocking* (taux θ) et les pénalités dues aux erreurs (taux e), en fonction de du nombre d'étages du *macro-pipelines*

de la fréquence (voir section 4.2.4.1). En choisissant une stratégie très conservative (par exemple en augmentant légèrement la fréquence si on n'observe pas d'erreur pendant une longue période et en la diminuant après une erreur), il est possible de rendre le taux d'erreur très faible. Dans ce cas, on assure que les performances sont améliorées et non détériorées, même dans le cas où des erreurs surviennent à un taux d'*overclocking* faible.

Dans le cas où l'application peut tolérer quelques tuiles erronées (qui pourraient conduire, par exemple, à des erreurs de classification), par exemple dans un contexte non critique, un faible taux d'erreur peut être acceptable. Certaines applications peuvent être capables de se dispenser de quelques résultats (par exemple sur un traitement vidéo en temps réel, certaines images peuvent parfois être ignorées sans impacter l'application). Ainsi, même si les erreurs ne sont pas tolérées, les données erronées peuvent simplement être ignorées. Dans ces deux cas, la détection d'erreur serait donc utilisée seulement pour réguler la fréquence dans le but de garder un taux d'erreur acceptable et les erreurs n'auraient pas besoin d'être corrigées. Notre approche n'entraînerait alors aucune latence supplémentaire et le gain en débit serait proportionnel au taux d'*overclocking*.

4.3.2 Correction de la détection d'erreur

Comme tout mécanisme de détection d'erreur, le mécanisme de détection d'erreur proposé peut dans certains cas avoir des faux positifs (détection d'une erreur alors qu'aucune

<table>
<tr><th colspan="2"></th><th colspan="3">Erreurs dans la couche de convolution</th></tr>
<tr><th colspan="2"></th><th>aucune</th><th>unique</th><th>multiple</th></tr>
<tr><td rowspan="3">Erreurs dans les sommes de contrôle</td><td>aucune</td><td>pas d’erreur</td><td>détection à 100%</td><td>faux négatif possible</td></tr>
<tr><td>unique</td><td>100% faux positif</td><td colspan="2" rowspan="2">faux négatif possible</td></tr>
<tr><td>multiple</td><td>faux positif possible</td></tr>
</table>

TABLE 4.2 – Résultats possibles de la détection d’erreur pour tous les scénarios : aucune erreur, erreur unique ou erreurs multiples dans les sommes de contrôle ou la sortie de la couche de convolution. Chaque résultat est soit une exécution sans erreur, soit une détection d’erreur, soit un faux positif, soit un faux négatif.

erreur ne s’est produite) ou des faux négatifs (pas de détection d’une erreur qui s’est produite).

Il existe plusieurs scénarios en fonction du nombre d’erreurs temporelles qui surviennent et de leur impact. Une erreur temporelle peut avoir un impact sur n’importe quel calcul du circuit : la convolution elle-même ou les calculs de sommes de contrôle. Les scénarios sont résumés dans la table 4.2. Les deux cas de base, lorsqu’aucune erreur ne se produit ou lorsqu’une seule erreur se produit dans la couche de convolution, sont mis en évidence. Dans ces cas, il n’y a pas de faux positif ou de faux négatif et l’erreur est nécessairement détectée. Dans les autres cas, le résultat peut potentiellement être un faux positif ou un faux négatif. Un faux positif n’affecte que les performances : la tuile doit être exécutée à nouveau même si son résultat était correct. En revanche, un faux négatif est problématique, car les erreurs ne sont pas détectées.

Nous proposons une explication montrant pourquoi un faible taux de faux négatifs peut être attendu dans notre cas en utilisant un modèle d’erreur simpliste. Les faux négatifs sont possibles si de multiples erreurs [5] se produisent. En effet, les erreurs temporelles survenant dans la couche de convolution modifient la somme de contrôle de sortie. Celle-ci étant une réduction des résultats de la convolution, des erreurs peuvent s’annuler si leurs effets sur la somme de contrôle sont opposés et la somme de contrôle de sortie sera identique

5. Une erreur correspond ici à une sortie de la convolution qui est erronée, possiblement de plusieurs bits. De multiples erreurs peuvent donc survenir dans une même tuile.

Entrées (bits)	Poids (bits)	Sommes de contrôle (bits)
16	16	54
16	8	46
16	4	42
16	2	40
16	1	39
8	8	38
4	4	30
1	1	24

TABLE 4.3 – Largeurs de mots des données et des sommes de contrôle utilisées dans les résultats expérimentaux (voir section 4.3.4). Les dimensions de cette couche de convolution sont $N = 192$, $M = 128$, $R = C = 13$, $K = 3$ et $S = 1$, mais les sommes de contrôle sont calculées sur des tuiles de plus petite taille. Les tuiles ont une taille de 32 sur la dimension N et une taille de 64 pour la dimension M.

(par exemple, si la valeur d'une sortie a une erreur numérique de +8, cette erreur peut être masquée par une erreur sur une autre sortie dont l'erreur numérique serait de -8). Intuitivement, la probabilité qu'une telle annulation se produise est liée à la précision (largeur de mot) utilisée pour le calcul de la somme de contrôle. Comme la somme de contrôle est calculée de manière exacte, sa largeur de mot b dépend des types de données d'entrée et du nombre d'accumulations. Pour une somme de contrôle sur une convolution complète, on a $b = D + W + \lceil \log_2(NK^2) + \log_2(RCM) \rceil$, où D est la largeur de mot des données d'entrée, W est la largeur de mot des données de poids et N, K, R, C et M sont les paramètres de convolution tels que définis dans la section 3.1.2. La table 4.3 résume les largeurs de mots des sommes de contrôle pour les modèles utilisés dans notre évaluation. Plus la largeur de la somme de contrôle est grande, plus les chances d'obtenir un faux négatif sont faibles. En supposant un modèle simple dans lequel une erreur provoque l'inversion d'un seul bit (comme dans l'exemple ci-dessus) avec une probabilité uniforme, les chances que plus d'une erreur s'annulent sont de $\frac{1}{2^b}$. La probabilité d'un faux négatif dû à plusieurs erreurs est donc, d'après ce modèle, faible. Par exemple, pour le cas le plus défavorable de la table (24 bits), cette probabilité est d'environ six sur cent millions.

Des erreurs peuvent également se produire dans le circuit de calcul des sommes de contrôle. Si les erreurs se produisent uniquement dans ce circuit, un faux positif est très probable (sauf si les erreurs s'annulent entre elles, dans ce cas il n'y a plus de problème). Si des erreurs se produisent à la fois dans le calcul de convolution et dans le calcul des

sommes de contrôle, un faux négatif est possible si les erreurs des sommes de contrôle permettent de masquer l'erreur ou les erreurs de la convolution.

Cependant, les erreurs temporelles devraient moins survenir dans les circuits de calcul des sommes de contrôle que dans le circuit du noyau de convolution. Cela est dû au fait que la mise en œuvre matérielle des sommes de contrôle est beaucoup plus simple que la mise en œuvre de la convolution en raison des niveaux de parallélisme plus faibles, d'un circuit plus petit et de contraintes temporelles plus faibles.

4.3.3 Convolution convertie en multiplication de matrice

Comme montré section 3.1.3, les convolutions des CNN peuvent être calculées grâce à une opération de multiplication matrice-matrice. Or, il existe déjà un mécanisme d'ABFT pour cette opération (voir 2.2.4.6). Il pourrait donc s'avérer intéressant d'utiliser une convolution implémentée comme une multiplication de matrice et ce mécanisme d'ABFT plutôt que notre approche afin de réutiliser des briques logicielles et matérielles existantes.

Calculer la convolution comme un produit de matrice naïf demande le même nombre d'opérations qu'une convolution naïve, mais les opérandes contiennent des données redondantes. Les matrices opérandes ont pour dimensions $RC \times NK^2$ et $NK^2 \times M$ et calculer la somme de contrôle d'entrée nécessite $NK^2(M + RC - 1) - 1$ sommes et NK^2 produits. La matrice résultat étant identique (excepté la forme de la matrice) à celle calculée avec une convolution classique, calculer la somme de contrôle de sortie nécessite le même nombre d'opérations : $MRC - 1$ sommes. Le calcul des vecteurs sommes de l'opérande gauche dans l'ABFT pour produit de matrice correspond aux sommes des cartes de fonction (notées $X_{n,i,j}$), tandis que le calcul des vecteurs sommes de l'opérande droit correspond aux sommes des poids sur la dimension M. La somme de contrôle d'entrée est le produit scalaire de ces deux vecteurs de taille NK^2. On retrouve donc bien le calcul de la somme de contrôle d'entrée présenté section 4.1.4 à une exception près : la réutilisation des sommes dans le calcul des valeurs de X. Sans cette dernière simplification, le calcul de la somme de contrôle d'entrée est plus coûteux. Conjuguer ces deux briques existantes n'est donc pas équivalent à notre approche.

Il serait toutefois possible d'effectuer cette simplification en sachant que le produit de matrice traite des matrices opérandes pour une convolution, qui ont une structure particulière et redondante et non des matrices ordinaires. Cette reformulation reviendrait cependant exactement à l'invariant de détection d'erreur que nous proposons.

4.3.4 Plateforme expérimentale

Nous utilisons une plateforme expérimentale afin de démontrer la validité de notre approche. L'objectif n'est pas de prouver que notre méthode fonctionne dans toutes les conditions possibles, mais de montrer expérimentalement qu'on peut obtenir des résultats satisfaisants et qu'elle est donc valide. Les expériences présentées dans cette section se basent sur une plateforme de développement Xilinx ZC706 contenant principalement un FPGA xc7z045ffg900-2. Les synthèses logiques sont réalisées avec la suite d'outils Xilinx SDx, Vivado HLS et Vivado en version 2018.2. Pour cette plateforme, nous utilisons les tailles de tuiles et les niveaux de déroulage suivants (voir section 4.2.1 pour leurs définitions) :

— $T_R = T_C = 13$;
— $T_M = 64$;
— $T_N = 32$;
— $U_M = 16$;
— $U_N = 32$.

Les synthèses ciblent la cinquième couche de convolution du modèle AlexNet (Krizhevsky, Sutskever et Geoffrey E Hinton 2012) dont les paramètres sont :

— $R = C = 13$;
— $M = 128$;
— $N = 192$;
— $K = 3$;
— $S = 1$.

Les mesures de puissance et les mesures de temps de calcul sur FPGA sont réalisées grâce à un analyseur de puissance Agilent N6705A. L'analyseur mesure la tension et le courant fournis à toute la carte de développement hors alimentation.

Les données d'entrées et de poids proviennent des opérandes de la cinquième couche d'AlexNet utilisant un modèle pré-entraîné [6]. Les opérandes sont extraits lors de l'exécution du réseau de neurones par tiny-dnn [7] sur le jeu de données ImageNet. L'exécution est réalisée en arithmétique à virgule flottante au format *binary32* et les données sont ensuite quantifiées au format à virgule fixe de l'accélérateur.

6. `https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet`
7. `https://github.com/tiny-dnn/tiny-dnn`

4.3.5 Amélioration des performances

Nous nous intéressons en premier à l'amélioration des performances que permet notre approche. Comme montré section 4.3.1, le taux d'*overclocking* doit être suffisant pour compenser les pénalités dues aux erreurs. Pour déterminer si notre approche permet bien d'augmenter les performances sur notre plateforme expérimentale, nous devons donc vérifier qu'il existe au moins un point de fonctionnement valide.

Pour cela, nous mesurons le taux maximal d'*overclocking* applicable sans observer d'erreur. Ne pas observer d'erreur à une fréquence donnée n'implique pas que le taux d'erreur est nul : lorsque le taux d'erreur est extrêmement faible, il est difficile à mesurer parce que les erreurs sont alors des évènements rares. Cependant, plus on augmente le nombre de calculs sans observer d'erreurs, plus le taux d'erreur probable tend vers zéro.

La figure 4.9 montre les mesures de débit en milliards d'opérations par seconde pour plusieurs accélérateurs avec ou sans notre approche. Pour chaque accélérateur, le débit de référence est le débit sans *overclocking* à la fréquence maximale donnée par les outils de synthèse. Le second débit est le débit mesuré à la fréquence maximale à laquelle on n'observe plus d'erreur. Les opérations comptabilisées sont uniquement les opérations *utiles*, c'est-à-dire les opérations de multiplication et d'addition uniquement pour la convolution. En particulier, les calculs de sommes de contrôle ne sont pas comptabilisés. Notre approche permet d'obtenir un gain en performance de 54 % en moyenne sur ces accélérateurs, jusqu'à 67 % pour l'accélérateur 16×16. Les accélérateurs avec convolution binaire (16×1 et 1×1) montrent des résultats moins bons (respectivement 21 % et 28 %).

4.3.6 Amélioration de l'efficacité énergétique

Comme pour les performances, notre approche permet d'augmenter l'efficacité énergétique si les gains compensent le surcoût en consommation d'énergétique. Les gains sont obtenus grâce à la réduction de l'énergie utilisée pour calculer une tuile malgré l'augmentation de la puissance instantanée (voir section 2.1.2). Les surcoûts sont dûs aux erreurs (les tuiles exécutées deux fois nécessitent plus d'énergie) et, dans une moindre mesure, à la détection d'erreur (voir section suivante).

Dans les conditions présentées précédemment, nous mesurons le temps de calcul ainsi que la puissance instantanée fournie à la plateforme de développement grâce à l'analyseur de puissance. Les opérations comptabilisées correspondent toujours aux opérations *utiles*, mais l'énergie comptabilisée est l'énergie consommée par la plateforme de développement

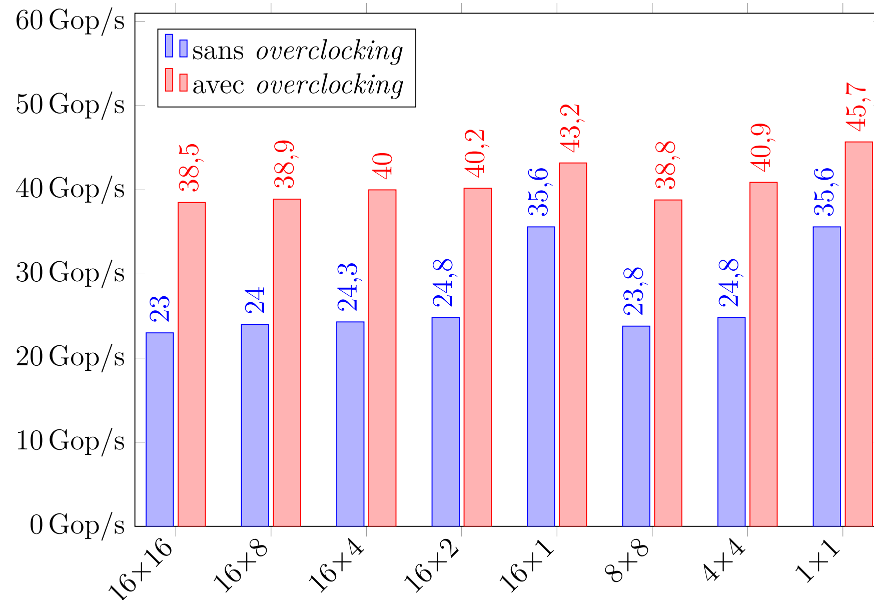


FIGURE 4.9 – Débits en milliards d'opérations par seconde pour plusieurs accélérateurs sans notre approche (en bleu) ou avec (en rouge). Les paramètres des accélérateurs sont la taille des mots notés $D \times W$ où D est la taille des mots d'entrée et W est la taille des mots de poids.

(hors alimentation), y compris la détection d'erreur, les mémoires, les périphériques, etc. La figure 4.10 montre les résultats d'efficacité énergétique en milliards d'opérations par joule. Notre approche permet un gain moyen de 46 %. Dans le meilleur cas, pour l'accélérateur 4×4, le gain est de 58 %. Tout comme pour l'amélioration des performances, les accélérateurs binaires ont un gain plus faible (19 % et 26 %).

4.3.7 Gains en performance, en efficacité énergétique et surcoût dû à la détection d'erreur

L'augmentation des performances et de l'efficacité énergétique de l'accélérateur de convolution est possible grâce à l'*overclocking* et la détection d'erreur. Or, le mécanisme de détection d'erreur utilise lui-même des ressources matérielles du FPGA et de l'énergie pour réaliser les calculs nécessaires (même s'il n'y a pas d'erreur). Il faut donc que le mécanisme de détection d'erreur utilise le moins de ressources et le moins d'énergie possible afin de maximiser les gains.

Par conséquent, il existe un point de bascule au-delà duquel l'approche n'est plus va-

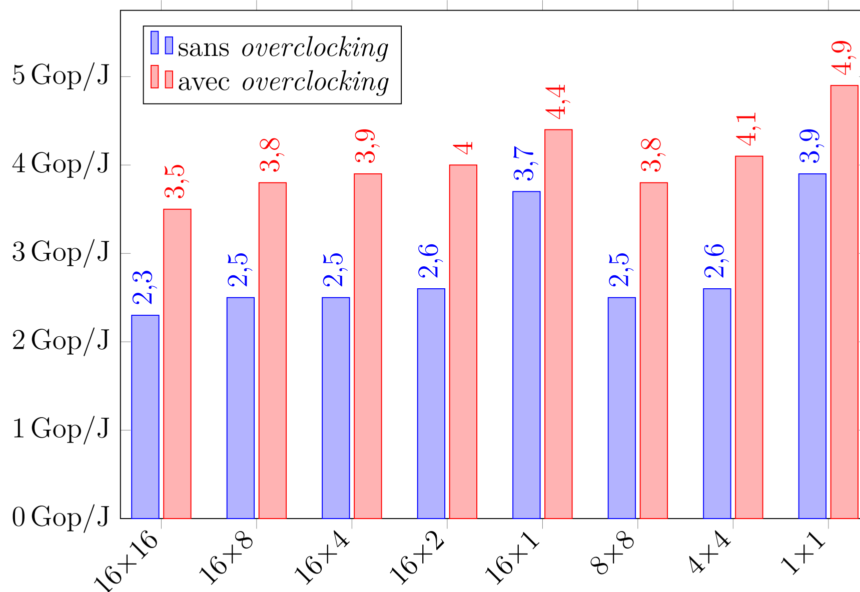
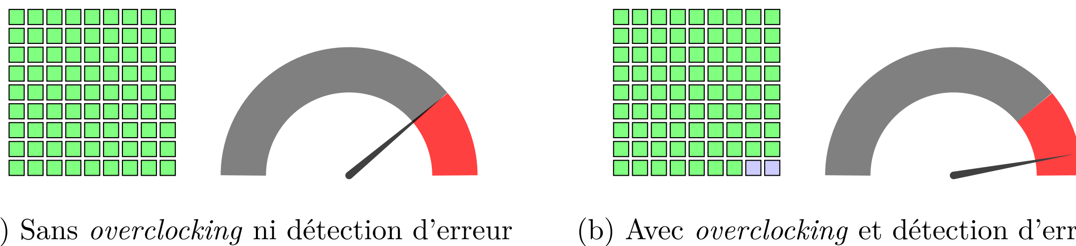


FIGURE 4.10 – Débits en milliards d'opérations par joule pour plusieurs accélérateurs sans notre approche (en bleu) ou avec (en rouge)

lide pour augmenter les performances et un point de bascule au-delà duquel l'approche n'est plus valide pour améliorer l'efficacité énergétique (même sans la présence d'erreur). Contrairement à la section 4.3.1 qui présente l'équilibre entre le gain dû à l'*overclocking* et les pénalités dues aux erreurs, nous comparons ici ce gain et le surcoût dû à la détection d'erreur. Si la détection d'erreur est trop coûteuse, il peut alors être plus intéressant d'utiliser un autre mécanisme de détection d'erreur, voire d'allouer les ressources directement au calcul de la convolution sans faire d'*overclocking*.

La figure 4.12 montre les ressources utilisées pour des accélérateurs avec et sans détection d'erreur par ABFT. On peut déduire de ces résultats le surcoût nécessaire pour implémenter l'intégralité du mécanisme de détection d'erreur sur notre plateforme expérimentale : le calcul des deux sommes de contrôle, les mémoires et le contrôle. On observe que le surcoût est faible pour tous les accélérateurs avec une moyenne de 1257 *slices* pour des valeurs entre 831 et 1633 *slices*. De plus, la détection d'erreur n'utilise aucun BRAM ou DSP qui sont les ressources les plus critiques pour implémenter la convolution : mémoires et multiplications-accumulations. Ces ressources ne peuvent donc être allouées qu'au calcul de convolution [8]. Par ailleurs, les ressources *slices* sont relativement peu utilisées par

8. La discussion sur l'allocation de ressources entre la convolution et la détection d'erreur serait donc plus pertinente pour un ASIC pour lequel les deux composantes utilisent les mêmes budgets de sur-



(a) Sans *overclocking* ni détection d'erreur (b) Avec *overclocking* et détection d'erreur

FIGURE 4.11 – Allouer une partie des ressources matérielles (représentées par des carrés) à la détection d'erreur (en bleu) plutôt qu'au calcul principal (en vert) réduit les performances, mais permet d'augmenter la fréquence au-delà de la fréquence nominale du circuit, ce qui permet d'augmenter les performances. L'allocation de l'énergie et l'augmentation de l'efficacité énergétique suit le même principe.

l'accélérateur de convolution [9], sauf dans le cas de la convolution binaire. Le surcoût en *slices* est donc virtuellement nul parce que les ressources présentes sont sous-utilisées.

Parce qu'elle est basée sur l'algorithme de convolution, la détection d'erreur fonctionne indifféremment des paramètres de convolution ou des implémentations et le surcoût sera du même ordre de grandeur quels que soient les paramètres. Par exemple, en choisissant $K = 5$ (dernier graphique de la figure 4.12), on obtient également un léger surcoût en *slices*.

La figure 4.13 montre une comparaison du surcoût de notre approche pour l'accélérateur 16×16 bits et des différents surcoûts d'un mécanisme de détection d'erreur basé sur l'arithmétique modulaire (voir section 2.2.4.3). Dans cette approche, seules les opérations de multiplications-accumulations sont protégées grâce à des calculs redondants dans une base modulaire (3, 7 et 15). On voit que notre mécanisme de détection d'erreur nécessite moins de ressources matérielles que toutes ces solutions. De plus, la couverture d'erreurs de ces approches est assez faible : par exemple, la détection d'erreur sur la base modulo 3 ne protège que 66 % des erreurs. En outre, protéger les autres calculs, comme le contrôle (compteurs d'itération, machine à états, etc.), nécessiterait plus de ressources, voire ne serait pas possible en utilisant cette méthode.

Nous n'avons pas de mesures du surplus de consommation énergétique dû à la detection d'erreur. Cependant, vu le faible nombre de ressources utilisées, on peut supposer que celui-ci soit très faible. De plus, contrairement à d'autres plateformes, la puissance

face (ressources) et puissance.

9. Il faut noter qu'une utilisation de 100 % est quasiment impossible parce que le placement et le routage deviendraient trop contraints. De plus, un taux d'utilisation trop élevé fera baisser la fréquence maximale du circuit.

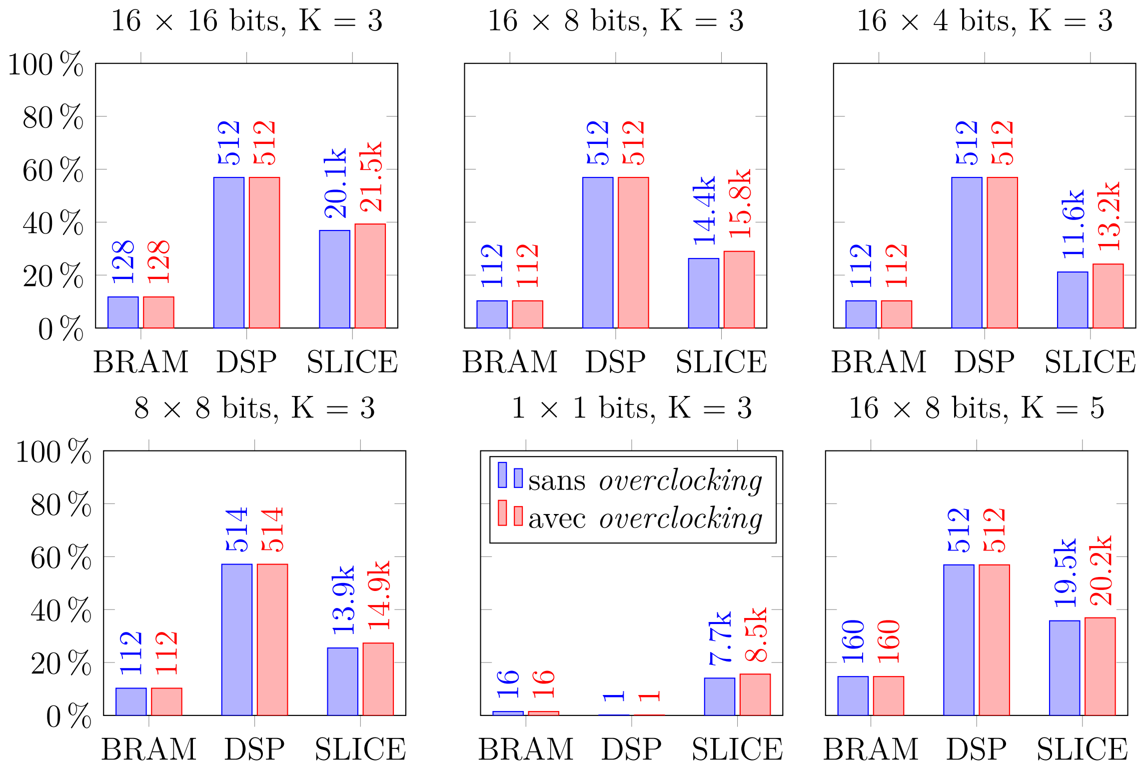


FIGURE 4.12 – Comparaison des ressources utilisées pour des accélérateurs de convolution avec et sans notre approche. Les paramètres des convolutions explorés sont la taille des filtres, K, et la taille des mots notés $D \times W$ où D est la taille des entrées et W est la taille des poids.

statique est significative pour un FPGA (voir section 1.1 sur les puissances statique et dynamique). L'utilisation de ces ressources supplémentaires augmentent donc la puissance moins que proportionnellement aux ressources utilisées. Il faudrait donc un gain en efficacité énergétique grâce à l'*overclocking* très faible pour que l'approche ne soit pas valide.

4.3.8 Erreurs observées

Pour observer les erreurs temporelles, nous avons collecté les erreurs dans la sortie de la convolution pour l'accélérateur 16 × 16 pour différentes fréquences avec un pas de 0,25 MHz. La fréquence maximale de l'accélérateur d'après les outils de synthèse est de 141 MHz. La figure 4.14 présentent les taux d'erreur en fonction de la fréquence. Les taux d'erreur sont reportés pour chaque bit de la sortie de convolution sur 16 bits. Les données d'entrées ainsi que les poids étaient tirés aléatoirement de manière uniforme. Ces observations permettent de tirer plusieurs conclusions, présentées ci-après. Cepen-

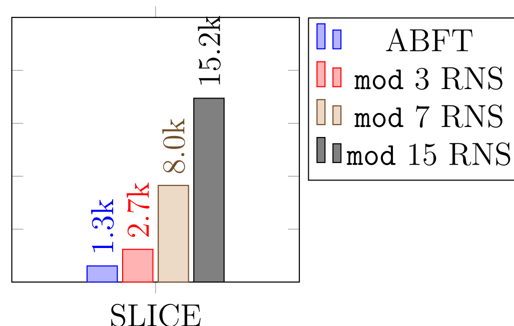


FIGURE 4.13 – Comparaison du surcoût de notre approche (ABFT) pour l'accélérateur 16×16 bits avec $K = 3$ et des surcoûts d'une approche basée sur l'arithmétique modulaire (RNS)

dant, ces conclusions ne sont pas forcément généralisables, pour d'autres stimuli, d'autres accélérateurs, ou tout autre changement de conditions.

On peut remarquer qu'il y a deux principales plages de fréquences disjointes : une plage sans erreurs observées de 141 MHz à 235,75 MHz (non représentée dans la figure) et une plage avec erreur à partir de 236 MHz (on n'observe pas d'erreur à 236,25 MHz mais cela peut s'expliquer par un nombre d'observations insuffisant). Il faut aussi noter qu'un taux d'erreur observé nul ne garantit pas que la fréquence correspondante est sûre : le taux d'erreur réel peut être trop faible pour avoir été mesuré, ou les stimuli peuvent ne pas avoir testé le pire cas.

On remarque aussi que plus la fréquence augmente, plus le taux d'erreur augmente, à quelques exceptions près. Ces observations peuvent permettre de choisir la manière de gérer dynamiquement la fréquence (voir section 4.2.4.1). En l'occurrence, il semble raisonnable de penser qu'une recherche dichotomique de la fréquence maximale à laquelle on n'observe pas d'erreur peut fonctionner, ou encore que les premières fréquences auxquelles on observe des erreurs correspondent à un faible taux d'erreur (et non à des tuiles entièrement erronées).

On remarque que les bits de poids faibles ont plus de chances d'être erronés que les bits de poids forts. Cela peut paraître contre-intuitif, mais cet effet pourrait être expliqué par la phase de routage, lors de la synthèse logique, qui essaie d'équilibrer les différents chemins du circuit. Le routage, dans un FPGA, est devenu une partie importante du délai de propagation des signaux et impacte le comportement temporel d'un accélérateur. De plus, les multiplications-accumulations utilisent les DSP et ceux-ci sont des boîtes noires pour lesquelles le comportement temporel ne peut pas être facilement analysé.

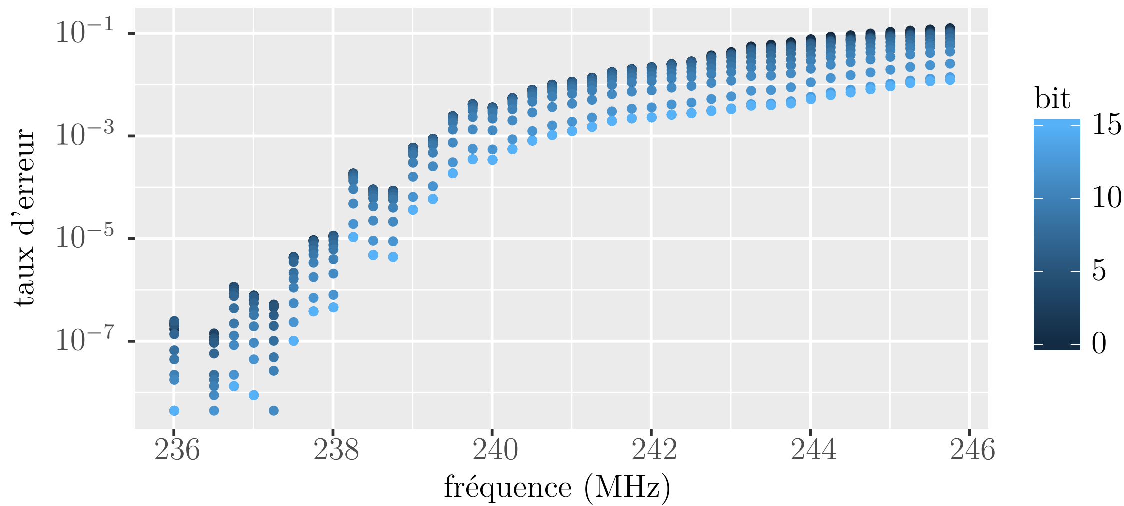


FIGURE 4.14 – Taux d'erreur par bit des sorties de convolution en fonction de la fréquence d'*overclocking*. Le taux d'erreur est considéré indépendamment pour chacun des 16 bits.

Cette observation montre que les techniques de détection d'erreur comme Razor (voir section 2.2.4.4), basées sur une selection des chemins à protéger sont risquées si l'analyse temporelle du circuit ne correspond pas à la réalité ou si le concepteur prend une mauvaise décision (ici, ne pas protéger les poids faibles).

En particulier, aux fréquences les plus faibles pour lesquelles on observe des erreurs, presque toutes les erreurs impactent les bits de poids faible. La plupart des erreurs ont donc une faible amplitude et ont moins de chance de provoquer une erreur au niveau application (par exemple, une erreur de classification) parce que les réseaux de neurones ne sont généralement pas sensibles au bruit (voir section 3.3). Cette observation pourrait être utilisée pour prendre la décision de tolérer quelques erreurs sans avoir à recalculer les tuiles erronées. De plus, cette particularité peut être utilisée pour détecter les fautes temporelles avant qu'elles n'impactent la sortie de la convolution, afin d'éviter toute erreur temporelle. En effet, la convolution est calculée de manière exacte, sans troncature ni risque de dépassement, afin, entre autres, de permettre le calcul des sommes de contrôle. En revanche, seuls certains bits des résultats exacts sont réellement utilisés pour la sortie, dans notre cas 16 bits. En considérant que les 16 bits bits de poids forts sont conservés, il y a au moins 16 bits qui ne sont pas conservés. Si une faute temporelle modifie l'un des bits non conservé, il n'y aura donc pas d'erreur temporelle, mais la comparaison des sommes de contrôle permettra de détecter la faute. Lorsqu'on observe une différence dans les bits

de poids faible des sommes de contrôle et en faisant l'hypothèse qu'une seule erreur est survenue [10], on peut conclure qu'il n'y a pas eu d'erreur temporelle dans la sortie de la convolution. Il n'y a donc pas besoin d'exécuter à nouveau la tuile, mais cette information peut être utilisée pour contrôler la fréquence et éviter de futures fautes.

4.3.9 Faux négatifs

Comme montré section 4.3.2, il y a un risque de faux positifs et de faux négatifs dans la détection d'erreur qui dépend de plusieurs facteurs, dont le taux d'*overclocking* et la taille des données et opérateurs. Nous avons utilisé les accélérateurs 8×8 et 4×4 pour mesurer les taux de faux positifs et faux négatifs observés.

Nous avons analysé le résultat de plus de 700 000 tuiles pour 21 fréquences choisies pour correspondre à un taux d'erreur (tuile erronée) de 0 % à 10 %. Le taux de 10 % est délibérément élevé (il ne serait a priori pas atteignable par une gestion correcte de la fréquence) afin de provoquer un nombre important d'erreur (à l'intérieur d'une tuile) et des erreurs de grande amplitude.

Nous n'avons observé aucun faux négatif pour l'accélérateur 8×8, mais quelques-uns pour l'accélérateur 4×4, ce qui est logique parce que plus les mots sont petits, plus le risque est élevé. Le taux de faux négatifs est au plus 0,4 % des tuiles incorrectes lorsque de nombreuses erreurs sont observées. Rapporté au nombre total de tuiles calculées, le taux est de 0,06 %. Lors d'une utilisation normale des accélérateurs, un taux de faux négatifs faible multiplié par un taux d'erreur très faible donne un risque total très faible. De plus, si l'on prend en considération que les rares erreurs ne provoquent pas forcément d'erreur au niveau application (voir la section précédente), le risque qu'une erreur au niveau application ne soit pas détectée est extrêmement faible.

4.4 Discussion

Dans cette section, nous présentons les possibilités et les limites de notre approche. Nous commençons par discuter son applicabilité à d'autres techniques que celles que nous avons utilisées : *undervolting*, types de données, algorithmes de convolution et autres architectures. Nous considérons ensuite son applicabilité à d'autres algorithmes que la

10. Autrement, deux erreurs impactant le même bit de poids fort peuvent s'annuler dans la somme de contrôle, comme pour un faux négatif.

convolution. Nous présentons ensuite les limites de notre approche, puis nous exposons une piste de travail pour utiliser celle-ci avec les arithmétiques à virgule flottante.

4.4.1 *Overclocking* et *undervolting*

Comme expliqué section 2.1.2, l'*overclocking* et l'*undervolting* ont des effets théoriquement équivalents. Nous avons choisi d'utiliser l'*overclocking* pour sa facilité de mise en œuvre sur FPGA. En effet, il est difficile de gérer finement la tension sur ces plateformes : il y a souvent un rail de puissance par type d'élément (BRAM, DSP, etc.) et non par zone ce qui empêche de gérer la tension spatialement, selon le composant (calcul de somme de contrôle, entrées/sorties, calcul de la convolution). Or, les entrées/sorties ne sont pas protégées par notre mécanisme de détection d'erreur, donc celles-ci ne doivent pas être sujettes aux erreurs temporelles [11]. Ces éléments du circuit ne doivent donc pas être soumis à l'*undervolting* ou l'*overclocking* et nous ne pouvons donc pas utiliser l'*undervolting* à cause de cette limitation. En revanche, la fréquence peut être gérée spatialement ce qui nous permet d'utiliser plusieurs fréquences pour le cœur de l'accélérateur et la gestion des entrées/sorties.

Des travaux existants explorent toutefois la faisabilité de l'*undervolting* sur FPGA (sans détection d'erreur). Par exemple, Salami et al. ont des résultats similaires aux nôtres avec un taux d'*undervolting* sans observer d'erreur de 33 % en moyenne (Salami et al. 2020). On peut noter que la différence entre la tension minimale sans erreur observée et la tension à laquelle l'accélérateur ne répond plus (plantage) n'est que de 30 mV en moyenne, soit 3,5 % de la fréquence nominale. Nous avons aussi pu observer des plantages, mais avec une fréquence nettement supérieure à la fréquence la plus faible à laquelle nous observions des erreurs. Un plantage peut être détecté et l'accélérateur rétabli, mais avec un impact plus important qu'une erreur de calcul.

L'utilisation de l'*undervolting* à la place de l'*overclocking* sur FPGA est donc possible, mais est plus complexe à mettre en œuvre et semble plus sujet aux plantages.

4.4.2 Représentation des données

La détection d'erreur étant basée sur le calcul et la comparaison de deux sommes de contrôle, le format des données importe peu tant que les sommes de contrôle peuvent

11. Cependant, il serait possible de détecter ces erreurs avec une approche complémentaire, comme les codes correcteurs d'erreur.

être calculées de manière exacte. Si ce n'est pas le cas, les sommes de contrôle pourraient être différentes même en l'absence d'erreur et l'invariant ne pourrait pas être vérifié correctement.

Par exemple, pour un format entier ou à virgule fixe, les calculs doivent être réalisés sur des opérateurs de taille suffisante et les résultats stockés dans des registres de taille adéquate (voir section 4.2.3). Ainsi, avec des entrées de 16 bits, le produit exact fait 32 bits et les sommes de contrôle sont plus larges selon le nombre d'accumulations.

Les opérations des arithmétiques à virgule flottante, y compris celles qui ont été développées spécifiquement pour l'apprentissage machine (voir section 3.1.4), ne respectent pas les règles usuelles d'associativité et de distributivité. Les sommes de contrôle ne peuvent donc pas, dans le cas général, être calculées de manière exacte. Il est toutefois possible, dans une certaine mesure, d'utiliser la tolérance aux fautes au niveau algorithmique en utilisant les méthodes décrites section 2.2.4.6 (seuil, estimation de l'erreur d'arrondi, etc.).

Nous avons exploré une nouvelle piste consistant en l'augmentation de la précision des calculs et du stockage des valeurs intermédiaires. Les valeurs sont alors des nombres à virgule fixe, mais les données traitées sont dans un format à virgule flottante. Ce principe d'opérations hybrides pour calculer une somme de produits est connu sous le nom d'accumulateur de Kulisch (KULISCH 2013). Avec des accumulateurs d'une précision suffisante, il devient possible de calculer les sommes de contrôle exactes, sans erreur d'arrondi et donc d'utiliser l'ABFT sur des données dans un format à virgule flottante. Ce travail n'est cependant pas finalisé et n'est donc pas présenté ici mais fera l'objet d'une publication future.

4.4.3 Variations algorithmiques de la convolution

La convolution étant de loin l'algorithme requérant la plus grande charge de travail dans les CNN, ses implémentations matérielles tirent parti de différentes optimisations (voir section 3.1.3). Ces méthodes permettent de réduire le nombre de calculs ou la mémoire nécessaires (élagage, transformation de Fourier, de Winograd), ou de réutiliser des accélérateurs déjà existants et performants (multiplication matrice-matrice).

Si une méthode ne change pas le résultat, qu'elle implémente toujours une convolution et non une approximation, la détection d'erreur au niveau algorithmique fonctionne toujours et donc notre approche fonctionne.

Cependant, il pourra y avoir des différences de résultats en performance et efficacité énergétique. Par exemple, l'élagage est pertinent si l'on utilise un accélérateur matériel

conçu pour tirer parti de la faible densité des données. L'accélérateur sera donc plus performant et utilisera moins d'énergie pour calculer une convolution « totale » (équivalent sans élagage). Le surcoût relatif de la détection d'erreur sera alors plus important.

Comme pour les arithmétiques à virgules flottantes, la détection d'erreur ne fonctionnera pas si les opérations ne sont pas associatives ou si la multiplication n'est pas distributive par rapport à l'addition. C'est le cas, par exemple, si un algorithme introduit des quantifications au cours de ses étapes intermédiaires. Il est toujours possible de ne pas effectuer les quantifications, mais le coût de l'implémentation sera alors plus élevé et peut donc rendre l'utilisation de cet algorithme moins intéressante.

4.4.4 Applicabilité à d'autres architectures

Nous avons utilisé une plateforme matérielle FPGA pour réaliser nos expériences, car celle-ci nous paraissait le choix le plus pertinent pour obtenir des gains importants en performance et en efficacité énergétique. Cependant, notre approche peut être généralisée à d'autres architectures comme les processeurs généralistes ou les GPU. Toutes les architectures de calcul fonctionnent à une certaine fréquence (ou plusieurs) qui garantit le bon fonctionnement du circuit. Ces fréquences nominales prennent toutes en compte une marge de sécurité et ces architectures ont donc un potentiel pour la spéculation temporelle.

Hari et al. ont utilisé notre mécanisme de détection d'erreur au niveau algorithmique pour convolution pour protéger une implémentation de CNN sur GPU contre les fautes matérielles (Hari et al. 2020). Leur CNN utilise des entrées au format entier sur 8 bits, ce qui permet de calculer les calculs intermédiaires et les sommes de contrôle de manière exacte en utilisant des entiers sur 32 ou 64 bits. Le surcoût en performance (temps d'exécution) pour quatre réseaux de neurones (VGG-16 sur des entrées de ImageNet ; VGG-16, ResNet-18 et ResNet-50 sur des entrées 1080×1920) va de 6 % à 23 %. Ils n'utilisent ni *overclocking* ni *undervolting*, mais Leng et al. ont pu mesurer que la marge de sécurité sur la tension atteint 11 %–22 % pour deux architectures de GPU, montrant le potentiel de notre approche sur ce type de plateforme (Leng et al. 2015).

Zhao et al. ont utilisé une méthode similaire à la nôtre pour détecter les erreurs de calcul d'un CNN sur un supercalculateur de 128 nœuds, chaque nœud étant équipé de deux processeurs Intel Xeon de 16 cœurs (Zhao et al. 2021). Ils utilisent la détection d'erreur afin de protéger les calculs des erreurs dues aux particules mais ne recourent pas à la spéculation temporelle. Ils parviennent à obtenir un surcoût en performance (temps d'exécution) de 1 %–4 % sans erreur et de 2 %–6 % en injectant des erreurs.

Pour utiliser la spéculation temporelle sur un processeur, comme pour un FPGA, il faut isoler les calculs de convolution qui sont protégés et peuvent utiliser la spéculation temporelle aux autres traitements (copies de données, réseau, etc.) qui ne sont pas protégés.

Certaines architectures peuvent aussi poser un problème de dissipation thermique parce que la puissance augmente linéairement avec la fréquence. Malgré tout, il reste possible d'utiliser la spéculation temporelle afin d'augmenter l'efficacité énergétique sans augmenter les performances (par rapport au niveau sans spéculation temporelle) avec l'*undervolting*.

4.4.5 Applicabilité à d'autres algorithmes et génération automatique

Nous avons montré, section 4.1, que les sommes de contrôle sont la réduction en somme de la matrice résultat de la convolution. En exprimant la somme de contrôle d'entrée directement à partir des opérandes puis en effectuant des simplifications (factorisations et réutilisations de sommes), nous diminuons la complexité du calcul. Conceptuellement, cette méthode de détection d'erreur revient donc à une duplication de calcul suivi d'une réduction, où l'un des calculs n'est pas simplifié afin d'obtenir le résultat tandis que l'autre est simplifié au maximum afin de réduire au plus le coût. La réduction et les simplifications appliquées sont donc fondamentaux pour l'ABFT.

Il est possible d'appliquer la même approche à d'autres algorithmes. Si le calcul est linéaire (sommes de produits), alors une réduction en somme fonctionne. D'autres types de réduction pourraient fonctionner pour d'autres types de calcul.

Gupta et Rajopadhye ont développé une technique d'analyse d'expressions qui utilisent l'opération de reduction : *simplifying reductions* (Gautam et Rajopadhye 2006). Cette analyse permet de trouver des sous-expressions réutilisées dans la reduction et d'exploiter celles-ci dans le but de réduire la complexité de l'expression. Cette technique utilise le modèle polyédrique, un formalisme permettant de représenter une classe de programmes et de raisonner sur ceux-ci (parallélisation, transformations, ordonnancement, génération de code, etc.). Grâce à elle, il est possible de retrouver automatiquement, ou semi-automatiquement, l'expression de la somme de contrôle d'entrée de l'algorithme à partir de l'expression de la réduction du résultat quand elle existe.

En utilisant d'autres outils du modèle polyédrique, il devient possible d'appliquer notre

approche automatiquement à n'importe quel programme qui s'y prête :

— analyse du programme et détection des parties pouvant être protégées par notre méthode ;
— génération des expressions de calcul des sommes de contrôle, simplifications de la somme de contrôle d'entrée ;
— génération de code pour la cible (processeur, FPGA, etc.) pour le cœur du calcul et pour les sommes de contrôle.

4.4.6 Limites

La première limite de notre approche est que la détection d'erreur ne protège que la partie algorithmique de l'accélérateur, la convolution dans notre cas. Elle ne protège pas les accès mémoires externes (par exemple, une erreur peut fausser le transfert d'un calcul pourtant correct), le contrôle externe (par exemple, le protocole de communication avec l'hôte) ou encore les autres couches de CNN. L'analyse et la définition du modèle d'erreur est donc un point clé pour utiliser notre approche. Si l'on considère que des erreurs peuvent survenir dans ces éléments, il faut utiliser d'autres techniques de tolérance aux fautes adaptées, comme les points de contrôle (*checkpointing*) (Bosilca et al. 2015), les chiens de garde (*watchdogs*), les codes correcteurs d'erreur, la duplication du contrôle (Fleming et Thomas 2016), etc. Même si le surcoût relatif de ces solutions peuvent être importants, le surcoût global sera faible si les composants protégés représentent une petite partie de l'accélérateur matériel.

Notre mécanisme de détection d'erreur ne fonctionne que pour la convolution. Or, dans les implémentations de CNN, la couche d'activation, ou couche non-linéaire est souvent fusionnée avec la convolution pour éviter les copies de données. De plus, les résultats exacts de la convolution doivent être utilisés pour calculer la somme de contrôle de sortie, les mots sont donc plus larges que la sortie tronquée de la convolution ou la sortie de la couche d'activation. Pour ne pas avoir de copie importante de données (et un impact sur les performances), le calcul de somme de contrôle devrait donc être réalisé juste après (temporellement et spatialement) la couche de convolution et avant la couche d'activation. Si les couches de convolution et d'activation sont fusionnées dans une implémentation existante, notre approche n'est pas directement applicable.

4.5 Conclusion

Nous avons présenté dans ce chapitre notre méthode de détection d'erreur au niveau algorithmique pour les convolutions des CNN et comment l'utiliser conjointement à la spéculation temporelle pour améliorer les performances et l'efficacité énergétique d'un accélérateur matériel. Notre prototype a permis de prouver la faisabilité de notre approche : celle-ci permet effectivement d'augmenter significativement le débit l'accélérateur matériel et de réduire l'énergie consommée par opération tout en évitant les erreurs au niveau applicatif. Nous avons montré que notre approche peut être généralisée autant que possible : pour d'autres implémentations de la convolution, pour d'autres architectures mais aussi pour d'autres algorithmes. Nous avons aussi donné les limites de l'approche : le champ de protection de la détection d'erreur et l'existence d'une méthode d'ABFT pour un algorithme donné.

Conclusion et travaux futurs

La fin de la loi empirique de Dennard dans les années 2000 a marqué la fin de la croissance exponentielle des performances et de l'efficacité énergétique des processeurs due à la réduction de la taille des transistors mais a permis l'essor des architectures spécialisées. Le travail sur les améliorations architecturales des processeurs et le développement des architectures spécialisées avait commencé bien avant, mais cet axe d'amélioration est aujourd'hui devenu incontournable. Les plateformes de calcul deviennent ainsi de plus en plus hétéroclites : processeurs multi-cœurs hétérogènes, GPGPU, ASIC spécialisés, etc.

Les accélérateurs matériels, sur ASIC ou FPGA, sont donc aujourd'hui souvent le meilleur choix pour exécuter de nombreux algorithmes intensifs en calculs. Dans le contexte actuel où la demande en capacité de calcul augmente, ils font donc l'objet d'un intérêt croissant aussi bien dans le monde académique que dans le monde industriel. Les recherches actuelles couvrent tous les niveaux d'abstraction : transistors, technologies de mémoire, organisations architecturales, analyses de circuits, optimisations algorithmiques automatiques, sécurité, outils de développement, etc. La spéculation temporelle est pertinente pour aider à répondre à cette demande parce qu'elle est une approche orthogonale à ces recherches et ces améliorations passées et futures. Elle est applicable à n'importe quelle génération de transistor, pour n'importe quel modèle d'exécution, etc. Quelles que soient les améliorations permises par ces recherches, la spéculation temporelle permettra d'obtenir un gain supplémentaire de performance et d'efficacité énergétique.

Nous avons montré que la spéculation temporelle doit être accompagnée d'un mécanisme de détection d'erreur. Celui-ci est indispensable, car les fréquences et les tensions auxquelles le circuit fonctionne sans erreur ne peuvent pas être connues à l'avance et changent au cours du temps. Il doit aussi avoir une bonne couverture d'erreur et doit avoir un surcoût (en ressources utilisées, en énergie et en impact sur les performances) le plus faible possible. Ces surcoûts doivent effectivement être suffisamment faibles pour que l'approche soit valable, mais aussi être les plus bas possibles pour maximiser les gains obtenus. Le mécanisme de détection d'erreur est donc un point clé pour permettre à la spéculation temporelle d'améliorer les performances ou l'efficacité énergétique d'accélérateurs matériels. L'existence d'un tel mécanisme de détection d'erreur et ses coûts sont la

pierre angulaire de la spéculation temporelle.

Nous avons présenté un nouveau mécanisme de détection d'erreur pour les réseaux de neurones convolutifs permettant d'utiliser la spéculation temporelle pour cette application. Les CNN sont de plus en plus utilisés pour différentes tâches : reconnaissance ou comptage d'objets dans une image, classification d'image, traitement du langage naturel, aide au diagnostic médical, etc. Ils sont donc une application pertinente pour le développement d'architectures spécialisées dans le but d'augmenter leur niveau de performance face à la demande croissante et d'augmenter leur efficacité énergétique afin de faire baisser les coûts. Ils sont, de surcroît, tout aussi pertinents pour la spéculation temporelle et nous avons montré que celle-ci fonctionne et quels gains peuvent être attendus.

Cependant, l'amélioration de l'efficacité énergétique de ces plateformes n'est pas épargnée par le paradoxe de Jevons : celui-ci stipule qu'une amélioration rendant une technologie plus efficace en énergie peut avoir pour effet d'augmenter la consommation totale d'énergie au lieu de la baisser comme on pourrait s'y attendre. Or, notre travail permet d'augmenter l'efficacité énergétique d'accélérateurs matériels mais aussi leurs performances. L'effet rebond est donc immédiat : il suffit d'utiliser notre approche sur un accélérateur dans le but de faire plus de calculs en consommant plus d'énergie (bien que l'efficacité énergétique soit meilleure) plutôt que de faire autant de calculs plus rapidement en consommant moins d'énergie. Les premières lignes de cette thèse mentionnent que les inventions technologiques ont marqué l'histoire, mais celle-ci est aussi marquée par des évènements sociaux. Aussi, dans le contexte actuel, ces recherches devraient être utilisées avec discernement.

Perspectives de recherche

Nous avons, dans ce travail, vérifié la faisabilité de l'approche que nous proposons et nos expérimentations ont permis de mesurer les gains en performance et en efficacité énergétique pour quelques accélérateurs de convolution. Ces mesures sur un prototype suffisent à prouver la validité de l'approche mais ne répondent pas à toutes les problématiques soulevées par celle-ci. Par exemple, une bonne gestion de la fréquence en fonction du taux d'erreur semble essentiel pour maximiser les gains. L'intégration dans un réseau de neurones complet soulève aussi des problèmes : gestion de multiples fréquences, compromis entre le coût des mémoires tampons et la latence en cas d'erreur, répartition de la charge de travail, etc. Nous avons utilisé le débit pour mesurer les performances de

la convolution mais, pour certaines applications, la latence est une métrique plus importante. Dans le cas où la latence est le critère principal, il y a un compromis entre le coût de la détection d'erreur (plus faible pour une plus grande tuile) et la pénalité en latence en cas d'erreur (plus importante pour une plus grande tuile).

L'autre perspective de recherche est l'applicabilité à d'autres algorithmes. Notre approche peut en effet être appliquée à d'autres algorithmes pour lesquels un moyen peu coûteux de détecter les erreurs existe, en particulier les méthodes de tolérance aux fautes au niveau algorithmique. Il est aussi possible de développer de nouvelles méthodes pour d'autres algorithmes. Une solution permettant de détecter et protéger automatiquement les parties pertinentes d'une application telle que présentée section 4.4.5 est un autre axe de recherche.

Enfin, nous aimerions poursuivre le travail sur les nombres à virgule flottante présenté section 4.4.2. Celui-ci permettrait effectivement d'étendre notre approche à de nombreux algorithmes qui nécessitent des données représentées dans ces formats, mais il pourrait aussi être utilisé dans d'autres contextes. Par exemple, la substitution d'un besoin en calcul par un besoin en mémoire pourrait permettre d'augmenter le débit de calcul d'une application sur FPGA en utilisant ces opérateurs pour une partie des calculs. De plus, une haute précision des calculs peut être souhaitable pour certaines applications : la précision et l'amplitude des données (opérandes et résultats) peuvent être dissociées de la précision des calculs intermédiaires qui peut être arbitrairement élevée. Enfin, afin de réduire le coût en mémoire, il pourrait être intéressant de réduire le nombre de lignes des accumulateurs et de spéculer sur le fait que le nombre de lignes choisi suffira. Cette spéculation a des chances de fonctionner parce que les applications couvrent rarement l'entièreté de l'amplitude d'un format à virgule flottante donné (par exemple 2^{277} soit environ 10^{83} pour le format *binary32*).

Liste des publications

Une partie des travaux de cette thèse ont été publiées dans un journal et une conférence internationaux, ainsi que dans une conférence francophone :

— Thibaut Marty, Tomofumi Yuki et Steven Derrien, **Safe Overclocking for CNN Accelerators through Algorithm-Level Error Detection** dans le journal *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2020

— Thibaut Marty, Tomofumi Yuki et Steven Derrien, **Enabling Overclocking Through Algorithm-Level Error Detection** à la *International Conference on Field-Programmable Technology (FPT)*, 2018, Naha, Japon ;
— Thibaut Marty, Tomofumi Yuki et Steven Derrien, ***Overclocking* sûr de CNN sur FPGA grâce à la détection d'erreur au niveau algorithmique** à la *Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS)*, 2019, Anglet, France

L'auteur de cette thèse a également participé aux travaux suivants :
— Steven Derrien, Thibaut Marty, Simon Rokicki et Tomofumi Yuki, **Toward Speculative Loop Pipelining for High-Level Synthesis** dans le journal *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2020 ;
— Angeliki Kritikakou, Thibaut Marty et Matthieu Roy, **DYNASCORE : DYNAmic Software COntroller to increase REsource utilization in mixed-critical systems** dans le journal *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2017.

Bibliographie

« 7 Series FPGAs Configurable Logic Block User Guide (UG474) » (2016). In : p. 74.

« 7 Series FPGAs Memory Resources User Guide » (2019). In : p. 88.

Abadi, Martín et al. (14 mars 2016). *TensorFlow : Large-Scale Machine Learning on Heterogeneous Distributed Systems.* arXiv : 1603.04467 [cs]. url : http://arxiv.org/abs/1603.04467 (visité le 12/02/2018).

ABDELOUAHAB, Kamel, Maxime Pelcat, François Berry et Jocelyn Sérot (jan. 2018). *Accelerating CNN Inference on FPGAs : A Survey.* Research Report. Université Clermont Auvergne ; Institut Pascal, Clermont Ferrand ; IETR/INSA Rennes. url : https://hal.archives-ouvertes.fr/hal-01695375 (visité le 15/05/2018).

Alemdar, Hande, Vincent Leroy, Adrien Prost-Boucle et Frédéric Pétrot (mai 2017). « Ternary Neural Networks for Resource-Efficient AI Applications ». In : *2017 International Joint Conference on Neural Networks (IJCNN).* 2017 International Joint Conference on Neural Networks (IJCNN), p. 2547-2554. doi : 10.1109/IJCNN.2017.7966166.

Avizienis, Algirdas, Jean-Claude Laprie et Brian Randell (2001). *Fundamental Concepts of Dependability.* University of Newcastle upon Tyne, Computing Science.

Binder, D., E. C. Smith et A. B. Holman (déc. 1975). « Satellite Anomalies from Galactic Cosmic Rays ». In : *IEEE Transactions on Nuclear Science* 22.6, p. 2675-2680. issn : 1558-1578. doi : 10.1109/TNS.1975.4328188.

Bosilca, George, Aurelien Bouteiller, Thomas Herault, Yves Robert et Jack Dongarra (10 jan. 2015). « Composing Resilience Techniques : ABFT, Periodic and Incremental Checkpointing ». In : *International Journal of Networking and Computing* 5.1, p. 2-25. issn : 2185-2839, 2185-2847. doi : 10.15803/ijnc.5.1_2. url : https://www.jstage.jst.go.jp/article/ijnc/5/1/5_2/_article/-char/ja/ (visité le 12/02/2018).

Braun, Claus, Sebastian Halder et Hans Joachim Wunderlich (juin 2014). « A-ABFT : Autonomous Algorithm-Based Fault Tolerance for Matrix Multiplications on Graphics Processing Units ». In : *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* 2014 44th Annual IEEE/IFIP Interna-

tional Conference on Dependable Systems and Networks, p. 443-454. DOI : 10.1109/DSN.2014.48.

Cai, Yaohui, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W. Mahoney et Kurt Keutzer (1er jan. 2020). *ZeroQ : A Novel Zero Shot Quantization Framework*. arXiv : 2001.00281 [cs]. URL : http://arxiv.org/abs/2001.00281 (visité le 18/02/2020).

Campbell, Keith A., Pranay Vissa, David Z. Pan et Deming Chen (juin 2015). « High-Level Synthesis of Error Detecting Cores through Low-Cost modulo-3 Shadow Datapaths ». In : *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), p. 1-6. DOI : 10.1145/2744769.2744851.

Canziani, Alfredo, Adam Paszke et Eugenio Culurciello (2016). *An Analysis of Deep Neural Network Models for Practical Applications*. arXiv : 1605.07678.

Cesana, Giorgio, Philippe Flatresse et Xavier Cauchy (fév. 2012). « Planar Fully Depleted Silicon Technology to Design Competitive SOC at 28nm and Beyond ». In : p. 16.

Chandrakasan, Anantha P., Samuel Sheng et Robert W. Brodersen (1992). « Low-Power CMOS Digital Design ». In : *IEICE Transactions on Electronics* 75.4, p. 371-382.

Chellapilla, Kumar, Sidd Puri et Patrice Simard (23 oct. 2006). « High Performance Convolutional Neural Networks for Document Processing ». In : Tenth International Workshop on Frontiers in Handwriting Recognition. Suvisoft. URL : https://hal.inria.fr/inria-00112631/document (visité le 12/02/2018).

Chen, Yu-Hsin, Joel Emer et Vivienne Sze (2016). « Eyeriss : A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks ». In : *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA '16. Piscataway, NJ, USA : IEEE Press, p. 367-379. ISBN : 978-1-4673-8947-1. DOI : 10.1109/ISCA.2016.40. URL : https://doi.org/10.1109/ISCA.2016.40 (visité le 12/02/2018).

Chen, Zizhong (2013). « Online-ABFT : An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods ». In : *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '13. New York, NY, USA : ACM, p. 167-176. ISBN : 978-1-4503-1922-5. DOI : 10.1145/2442516.2442533. URL : http://doi.acm.org/10.1145/2442516.2442533 (visité le 12/02/2018).

Courbariaux, Matthieu, Itay Hubara, Daniel Soudry, Ran El-Yaniv et Yoshua Bengio (17 mars 2016). *Binarized Neural Networks : Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. arXiv : 1602.02830 [cs]. url : http://arxiv.org/abs/1602.02830 (visité le 22/02/2021).

Dally, William J., James Balfour, David Black-Shaffer, James Chen, R. Curtis Harting, Vishal Parikh, Jongsoo Park et David Sheffield (juill. 2008). « Efficient Embedded Computing ». In : *Computer* 41.7, p. 27-32. issn : 1558-0814. doi : 10.1109/MC.2008.224.

Davis, J. J. et P. Y. K. Cheung (sept. 2014). « Achieving Low-Overhead Fault Tolerance for Parallel Accelerators with Dynamic Partial Reconfiguration ». In : *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014 24th International Conference on Field Programmable Logic and Applications (FPL), p. 1-6. doi : 10.1109/FPL.2014.6927447.

De Dinechin, Benôit Dupont et al. (sept. 2013). « A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications ». In : *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. 2013 IEEE High Performance Extreme Computing Conference (HPEC), p. 1-6. doi : 10.1109/HPEC.2013.6670342.

De Dinechin, Florent, Silviu-Ioan Filip, Martin Kumm et Luc Forget (juin 2019). « Table-Based versus Shift-And-Add Constant Multipliers for FPGAs ». In : *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), p. 151-158. doi : 10.1109/ARITH.2019.00037.

Deng, Jiacnao et al. (mars 2015). « Retraining-Based Timing Error Mitigation for Hardware Neural Networks ». In : *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015 Design, Automation Test in Europe Conference Exhibition (DATE), p. 593-596.

Derrien, Steven, Thibaut Marty, Simon Rokicki et Tomofumi Yuki (nov. 2020). « Toward Speculative Loop Pipelining for High-Level Synthesis ». In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11, p. 4229-4239. issn : 1937-4151. doi : 10.1109/TCAD.2020.3012866.

Dumoulin, Vincent et Francesco Visin (23 mars 2016). *A Guide to Convolution Arithmetic for Deep Learning*. arXiv : 1603.07285 [cs, stat]. url : http://arxiv.org/abs/1603.07285 (visité le 12/02/2018).

ERNST, Dan et al. (2003). « Razor : A Low-Power Pipeline Based on Circuit-Level Timing Speculation ». In : *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 36. Washington, DC, USA : IEEE Computer Society, p. 7-. ISBN : 978-0-7695-2043-8. URL : `http://dl.acm.org/citation.cfm?id=956417.956571` (visité le 12/02/2018).

FLEMING, S. T. et D. B. THOMAS (juin 2016). « StitchUp : Automatic Control Flow Protection for High Level Synthesis Circuits ». In : *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), p. 1-6. DOI : `10.1145/2897937.2898097`.

FOJTIK, M., D. FICK, Y. KIM, N. PINCKNEY, D. M. HARRIS, D. BLAAUW et D. SYLVESTER (jan. 2013). « Bubble Razor : Eliminating Timing Margins in an ARM Cortex-M3 Processor in 45 Nm CMOS Using Architecturally Independent Error Detection and Correction ». In : *IEEE Journal of Solid-State Circuits* 48.1, p. 66-81. ISSN : 0018-9200. DOI : `10.1109/JSSC.2012.2220912`.

GAUTAM et S. RAJOPADHYE (2006). « Simplifying Reductions ». In : *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. New York, NY, USA : ACM, p. 30-41. ISBN : 978-1-59593-027-9. DOI : `10.1145/1111037.1111041`. URL : `http://doi.acm.org/10.1145/1111037.1111041` (visité le 12/02/2018).

GOJMAN, Benjamin, Sirisha NALMELA, Nikil MEHTA, Nicholas HOWARTH et André DEHON (2013). « GROK-LAB : Generating Real on-Chip Knowledge for Intra-Cluster Delays Using Timing Extraction ». In : *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, p. 81-90.

HAMMING, R. W. (avr. 1950). « Error Detecting and Error Correcting Codes ». In : *The Bell System Technical Journal* 29.2, p. 147-160. ISSN : 0005-8580. DOI : `10.1002/j.1538-7305.1950.tb00463.x`.

HAN, Song, Xingyu LIU, Huizi MAO, Jing PU, Ardavan PEDRAM, Mark A. HOROWITZ et William J. DALLY (3 mai 2016). *EIE : Efficient Inference Engine on Compressed Deep Neural Network*. arXiv : `1602.01528 [cs]`. URL : `http://arxiv.org/abs/1602.01528` (visité le 02/08/2020).

HAN, Song, Jeff POOL, John TRAN et William DALLY (2015). « Learning Both Weights and Connections for Efficient Neural Network ». In : *Advances in Neural Information Processing Systems*, p. 1135-1143.

HARI, Siva Kumar Sastry, Michael B. SULLIVAN, Timothy TSAI et Stephen W. KECKLER (8 juin 2020). *Making Convolutions Resilient via Algorithm-Based Error Detection Techniques*. arXiv : 2006.04984 [cs]. URL : http://arxiv.org/abs/2006.04984 (visité le 22/07/2020).

HE, Kaiming, Xiangyu ZHANG, Shaoqing REN et Jian SUN (2016). « Deep Residual Learning for Image Recognition ». In : Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, p. 770-778. URL : https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html (visité le 09/02/2021).

HENNESSY, John L. et David A. PATTERSON (2011). *Computer Architecture : A Quantitative Approach*. Elsevier.

HIEMSTRA, D. M. et V. KIRISCHIAN (juill. 2012). « Single Event Upset Characterization of the Virtex-6 Field Programmable Gate Array Using Proton Irradiation ». In : *2012 IEEE Radiation Effects Data Workshop*. 2012 IEEE Radiation Effects Data Workshop, p. 1-4. DOI : 10.1109/REDW.2012.6353716.

HUANG, Kuang-Hua et J. A. ABRAHAM (juin 1984). « Algorithm-Based Fault Tolerance for Matrix Operations ». In : *IEEE Transactions on Computers* C-33.6, p. 518-528. ISSN : 0018-9340. DOI : 10.1109/TC.1984.1676475.

IOFFE, Sergey et Christian SZEGEDY (6 juill. 2015). « Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift ». In : *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France : JMLR.org, p. 448-456.

JACOBS, A., G. CIESLEWSKI et A. D. GEORGE (août 2012). « Overhead and Reliability Analysis of Algorithm-Based Fault Tolerance in FPGA Systems ». In : *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 22nd International Conference on Field Programmable Logic and Applications (FPL), p. 300-306. DOI : 10.1109/FPL.2012.6339222.

JIAO, Xun, Mulong LUO, Jeng-Hau LIN et Rajesh K. GUPTA (2017). « An Assessment of Vulnerability of Hardware Neural Networks to Dynamic Voltage and Temperature Variations ». In : *Proceedings of the 36th International Conference on Computer-Aided Design*. ICCAD '17. Piscataway, NJ, USA : IEEE Press, p. 945-950. URL : http://dl.acm.org/citation.cfm?id=3199700.3199830 (visité le 16/07/2018).

JOHNSON, Jeff (1er nov. 2018). *Rethinking Floating Point for Deep Learning*. arXiv : 1811.01721 [cs]. URL : http://arxiv.org/abs/1811.01721 (visité le 22/02/2021).

Jouppi, Norman P. et al. (24 juin 2017). « In-Datacenter Performance Analysis of a Tensor Processing Unit ». In : *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. New York, NY, USA : Association for Computing Machinery, p. 1-12. isbn : 978-1-4503-4892-8. doi : 10.1145/3079856.3080246. url : https://doi.org/10.1145/3079856.3080246 (visité le 01/06/2021).

Kahng, Andrew B., Jens Lienig, Igor L. Markov et Jin Hu (2011). *VLSI Physical Design : From Graph Partitioning to Timing Closure*. Springer Science & Business Media.

Kiamehr, Saman, Farshad Firouzi et Mehdi. B. Tahoori (mars 2013). « Aging-Aware Timing Analysis Considering Combined Effects of NBTI and PBTI ». In : *International Symposium on Quality Electronic Design (ISQED)*. International Symposium on Quality Electronic Design (ISQED), p. 53-59. doi : 10.1109/ISQED.2013.6523590.

Kritikakou, Angeliki, Thibaut Marty et Matthieu Roy (oct. 2017). « DYNASCORE : DYNAmic Software COntroller to Increase REsource Utilization in Mixed-Critical Systems ». In : *ACM Trans. Des. Autom. Electron. Syst.* 23.2, 13 :1-13 :26. issn : 1084-4309. doi : 10.1145/3110222. url : http://doi.acm.org/10.1145/3110222 (visité le 12/02/2018).

Krizhevsky, Alex, Ilya Sutskever et Geoffrey E Hinton (2012). « ImageNet Classification with Deep Convolutional Neural Networks ». In : *Advances in Neural Information Processing Systems 25*. Sous la dir. de F. Pereira, C. J. C. Burges, L. Bottou et K. Q. Weinberger. Curran Associates, Inc., p. 1097-1105. url : http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf (visité le 12/02/2018).

Kulisch, Ulrich (2013). *Computer Arithmetic and Validity : Theory, Implementation, and Applications*. T. 33. Walter de Gruyter.

Lavin, Andrew et Scott Gray (2016). « Fast Algorithms for Convolutional Neural Networks ». In : *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, p. 4013-4021.

Lawrence, S., C. L. Giles, Ah Chung Tsoi et A. D. Back (jan. 1997). « Face Recognition : A Convolutional Neural-Network Approach ». In : *IEEE Transactions on Neural Networks* 8.1, p. 98-113. issn : 1045-9227. doi : 10.1109/72.554195.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard et L. D. Jackel (1er déc. 1989). « Backpropagation Applied to Handwritten Zip Code Recognition ». In : *Neural Computation* 1.4, p. 541-551. issn : 0899-7667. doi : 10.

1162/neco.1989.1.4.541. URL : https://doi.org/10.1162/neco.1989.1.4.541 (visité le 29/08/2020).

LECUN, Y., L. BOTTOU, Y. BENGIO et P. HAFFNER (nov. 1998). « Gradient-Based Learning Applied to Document Recognition ». In : *Proceedings of the IEEE* 86.11, p. 2278-2324. ISSN : 1558-2256. DOI : 10.1109/5.726791.

LECUN, Yann, Yoshua BENGIO et Geoffrey HINTON (mai 2015). « Deep Learning ». In : *Nature* 521.7553 (7553), p. 436-444. ISSN : 1476-4687. DOI : 10.1038/nature14539. URL : https://www.nature.com/articles/nature14539 (visité le 02/02/2021).

LENG, Jingwen, Alper BUYUKTOSUNOGLU, Ramon BERTRAN, Pradip BOSE et Vijay Janapa REDDI (déc. 2015). « Safe Limits on Voltage Reduction Efficiency in GPUs : A Direct Measurement Approach ». In : *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), p. 294-307. DOI : 10.1145/2830772.2830811.

LEVINE, J. M., E. STOTT, G. A. CONSTANTINIDES et P. Y. K. CHEUNG (avr. 2012). « Online Measurement of Timing in Circuits : For Health Monitoring and Dynamic Voltage #x0026 ; Frequency Scaling ». In : *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, p. 109-116. DOI : 10.1109/FCCM.2012.27.

— (sept. 2013). « SMI : Slack Measurement Insertion for Online Timing Monitoring in FPGAs ». In : *2013 23rd International Conference on Field Programmable Logic and Applications*. 2013 23rd International Conference on Field Programmable Logic and Applications, p. 1-4. DOI : 10.1109/FPL.2013.6645598.

LEVINE, Joshua M., Edward STOTT et Peter Y.K. CHEUNG (2014). « Dynamic Voltage & Frequency Scaling with Online Slack Measurement ». In : *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '14. New York, NY, USA : ACM, p. 65-74. ISBN : 978-1-4503-2671-1. DOI : 10.1145/2554688.2554784. URL : http://doi.acm.org/10.1145/2554688.2554784 (visité le 23/05/2018).

LI, Guanpeng, Siva Kumar Sastry HARI, Michael SULLIVAN, Timothy TSAI, Karthik PATTABIRAMAN, Joel EMER et Stephen W. KECKLER (2017). « Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications ». In : *Proceedings of the International Conference for High Performance Com-*

puting, Networking, Storage and Analysis. SC '17. New York, NY, USA : ACM, 8 :1-8 :12. ISBN : 978-1-4503-5114-0. DOI : 10.1145/3126908.3126964. URL : http://doi.acm.org/10.1145/3126908.3126964 (visité le 01/06/2018).

LIANG, Xin, Jieyang CHEN, Dingwen TAO, Sihuan LI, Panruo WU, Hongbo LI, Kaiming OUYANG, Yuanlai LIU, Fengguang SONG et Zizhong CHEN (2017). « Correcting Soft Errors Online in Fast Fourier Transform ». In : *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado). SC '17. New York, NY, USA : ACM, 30 :1-30 :12. ISBN : 978-1-4503-5114-0. DOI : 10.1145/3126908.3126915. URL : http://doi.acm.org/10.1145/3126908.3126915 (visité le 12/11/2019).

LIANG, Y., L. LU, Q. XIAO et S. YAN (2019). « Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs ». In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. 1-1. ISSN : 0278-0070. DOI : 10.1109/TCAD.2019.2897701.

LIBANO, F., B. WILSON, M. WIRTHLIN, P. RECH et J. BRUNHAVER (juill. 2020). « Understanding the Impact of Quantization, Accuracy, and Radiation on the Reliability of Convolutional Neural Networks on FPGAs ». In : *IEEE Transactions on Nuclear Science* 67.7, p. 1478-1484. ISSN : 1558-1578. DOI : 10.1109/TNS.2020.2983662.

LIN, Y., S. ZHANG et N. R. SHANBHAG (oct. 2016). « Variation-Tolerant Architectures for Convolutional Neural Networks in the Near Threshold Voltage Regime ». In : *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*. 2016 IEEE International Workshop on Signal Processing Systems (SiPS), p. 17-22. DOI : 10.1109/SiPS.2016.11.

LINDHOLM, Erik, John NICKOLLS, Stuart OBERMAN et John MONTRYM (mars 2008). « NVIDIA Tesla : A Unified Graphics and Computing Architecture ». In : *IEEE Micro* 28.2, p. 39-55. ISSN : 1937-4143. DOI : 10.1109/MM.2008.31.

LU, Liqiang, Jiaming XIE, Ruirui HUANG, Jiansong ZHANG, Wei LIN et Yun LIANG (avr. 2019). « An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs ». In : *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), p. 17-25. DOI : 10.1109/FCCM.2019.00013.

MARTY, Thibaut, Tomofumi YUKI et Steven DERRIEN (déc. 2018). « Enabling Overclocking Through Algorithm-Level Error Detection ». In : *2018 International Conference*

on Field-Programmable Technology (FPT). 2018 International Conference on Field-Programmable Technology (FPT), p. 174-181. DOI : 10.1109/FPT.2018.00034.

— (déc. 2020). « Safe Overclocking for CNN Accelerators Through Algorithm-Level Error Detection ». In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.12, p. 4777-4790. ISSN : 1937-4151. DOI : 10.1109/TCAD.2020.2981056.

MATHIEU, Michael, Mikael HENAFF et Yann LECUN (6 mars 2014). *Fast Training of Convolutional Networks through FFTs*. arXiv : 1312.5851 [cs]. URL : http://arxiv.org/abs/1312.5851 (visité le 18/02/2021).

MICHAUD, Pierre, André SEZNEC et Stéphan JOURDAN (1er fév. 2001). « An Exploration of Instruction Fetch Requirement in Out-of-Order Superscalar Processors ». In : *International Journal of Parallel Programming* 29.1, p. 35-58. ISSN : 1573-7640. DOI : 10.1023/A:1026431920605. URL : https://doi.org/10.1023/A:1026431920605 (visité le 27/05/2021).

MITTAL, Sparsh et Jeffrey S. VETTER (25 août 2014). « A Survey of Methods for Analyzing and Improving GPU Energy Efficiency ». In : *ACM Computing Surveys* 47.2, 19 :1-19 :23. ISSN : 0360-0300. DOI : 10.1145/2636342. URL : https://doi.org/10.1145/2636342 (visité le 01/06/2021).

MOHAN, P. V. Ananda (2002). *Residue Number Systems : Algorithms and Architectures*. The Springer International Series in Engineering and Computer Science. Springer US. ISBN : 978-1-4020-7031-0. URL : //www.springer.com/us/book/9781402070310 (visité le 28/09/2018).

— (14 oct. 2016). *Residue Number Systems : Theory and Applications*. Birkhäuser. 353 p. ISBN : 978-3-319-41385-3. Google Books : VaFDDQAAQBAJ.

MOLAHOSSEINI, A. S., K. NAVI, C. DADKHAH, O. KAVEHEI et S. TIMARCHI (avr. 2010). « Efficient Reverse Converter Designs for the New 4-Moduli Sets$2^n - 1, 2^n, 2^n + 1, 2^2n + 1 - 1$and2 on New CRTs ». In : *IEEE Transactions on Circuits and Systems I : Regular Papers* 57.4, p. 823-835. ISSN : 1549-8328. DOI : 10.1109/TCSI.2009.2026681.

NUNEZ-YANEZ, J. L. (2018). « Energy Proportional Neural Network Inference with Adaptive Voltage and Frequency Scaling ». In : *IEEE Transactions on Computers*, p. 1-1. ISSN : 0018-9340. DOI : 10.1109/TC.2018.2879333.

NUNEZ-YANEZ, Jose (10 sept. 2013). « Energy Proportional Computing in Commercial FPGAs with Adaptive Voltage Scaling ». In : *Proceedings of the 10th FPGAworld Conference*. FPGAworld '13. Stockholm, Sweden : Association for Computing Ma-

chinery, p. 1-5. ISBN : 978-1-4503-2496-0. DOI : 10.1145/2513683.2513689. URL : https://doi.org/10.1145/2513683.2513689 (visité le 23/06/2020).

« NVIDIA's Next Generation CUDA Compute Architecture » (2009). « NVIDIA's Next Generation CUDA Compute Architecture : Fermi ». In : *Nvidia Whitepaper*.

Oh, Nahmsuk, Philip P. Shirvani et Edward J. McCluskey (2002). « Control-Flow Checking by Software Signatures ». In : *IEEE transactions on Reliability* 51.1, p. 111-122.

Piestrak, S. J. (jan. 1994). « Design of Residue Generators and Multioperand Modular Adders Using Carry-Save Adders ». In : *IEEE Transactions on Computers* 43.1, p. 68-77. ISSN : 0018-9340. DOI : 10.1109/12.250610.

Piestrak, S. J. et P. Patronik (août 2014). « Design of Fault-Secure Transposed FIR Filters Protected Using Residue Codes ». In : *2014 17th Euromicro Conference on Digital System Design*. 2014 17th Euromicro Conference on Digital System Design, p. 575-582. DOI : 10.1109/DSD.2014.110.

Poli, Alain et Llorenç Huguet (1992). *Error Correcting Codes : Theory and Applications*. Prentice Hall. 536 p. ISBN : 978-0-13-284894-7.

Prost-Boucle, Adrien, Alban Bourge et Frédéric Pétrot (12 déc. 2018). « High-Efficiency Convolutional Ternary Neural Networks with Custom Adder Trees and Weight Compression ». In : *ACM Transactions on Reconfigurable Technology and Systems* 11.3, 15 :1-15 :24. ISSN : 1936-7406. DOI : 10.1145/3270764. URL : https://doi.org/10.1145/3270764 (visité le 10/10/2021).

Qiu, Jiantao et al. (2016). « Going Deeper with Embedded FPGA Platform for Convolutional Neural Network ». In : *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. New York, NY, USA : ACM, p. 26-35. ISBN : 978-1-4503-3856-1. DOI : 10.1145/2847263.2847265. URL : http://doi.acm.org/10.1145/2847263.2847265 (visité le 12/02/2018).

Quinn, H., P. Graham, K. Morgan, Z. Baker, M. Caffrey, D. Smith et R. Bell (juill. 2012). « On-Orbit Results for the Xilinx Virtex-4 FPGA ». In : *2012 IEEE Radiation Effects Data Workshop*. 2012 IEEE Radiation Effects Data Workshop, p. 1-8. DOI : 10.1109/REDW.2012.6353715.

Quinton, P. et Yves Robert (1989). *Algorithmes et Architectures Systoliques*. Masson, 353 p. URL : https://hal.inria.fr/hal-00857013 (visité le 09/06/2021).

Rastegari, Mohammad, Vicente Ordonez, Joseph Redmon et Ali Farhadi (2016). « Xnor-Net : Imagenet Classification Using Binary Convolutional Neural Networks ». In : *European Conference on Computer Vision*. Springer, p. 525-542.

Rebaud, B., M. Belleville, E. Beigné, C. Bernard, M. Robert, P. Maurine et N. Azemard (1er mai 2011). « Timing Slack Monitoring under Process and Environmental Variations : Application to a DSP Performance Optimization ». In : *Microelectronics Journal* 42.5, p. 718-732. issn : 0026-2692. doi : 10.1016/j.mejo.2011.02.005. url : http://www.sciencedirect.com/science/article/pii/S0026269211000292 (visité le 16/10/2018).

Reddy, A. L. N. et P. Banerjee (oct. 1990). « Algorithm-Based Fault Detection for Signal Processing Applications ». In : *IEEE Transactions on Computers* 39.10, p. 1304-1308. issn : 0018-9340. doi : 10.1109/12.59860.

Reis, George A., Jonathan Chang, Neil Vachharajani, Ram Rangan et David I. August (2005). « SWIFT : Software Implemented Fault Tolerance ». In : *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '05. Washington, DC, USA : IEEE Computer Society, p. 243-254. isbn : 978-0-7695-2298-2. doi : 10.1109/CGO.2005.34. url : http://dx.doi.org/10.1109/CGO.2005.34 (visité le 12/02/2018).

Roy-Chowdhury, A., N. Bellas et P. Banerjee (avr. 1996). « Algorithm-Based Error-Detection Schemes for Iterative Solution of Partial Differential Equations ». In : *IEEE Transactions on Computers* 45.4, p. 394-407. issn : 0018-9340. doi : 10.1109/12.494098.

Rumelhart, David E., Geoffrey E. Hinton et Ronald J. Williams (oct. 1986). « Learning Representations by Back-Propagating Errors ». In : *Nature* 323.6088 (6088), p. 533-536. issn : 1476-4687. doi : 10.1038/323533a0. url : https://www.nature.com/articles/323533a0 (visité le 09/02/2021).

Ruospo, Annachiara, Ernesto Sanchez, Marcello Traiola, Ian O'Connor et Alberto Bosio (1er oct. 2021). « Investigating Data Representation for Efficient and Reliable Convolutional Neural Networks ». In : *Microprocessors and Microsystems* 86, p. 104318. issn : 0141-9331. doi : 10.1016/j.micpro.2021.104318. url : https://www.sciencedirect.com/science/article/pii/S0141933121004786 (visité le 17/12/2021).

Salami, Behzad, Erhan Baturay Onural, Ismail Emir Yuksel, Fahrettin Koc, Oguz Ergin, Adrian Cristal Kestelman, Osman S. Unsal, Hamid Sarbazi-Azad et

Onur Mutlu (4 mai 2020). *An Experimental Study of Reduced-Voltage Operation in Modern FPGAs for Neural Network Acceleration*. arXiv : 2005.03451 [cs]. url : http://arxiv.org/abs/2005.03451 (visité le 15/05/2020).

Salman, Emre, Ali Dasdan, Feroze Taraporevala, Kayhan Kucukcakar et Eby G. Friedman (juin 2007). « Exploiting Setup–Hold-Time Interdependence in Static Timing Analysis ». In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.6, p. 1114-1125. issn : 1937-4151. doi : 10.1109/TCAD.2006.885834.

Sankaradas, M., V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto et H. P. Graf (juill. 2009). « A Massively Parallel Coprocessor for Convolutional Neural Networks ». In : *2009 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors*. 2009 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors, p. 53-60. doi : 10.1109/ASAP.2009.25.

Sentieys, Olivier, Silviu Filip, David Briand, David Novo, Etienne Dupuis, Ian O'Connor et Alberto Bosio (avr. 2021). « AdequateDL : Approximating Deep Learning Accelerators ». In : *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 2021 24th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS), p. 37-40. doi : 10.1109/DDECS52668.2021.9417026.

Simonyan, Karen et Andrew Zisserman (10 avr. 2015). *Very Deep Convolutional Networks for Large-Scale Image Recognition*. arXiv : 1409.1556 [cs]. url : http://arxiv.org/abs/1409.1556 (visité le 29/08/2020).

Stott, E., P. Sedcole et P. Cheung (mai 2010). « Fault Tolerance and Reliability in Field-Programmable Gate Arrays ». In : *IET Computers Digital Techniques* 4.3, p. 196-210. issn : 1751-861X. doi : 10.1049/iet-cdt.2009.0011.

Stott, Edward, Zhenyu Guan, Joshua M. Levine, Justin S. J. Wong et Peter Y. K. Cheung (déc. 2013). « Variation and Reliability in FPGAs ». In : *IEEE Design Test* 30.6, p. 50-59. issn : 2168-2364. doi : 10.1109/MDAT.2013.2266652.

Suda, Naveen, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo et Yu Cao (2016). « Throughput-Optimized OpenCL-Based FPGA Accelerator for Large-Scale Convolutional Neural Networks ». In : *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. New York, NY, USA : ACM, p. 16-25. isbn : 978-1-4503-3856-1. doi :

10.1145/2847263.2847276. URL : http://doi.acm.org/10.1145/2847263.2847276 (visité le 12/02/2018).

Sze, V., Y. Chen, T. Yang et J. S. Emer (déc. 2017). « Efficient Processing of Deep Neural Networks : A Tutorial and Survey ». In : *Proceedings of the IEEE* 105.12, p. 2295-2329. ISSN : 1558-2256. DOI : 10.1109/JPROC.2017.2761740.

Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke et Andrew Rabinovich (juin 2015). « Going Deeper with Convolutions ». In : *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), p. 1-9. DOI : 10.1109/CVPR.2015.7298594.

Torres-Huitzil, Cesar et Bernard Girau (déc. 2017). « Fault Tolerance in Neural Networks : Neural Design and Hardware Implementation ». In : *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig), p. 1-6. DOI : 10.1109/RECONFIG.2017.8279793.

Umuroglu, Yaman, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre et Kees Vissers (2017). « FINN : A Framework for Fast, Scalable Binarized Neural Network Inference ». In : *Proceedings of the 2017 ACM/-SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA). FPGA '17. New York, NY, USA : ACM, p. 65-74. ISBN : 978-1-4503-4354-1. DOI : 10.1145/3020078.3021744. URL : http://doi.acm.org/10.1145/3020078.3021744 (visité le 14/02/2019).

Venieris, Stylianos I., Alexandros Kouris et Christos-Savvas Bouganis (juin 2018). « Toolflows for Mapping Convolutional Neural Networks on FPGAs : A Survey and Future Directions ». In : *ACM Comput. Surv.* 51.3, 56 :1-56 :39. ISSN : 0360-0300. DOI : 10.1145/3186332. URL : http://doi.acm.org/10.1145/3186332 (visité le 20/07/2018).

Winograd, Shmuel (1980). *Arithmetic Complexity of Computations*. T. 33. Siam.

Wulf, Wm A. et Sally A. McKee (1995). « Hitting the Memory Wall : Implications of the Obvious ». In : *ACM SIGARCH computer architecture news* 23.1, p. 20-24.

Al-Yamani, A. A., N. Oh et E. J. McCluskey (2001). « Performance Evaluation of Checksum-Based ABFT ». In : *Proceedings 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. Proceedings 2001 IEEE International

Symposium on Defect and Fault Tolerance in VLSI Systems, p. 461-466. DOI : 10.1109/DFTVS.2001.966800.

YANG, Tien-Ju, Yu-Hsin CHEN et Vivienne SZE (2017). « Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning ». In : *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, p. 5687-5695.

ZHANG, Chen, Peng LI, Guangyu SUN, Yijin GUAN, Bingjun XIAO et Jason CONG (2015). « Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks ». In : *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. New York, NY, USA : ACM, p. 161-170. ISBN : 978-1-4503-3315-3. DOI : 10.1145/2684746.2689060. URL : http://doi.acm.org/10.1145/2684746.2689060 (visité le 12/02/2018).

ZHANG, Chi et Viktor PRASANNA (22 fév. 2017). « Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System ». In : *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. New York, NY, USA : Association for Computing Machinery, p. 35-44. ISBN : 978-1-4503-4354-1. DOI : 10.1145/3020078.3021727. URL : https://doi.org/10.1145/3020078.3021727 (visité le 18/02/2021).

ZHANG, Jeff, Zahra GHODSI, Kartheek RANGINENI et Siddharth GARG (2019). « Enabling Timing Error Resilience for Low-Power Systolic-Array Based Deep Learning Accelerators ». In : *IEEE Design Test*, p. 1-1. ISSN : 2168-2364. DOI : 10.1109/MDAT.2019.2947271.

ZHAO, Kai, Sheng DI, Sihuan LI, Xin LIANG, Yujia ZHAI, Jieyang CHEN, Kaiming OUYANG, Franck CAPPELLO et Zizhong CHEN (juill. 2021). « FT-CNN : Algorithm-Based Fault Tolerance for Convolutional Neural Networks ». In : *IEEE Transactions on Parallel and Distributed Systems* 32.7, p. 1677-1689. ISSN : 1558-2183. DOI : 10.1109/TPDS.2020.3043449.

UNIVERSITÉ DE RENNES 1

Titre : Spéculation temporelle pour accélérateurs matériels

Mot clés : Efficacité énergétique, Tolérance aux fautes, Accélération matérielle, Réseaux de neurones convolutifs

Résumé : Cette thèse porte sur l'utilisation de la spéculation temporelle pour améliorer les performances et l'efficacité énergétique d'accélérateurs matériels. La spéculation temporelle consiste en l'utilisation d'un circuit en utilisant une fréquence ou une tension à laquelle son fonctionnement n'est plus garanti. Elle permet d'augmenter les performances du circuit (calculs par seconde) mais aussi son efficacité énergétique (calculs par joule). Comme le fonctionnement du circuit n'est plus garanti, elle doit être accompagnée d'un mécanisme de détection d'erreur. Celui-ci doit avoir un coût en ressources utilisées, en énergie et un impact sur les performances les plus faibles possibles. Ces surcoûts doivent effectivement être suffisamment faibles pour que l'approche vaille le coup, mais aussi être le plus bas possible pour maximiser les gains obtenus. Nous présentons un nouveau mécanisme de détection d'erreur au niveau algorithmique pour les convolutions utilisées dans les réseaux de neurones convolutifs qui remplit ces conditions. Nous montrons que la combinaison de ce mécanisme avec la spéculation temporelle permet d'améliorer les performances et l'efficacité énergétique d'un accélérateur matériel de convolution.

Title: Timing speculation for hardware accelerators

Keywords: Energy efficiency, Fault tolerance, Hardware acceleration, Convolutional neural networks

Abstract: This thesis is focused on the use of timing speculation to improve the performance and energy efficiency of hardware accelerators. Timing speculation is the use of a circuit using a frequency or a voltage at which its operation is no longer guaranteed. It increases the performance of the circuit (computations per second) but also its energy efficiency (computations per joule). As the correct operation of the circuit is no longer guaranteed, it must be accompanied by an error detection mechanism. This mechanism must have the lowest possible additional cost in terms of resources used, energy and impact on performance. These overheads must indeed be low enough to make the approach worthwhile, but also be as low as possible to maximize the gain obtained. We present a new algorithm-level error detection mechanism for convolutions used in convolutional neural networks that meets these conditions. We show that combining this mechanism with timing speculation can improve the performance and energy efficiency of a convolution hardware accelerator.