Master research Internship

Bibliographic report

Spéculation temporelle au niveau algorithmique pour les réseaux de neurones convolutifs

Domain: Hardware Architecture

Author:
Thibaut Marty

Supervisor:
Steven Derrien
Cairn

Résumé: Un intérêt croissant est porté à l'implémentation des réseaux de neurones convolutifs (*Convolutional Neural Network*, CNN), dans le domaine de l'apprentissage profond (*deep learning*). En effet, les CNN donnent de meilleurs résultats dans la réalisation de tâches complexes que les algorithmes classiques. Des efforts ont été réalisés pour développer des plateformes améliorant les performances ou l'efficacité énergétique des CNN. Pourtant, la charge de calcul nécessaire reste encore hors de portée pour des systèmes embarqués de faible puissance. Cela reste très coûteux même dans le cas de composants classiques.
L'objectif de ce stage est d'explorer une nouvelle approche basée sur la spéculation temporelle. Cette technique permet à un circuit électronique de consommer moins d'énergie au prix de l'apparition de certains types d'erreurs de calcul. Dans le cas des CNN, ces types d'erreurs ne peuvent être permises. Un moyen très efficace de les détecter pour les corriger est la tolérance aux fautes au niveau algorithmique (*Algorithm-Based Fault Tolerance*, ABFT). Ce travail expose le contexte de ce stage ainsi que les perspectives de travail. Notamment, en plus de quelques quelques schémas d'ABFT existants, il présente un nouveau schéma d'ABFT applicable aux produit de convolution, opération élémentaire utilisée dans les CNN.

Table des matières

1 Introduction

Un intérêt croissant est porté à la conception et l'implémentation des réseaux de neurones convolutifs (*Convolutional Neural Network*, CNN), dans le domaine de l'apprentissage profond (*deep learning*). En effet, ceux-ci donnent de très bons résultats dans la réalisation de tâches complexes comme la vision par ordinateur [10], la reconnaissance faciale [11], le traitement automatique du langage naturel [9], ou encore l'intelligence artificielle [19].

Pourtant, la charge de calcul nécessaire est souvent hors de portée pour des systèmes embarqués de faible puissance, ou très coûteuse sur des composants classiques. Des efforts pour développer des plateformes améliorant les performances ou l'efficacité énergétique ont été réalisés aussi bien sur GPU [10] que sur FPGA [20, 6].

Parmi les techniques utilisées pour limiter la consommation énergétique de circuits électroniques, l'abaissement de la tension d'alimentation est très efficace [5]. Cette solution peut être plus intéressante que l'autre solution principale consistant à baisser la fréquence de fonctionnement. En effet, baisser cette tension permet aux transistors de consommer moins d'énergie, mais les rend moins dynamiques. Un risque d'erreur de calculs (dites temporelles, de *timing*) apparaît donc à cause de l'augmentation des temps de propagation. Ce principe de relâcher les contraintes de tension ou de fréquence est appelée spéculation temporelle [17].

La détection et correction d'erreur est réalisée par un mécanisme de tolérance aux fautes. Détecter et corriger les erreurs au niveau algorithmique plutôt qu'au niveau des instructions permet de réduire considérablement le surcout [8]. Cette approche, nommée ABFT, se base sur des propriétés mathématiques du calcul effectué. Les résultats obtenus avec cette approche sont donc spécifiques, mais très efficaces. Bien que le travail séminal sur l'ABFT date des années 80, ces méthodes sont remises au goût du jour dans le cadre du HPC (*High Performance Computer*) et des systèmes exascale [1, 2, 3].

Ce travail explore une nouvelle approche pour améliorer l'efficacité des implémentations de CNN sur FPGA basée sur la spéculation temporelle et la tolérance aux fautes au niveau algorithmique.

La suite de ce travail présente quelques aspects importants relatifs à cette nouvelle approche. Elle est organisée comme suit : la section 2 donne un aperçu du fonctionnement des réseaux de neurones convolutifs et discute des implémentations existantes. La section 3 présente la spéculation temporelle et discute des erreurs qu'elle génère. La tolérance aux fautes au niveau algorithmique est introduite dans la section 4. La section 5 discute des objectifs de ce stage.

2 Réseaux de neurones convolutifs

Cette section présente les réseaux de neurones convolutifs, leurs principes, ainsi que les différentes implémentations existantes.

Les réseaux de neurones convolutifs sont un type bien connu de réseau de neurones artificiels inspiré par le fonctionnement de la vision des êtres vivants. Ils sont utilisés dans des domaines où les algorithmes classiques échouent aujourd'hui, comme la reconnaissance d'objets dans des images [10], la reconnaissance faciale [11], le traitement automatique du langage naturel [9] ou encore l'intelligence artificielle [10].

Leur avantage par rapport à des réseaux de neurones artificiels classiques est que, à taille similaire, un CNN a moins de connexions et de paramètres et est donc plus facile à entraîner, tout en ayant des performances théoriquement à peine plus faibles [10].

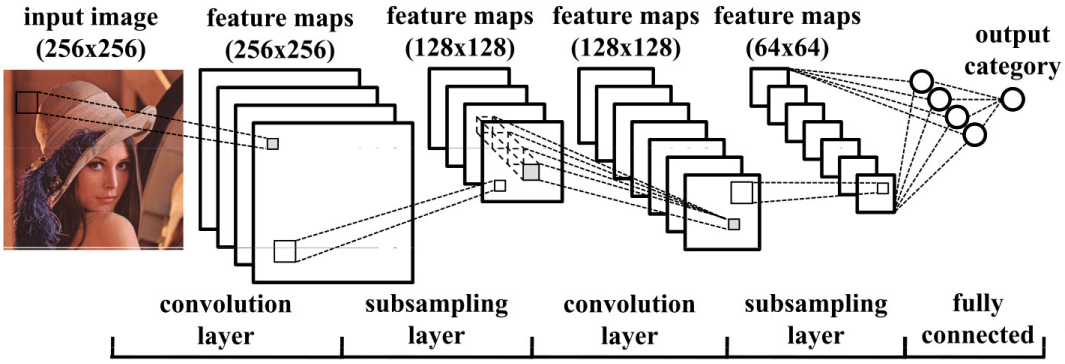

FIGURE 1 – Exemple de CNN classificateur d'image, pris de [4]

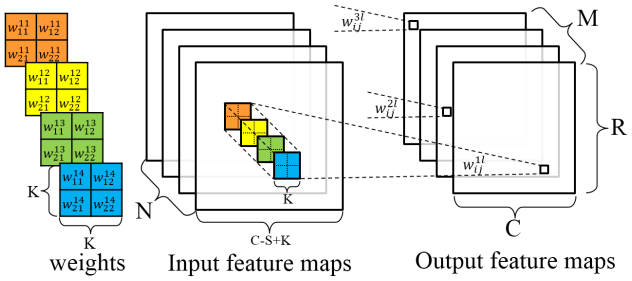

FIGURE 2 – Graphique d'une couche de convolution, prise de [20]

2.1 Principes

Un CNN est principalement composé d'un extracteur de caractéristiques et d'un classificateur. L'extracteur de caractéristiques est composé d'un chaînage de couches de convolution et de couches de sous-échantillonnage. Chaque couche de convolution permet d'extraire des cartes de caractéristiques locales (*feature maps*) depuis les cartes de caractéristiques en entrée, comme illustré sur la figure 2. Les couches de sous-échantillonnage permettent d'abstraire ces caractéristiques à une plus faible résolution. Ce sous-échantillonnage n'est pas linéaire. Par exemple, il peut s'agir du maximum des pixels considérés (*max pooling*). Les caractéristiques peuvent être des bords, des coins, etc. Le classificateur est généralement basé sur un réseau de neurones classique, avec des couches de neurones complètement connectées. Celui-ci prend les carte de caractéristiques en entrée et donne la vraisemblance d'appartenance de l'entrée du CNN à chaque catégorie. De plus, une fonction d'activation non linéaire est appliquée à chaque pixel des cartes de caractéristiques pour imiter le fonctionnement des neurones. Cet architecture est illustrée par la figure 1.

Un CNN est normalement utilisé dans le sens direct, c'est-à-dire que les informations transitent uniquement de l'entrée vers la sortie. Cependant, il a besoin d'être entrainé, autrement dit en utilisant une boucle de rétroaction. L'entraînement est effectué sur des données d'exemple afin de régler tous les paramètres du réseau (noyaux convolutifs et poids du réseau de neurones classificateur).

Une fois entraîné, le CNN est utilisé uniquement dans le sens direct. L'optimisation pour cette utilisation est donc la plus importante, car c'est elle qui est utilisée en fonctionnement normal. De plus, Cong et Xiao ont montré expérimentalement que les couches de convolutions représentent plus

```
// fm signifie feature map, ou carte de caractéristiques
for(to = 0; to < M ; to++) {          // fm de sortie
  for(ti = 0; ti < N ; ti++) {        // fm d'entrée
    for(row = 0; row < R ; row++) {   // ligne dans la fm de sortie
      for(col = 0; col < C ; col++) { // colonne dans la fm de sortie
        for(i = 0; i < K ; i++) {     // ligne du noyau de convolution
          for(j = 0; j < K ; j++) {   // colonne du noyau de convolution
            // calcul de convolution
            output_fm[to][row][col] +=
              weights[to][ti][i][j] *
              input_fm[ti][S * row + i][S * col + j];
} } } } } }
```

FIGURE 3 – Code d'une couche de convolution

de 90% du temps de calcul d'un CNN lors de la reconnaissance de 256 images [4]. L'optimisation de l'exécution des convolutions est donc primordiale afin d'optimiser l'ensemble d'un CNN.

Une couche de convolution peut être calculée grâce à un nid de boucles imbriquées, comme indiqué sur la figure 3. Dans ce cas, la couche traite N carte de caractéristiques et en génère M. Chaque carte de caractéristiques a une taille $C \times R$. Le noyau de convolution a une taille $K \times K$ La complexité en temps pour ce calcul est $O(RCMNK^2)$.

2.2 Implémentations

La complexité des opérations utilisées dans les CNN ainsi que le volume de données qu'ils doivent traiter rendent leur implémentation délicate et les optimisations importantes.

Farabet et al. [6] ont montré que les CPU classiques sont peu performants en termes de consommation énergétique et de débit par rapport à des implémentations matérielles sur FPGA ou ASIC. Ils proposent une implémentation pour FPGA ou ASIC basée sur un processeur SIMD (*Single Instruction Multiple Data*) comportant des instructions accélérées matériellement spécifiques pour les CNN.

Zhang et al. [20] optimisent les calculs et les accès mémoires sur une implémentation sur FPGA en exploitant les opportunités de parallélisme intrinsèques aux CNN. Afin de faire tenir les données dans les mémoires internes du FPGA, les boucles externes sont tuilées. La boucle intermédiaire, soit la première boucle de chaque tuile, est transformée pour que ses exécutions se chevauchent (*loop pipelining*). Les deux boucles internes sont entièrement déroulées et toutes les itérations sont exécutées en parallèle.

Ces travaux montrent qu'un intérêt particulier est porté à l'optimisation des implémentations de CNN. Cependant, ces implémentations ne sont toujours pas suffisamment efficaces énergétiquement pour permettre, par exemple, d'utiliser un CNN sur un système embarqué. Dans cette optique, les techniques de spéculation temporelle peuvent encore améliorer l'efficacité de ces implémentations.

3 Spéculation temporelle

Après une présentation du contexte dans lequel elle s'inscrit, cette section introduit les techniques de spéculation temporelle. Les modèles probabilistes d'erreur sont discutés en fin de section.

3.1 Contexte

Dans les circuits électroniques, les signaux binaires sont représentés par deux plages de tension qui ne se chevauchent pas, entre 0V et la tension d'alimentation V_{dd} du circuit. Ces signaux suivent des « chemins de données » qui traversent, entre autres, des portes logiques. Chaque porte logique, réalisée avec des transistors, effectue une opération élémentaire. Le délai entre un changement d'état de l'entrée d'une porte logique et le changement d'état de sa sortie est non nul et est appelé temps de propagation. Le temps de propagation dépend, entre autres, de la V_{dd} : lorsque celle-ci est plus faible, les transistors commutent moins vite et le temps de propagation est plus grand. La somme des temps de propagation des portes d'un chemin de données est une borne inférieure de la période avec laquelle on peut utiliser ce chemin. Le chemin le plus long, en termes de temps de propagation, est appelé chemin critique. Son temps de propagation est une borne inférieure de la période du circuit et, donc, une borne supérieure de sa fréquence de fonctionnement. Si V_{dd} diminue, cette borne de fréquence diminue donc. Si la fréquence n'est pas aussi réduite, alors des erreurs temporelles peuvent apparaître : le circuit ne fonctionne plus correctement. Ces erreurs sont donc les mêmes que celles apparaissant lorsque la fréquence de fonctionnement d'un circuit est augmentée au-delà de sa limite (*overclocking*). Elles sont dites *soft* car elles altèrent seulement une donnée et non le système complet.

L'avantage de baisser V_{dd} d'un circuit est de réduire sa consommation énergétique, et donc son efficacité énergétique. En effet, la consommation d'énergie est proportionnelle au carré de la tension d'alimentation. Le ratio du nombre de calculs par seconde sur l'énergie consommée peut donc être augmenté.

Par exemple, un additionneur n bits peut être réalisé simplement à l'aide de n additionneurs complets branchés en série. Chaque additionneur complet calcule, pour deux bits d'opérandes, leur somme sur deux bits (résultat et retenue). Ils sont appelés *ripple carry adder* (RCA) car la retenue est simplement propagée du poids faible vers le poids fort. Le chemin critique de l'additionneur est donc la propagation de retenue, des opérandes de poids faible à la retenue finale de l'addition. La figure 4 présente un additionneur RCA 4 bits, mais le même principe peut être appliqué à des additionneurs 32 bits, 64 bits, etc.

Si le temps de propagation du chemin critique d'un tel additionneur n'est pas respecté, l'information de retenue n'aura pas le temps d'être prise en compte pour le calcul des bits de poids fort. Le résultat de l'addition pourra donc être faussé selon les valeurs des différentes retenues intermédiaires. L'apparition de l'erreur dépend des opérandes, selon si la prise en compte d'une retenue intermédiaire change le résultat ou non. Les erreurs sont rares mais sont souvent importantes car ce sont souvent les bits de poids forts qui sont erronés.

3.2 Dans le cadre des processeurs

Sartori et al. [17] ont montré que la distribution des erreurs dépend directement de la distribution de l'activité d'un processeur. En effet, certaines opérations utilisent des chemins plus courts que le chemin critique du processeur. Elles ne génèrent donc pas d'erreur temporelles. Les optimisations de

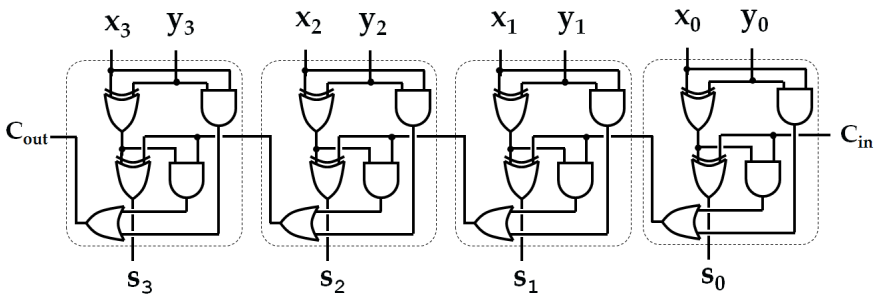

FIGURE 4 – Additionneur RCA 4 bits. Les lignes pointillées délimitent les quatre additionneurs complets. Le résultat $s_{3..0}$ est l'addition des opérandes $x_{3..0}$ et $y_{3..0}$. La retenue est c_{out}. La retenue entrante c_{in} permet de chaîner plusieurs additionneurs

compilations modifient l'exécutable final et, par conséquent, modifient l'activité du processeur. Ainsi, on peut tirer parti de ces optimisations pour éviter d'utiliser les chemins critiques du processeur et réduire, voire éliminer, les erreurs temporelles.

Par exemple, ils ont montré qu'un programme avec plus de dépendances RAW (*Read-After-Write*) génère plus d'erreurs temporelles sur leur plateforme d'expérimentation (architecture FabScalar). Cette architecture superscalaire exécute les instructions dans plusieurs *pipelines*, et réorganise donc les instructions exécutées (*out-of-order exection*). Lorsqu'une instruction dépend du résultat de l'instruction précédente, l'exécution de cette instruction doit être retardée afin de ne pas casser la dépendance RAW. L'*operand forwarding* est une amélioration des *pipelines*, implémentée dans FabScalar. Elle permet de ne pas attendre la fin de l'exécution de l'instruction précédente pour obtenir son résultat, l'opérande manquante, et donc d'éviter ce retard. Les dépendances RAW amènent donc le processeur à utiliser ce mécanisme. Le processeur doit détecter ces dépendances entre les accès mémoires. Cela est fait par son module LSU (*Load Store Unit*), chargé de transférer les données entre sa mémoire et ses registres. Or, la gestion du *forwarding* est sur le chemin critique du LSU, ce qui explique que les dépendances RAW amène le processeur spéculatif à faire plus d'erreur.

La technique d'optimisation des boucles *loop splitting* permet d'éliminer des dépendances. Cette transformation permet donc de limiter l'utilisation du *forwarding* et donc d'obtenir un programme générant moins d'erreur. De manière semblable, d'autres transformations classiques (déroulage de boucles, fusion de boucles) permettent aussi d'obtenir des programmes générant moins d'erreurs.

La compilation d'un programme peut donc être optimisée pour les processeurs spéculatifs, en générant des programmes évitant au processeur d'utiliser ses chemins critiques.

Ernst et al. [5] étendent l'ajustement dynamique de la tension (*Dynamic Voltage Scaling*, DVS) des processeurs afin d'augmenter son efficacité énergétique par un mécanisme appelé *Razor*. Le principe du DVS est de baisser la tension d'alimentation du processeur (ainsi que sa fréquence) lorsque l'utilisation de celui-ci est faible afin de consommer moins d'énergie. Les différents niveaux de tension sont calculés statiquement pour garantir le bon fonctionnement du processeur même dans les pires conditions d'environnement. Pourtant, ces conditions surviennent très rarement. Les niveaux de tension ont donc, la plupart du temps, des marges importantes avec les tensions idéales. L'idée de *Razor* pour pallier ce problème est de modifier la tension dynamiquement en fonction du taux d'erreurs généré. Cela permet d'enlever toute marge inutile, mais nécessite de détecter et corriger les erreurs.

3.3 Dans le cadre des circuits logiques programmables

Les circuits logiques programmables sont des circuits intégrés dont le fonctionnement peut être reconfiguré. Les FPGA, une architecture de circuits logiques programmables, sont composés de blocs logiques, d'opérateurs (e.g. multiplieurs), de mémoires volatiles et non volatiles, de générateurs d'horloge, etc. La reconfiguration modifie les opérations logiques des blocs ainsi que les connexions entre tous les composants. Il ne s'agit donc pas d'un processeur, bien qu'on puisse les configurer pour agir comme tel. La synthèse logique est l'étape de compilation de la description matérielle abstraite du comportement voulu en une configuration concrète pour une cible donnée. Cette synthèse passe par une étape d'analyse statique temporelle, basée sur les temps de propagation, calculant la fréquence d'utilisation maximale du circuit.

Cette analyse est très conservative afin d'éviter de générer un circuit dont l'exécution produirait des erreurs temporelles, même dans des conditions de fonctionnement défavorables. Cela conduit à une différence importante entre la fréquence prédite par l'analyseur et la fréquence ne produisant pas d'erreurs temporelles en pratique, lorsque les conditions de fonctionnement sont normales [7]. Comme expliqué dans la section 3.1, cette marge peut profiter à une réduction de la tension d'alimentation plutôt qu'à un accroissement de la fréquence.

Le principe de *Razor* est techniquement applicable aux FPGA [12]. Cependant, sa réalisation n'est pas aisée car elle ne peut pas être automatisée avec les outils classiques de description de matériel.

3.4 Modèle probabiliste d'erreur

Un modèle probabiliste d'erreur permet de connaitre le comportement du système considéré et de comparer plusieurs approches.

Par exemple, dans le cadre de la conception de circuits devant respecter une contrainte de latence maximale, une approche courante est de réduire la longueur des mots (le nombre de bits) traités. Cela permet de réduire la longueur du chemin critique, ce qui permet de respecter la contrainte de latence. Si on retire les bits de poids fort, alors l'amplitude des nombres représentables est plus faible. Si on retire les bits de poids faible, alors les nombres sont moins précis et ont une erreur de quantification plus importante. C'est souvent cette deuxième solution qui est choisie car l'erreur est systématique mais moins importante qu'un dépassement.

Shi et al. [18] ont montré que l'espérance de l'erreur due à la troncature nécessaire pour respecter la contrainte de latence dans le résultat d'un RCA est :

$$E_{trad} = 2^{-b+1} - 2^{-k}$$

Avec k le nombre initial de bits et $b-1$ le nombre de bits après troncature. En gardant le nombre de bits k mais en augmentant la fréquence d'utilisation du RCA, la propagation de retenue aura lieu sans erreur seulement sur les b premiers bits. L'espérance de l'erreur est donc dans ce cas :

$$E_{new} = 2^{-b} - 2^{-k-1}$$

L'espérance de la seconde solution est deux fois plus petite que l'approche classique :

$$\frac{E_{new}}{E_{trad}} = \frac{2^{-b} - 2^{-k-1}}{2^{-b+1} - 2^{-k}} = \frac{1}{2}$$

Intuitivement, cela est dû au compromis entre la fréquence d'apparition des erreurs et leur amplitude.

Ils ont montré qualitativement que le rapport signal sur bruit dans le cas du surcadençage est inférieur à celui dans le cas de la perte de précision sur une application de traitement d'image simple.

Si, dans le cas du traitement d'image, l'espérance de l'erreur peut être une bonne métrique, ce n'est pas le cas pour les CNN. En effet, les erreurs importantes, même rares, peuvent avoir un impact considérable sur le bon fonctionnement du réseau de neurones. À cause des non-linéarités (sous-échantillonnage et fonction d'activation), une grande erreur peut faire basculer le résultat du réseau. Les petites erreurs sont, par contre, peu importantes car les CNN sont résistants au bruit par essence. Il est donc important de garantir que les erreurs de grande amplitude n'auront pas lieu. Or, la spéculation temporelle génère justement ce type d'erreur. Une solution est donc de permettre ces erreurs, mais de les détecter pour les corriger.

Une des solutions pour accomplir cela est la tolérance aux fautes au niveau algorithmique.

4 Tolérance aux fautes au niveau algorithmique

Cette section débute par une présentation générale des techniques de détection et de correction d'erreurs de calcul. Les principes généraux de la tolérance aux fautes au niveau algorithmique sont ensuite expliqués. Enfin, des exemples de schémas de tolérance aux fautes pour différents algorithmes sont présentés.

4.1 Outils de détection et correction d'erreur

La détection et la correction d'erreur, aussi appelée tolérance aux fautes, sont toujours réalisées grâce à une redondance des données ou des calculs. La solution la plus simple est appelée *triple modular redundancy* (TMR). Elle consiste à tripliquer la réalisation d'une tâche puis d'effectuer un vote par majorité sur les trois résultats. De cette manière, si un des résultats est faux, l'erreur est masquée par les deux autres résultats. Cette solution est souvent implémentée matériellement, simplement en triplant les composants.

Reis et al. [15] proposent une approche basée sur ce concept, mais de manière logicielle. Autrement dit, la triplication n'a pas lieu dans l'espace (en multipliant le matériel) mais dans le temps (en réalisant trois fois le calcul). Ils profitent de ressources ILP (*Instruction-Level Parallelism*) inutilisées pour réduire le coût de cette redondance.

Ernst et al. [5] proposent, pour *Razor*, de réaliser un double échantillonnage des signaux sur les chemins critiques. Cela est réalisé grâce à une seconde horloge cadencée à la même fréquence que l'horloge principale mais ayant un léger retard. Des circuits supplémentaires peuvent ainsi comparer les valeurs des signaux échantillonnés au top de l'horloge principale avec celles des signaux échantillonnés au top de l'horloge retardée. Si une différence existe, alors le temps de propagation du circuit logique impliqué était trop long car son résultat a changé entre les deux échantillonnages. L'échantillon sur l'horloge principal est donc faux. L'erreur est donc détectée et peut être corrigée en restaurant le programme dans un état normal. La tension d'alimentation pourra ensuite être corrigée afin d'éviter d'autres erreurs.

Détecter et corriger les erreurs à un niveau d'abstraction plus haut, c'est-à-dire au niveau algorithmique plutôt qu'au niveau des instructions, permet de réduire considérablement le coût [8]. Cette approche se base sur des propriétés mathématiques du calcul effectué. Les résultats de cette approche sont donc spécifiques, mais très efficaces.

4.2 Principes

L'idée fondamentale de la tolérance aux fautes au niveau algorithmique est d'encoder les données et de changer légèrement l'algorithme considéré afin qu'il puisse traiter les données encodées. La sortie de l'algorithme modifié est aussi encodée. Le résultat encodé du calcul, l'information utile de sa sortie, est récupéré après la phase de détection et correction d'erreur.

Si les données encodées respectent une propriété choisie, qu'on appelle invariant, et que cette propriété est préservée lors du calcul, alors le résultat de l'algorithme doit aussi respecter cette propriété. Si l'invariant n'est pas respecté, alors au moins une erreur a eu lieu. Cependant, des cas faux positifs et faux négatifs peuvent survenir.

En utilisant ces principes, on peut obtenir un résultat *ad hoc* pour un algorithme considéré. Cette méthode n'est pas générale mais permet, lorsqu'elle est applicable, d'être très peu coûteuse.

Un des défauts de cette approche est que la détection d'erreur a lieu après l'exécution de l'algorithme. Une détection tardive implique des calculs inutiles et une efficacité amoindrie. Ce fonctionnement est dit « hors-ligne ». Les algorithmes itératifs ont l'avantage de permettre la vérification « en ligne » de l'invariant à chaque étape de calcul (e.g. [16, 1]). Afin de détecter l'erreur le plus tôt possible, Chen [3] propose une approche nommée Online-ABFT.

Un autre défaut est que cette tolérance aux fautes n'a lieu que sur le cœur de l'algorithme. Toutes les autres phases d'un programme ne sont pas contrôlées. Dans un programme complet, elle doit donc être couplée avec d'autres mécanismes de tolérance aux fautes [1].

Il y a un défaut supplémentaire lorsque l'algorithme utilise des opérations en virgule flottante car celles-ci ne sont pas associatives. Cela rend la vérification d'invariants plus délicate dans ce cas. Souvent, un seuil de tolérance est appliqué pour permettre les erreurs d'arrondi.

Roy-Chowdhury et al. [16] proposent une nouvelle approche pour ce problème. La marge de tolérance est couramment évaluée expérimentalement à partir d'un échantillon limité de données. Cela rend la détection d'erreur sujette à des faux positifs et des faux négatifs. Ils proposent de calculer la borne supérieure de l'erreur effectuée en fonction de l'accumulation de l'erreur de chaque opération en virgule flottantes utilisée. En effet, l'erreur d'une opération en virgule flottante est bornée par 2^{-t}, où t est la taille de la mantisse. Autrement dit, pour une opération en virgule flottante notée $\oplus$, le résultat z est :

$$z = (x \oplus y)(1 + \delta)$$

où $|\delta| \leqslant 2^{-t}$ et l'erreur est :

$$err(z) = (x \oplus y)\delta$$

4.3 Application visées

Le cas des systèmes linéaires est propice à l'ABFT comme les opérations sur les matrices (multiplication, addition, produit scalaire, décomposition LU) [8]. D'autres exemples sont la méthode du pivot de Gauss ; la transformation de Fourier rapide ; la décomposition QR d'une matrice ; la décomposition en valeurs singulières ; des méthodes itératives pour résoudre des équations aux dérivées partielles [16] ou des systèmes linéaires creux [16].

Dans le cas de l'algorithme de transformation de Fourier rapide, on peut par exemple utiliser le théorème de Parseval [13]. Il stipule que l'énergie d'un signal temporel est égale à l'énergie du

```
for(i = 0; i < N ; i++) {
  t += x[i] * x[i]; // domaine temporel
  f += X[i] * X[i]; // domaine fréquentiel
}
if(fabs(t * N - f) < epsilon)
  // erreur
```

FIGURE 5 – Programme de vérification d'un invariant pour une transformation de Fourier

même signal dans le domaine fréquentiel, c'est-à-dire :

$$N \sum_{j=0}^{N-1} |x(j)|^2 = \sum_{j=0}^{N-1} |X(j)|^2.$$

où x est le signal discret temporel et X sa transformée de Fourier discrete.

Cet invariant peut être vérifiée à la fin du calcul par le programme simple de la figure 5. La comparaison des calculs se fait avec une marge de tolérance ϵ à cause des erreurs d'arrondi.

Cette méthode est très simple, mais elle permet uniquement la détection d'erreur. Elle n'indique pas quelle partie du calcul contient une erreur et ne permet pas de correction. En plus, cette détection à lieu seulement à la fin du calcul.

Ce constat est le même pour les résolutions d'équations : le résultat peut être validé simplement en vérifiant si la solution satisfait les équations, mais aucune correction n'est possible.

4.4 Exemples concrets

4.4.1 Multiplication matricielle

Cette section présente l'ABFT dans le cas de la multiplication de matrices comme expliqué par Huang et Abraham dans leur travail séminal [8].

La matrice opérande gauche G de taille $m \times n$ est encodée en une matrice, notée G^c, de taille $(m+1) \times n$. Ses m premières lignes sont égales aux m premières lignes de la matrice G. Les valeurs de sa dernière ligne $m+1$ sont les sommes sur les colonnes des lignes précédentes (sommes de contrôle, ou *checksum* des colonnes). Plus formellement, si on note $a_{i,j}$ la valeur de la ligne i et de la colonne j de la matrice A :

$$g^c_{i,j} = \begin{cases} g_{i,j} & \text{si } i \leqslant m \\ \sum_{k=1}^{m} g_{k,j} & \text{si } i = m+1 \end{cases}$$

De manière similaire, la matrice opérande droite D de taille $n \times p$ est encodée en une matrice, notée D^l, de taille $n \times (p+1)$. Ses p premières colonnes sont égales aux p premières colonnes de la matrice D. Les valeurs de sa dernière colonne $p+1$ sont les sommes sur les lignes des colonnes précédentes. Plus formellement :

$$d^l_{i,j} = \begin{cases} d_{i,j} & \text{si } j \leqslant p \\ \sum_{k=1}^{p} d_{i,k} & \text{si } j = p+1 \end{cases}$$

Le résultat du produit matriciel $G^c \times D^l$, noté R^t, est une matrice de taille $(m+1) \times (p+1)$, comme illustré sur la figure 6. Elle contient :

— le résultat de la multiplication $R = G \times D$ dans ses m premières lignes et ses p premières colonnes ;

— les sommes sur les lignes des colonnes 1 à m dans sa dernière ligne ;

— les sommes sur les colonnes des lignes 1 à p dans sa dernière colonne.

La valeur $G^t_{m+1,p+1}$ est donc la somme de la ligne de sommes, ainsi que la somme de la colonne de sommes. Plus formellement :

$$r^t_{i,j} = \sum_{k=1}^{n} g^c_{i,k} \times d^l_{k,j} = \begin{cases} r_{i,j} & \text{si } i \leqslant m \wedge j \leqslant p \\ \sum_{k=1}^{p} r_{i,k} & \text{si } j = p+1 \wedge i \leqslant m \\ \sum_{k=1}^{m} r_{k,j} & \text{si } i = m+1 \wedge j \leqslant p \\ \sum_{k=1}^{p} r^t_{m+1,k} = \sum_{k=1}^{m} r^t_{k,p+1} & \text{si } i = m+1 \wedge j = p+1 \end{cases}$$

Le résultat du calcul, l'information utile R, est donc directement inclus dans le résultat encodé. Les sommes constituent les données redondantes permettant le tolérance aux fautes (détection et correction d'une erreur).

En effet, d'après la construction ci-dessus, le résultat d'un produit matriciel encodées effectué sans erreur doit respecter l'invariant :

$$\bigwedge \begin{array}{l} \forall i \in [1,m], r_{i,p+1} = \sum_{k=1}^{p} r_{i,k} \\ \forall j \in [1,p], r_{m+1,j} = \sum_{k=1}^{m} r_{k,j} \end{array}$$

Si ce n'est pas le cas, alors au moins une erreur a eu lieu. L'erreur peut être survenue dans le calcul matriciel ou dans le calcul de sommes de contrôle. En vérifiant l'invariant, on peut savoir pour quelles lignes i et quelles colonnes j le calcul de somme est erroné. Selon ces vérifications, quatre cas sont possibles :

1. Une seule ligne i et une seule colonne j sont fautives, alors la valeur $R^t_{i,j}$ à l'intersection de cette ligne et de cette colonne est erronée. Elle peut être corrigée en lui ajoutant la différence entre la somme de la ligne (ou la colonne) concernée et la somme de contrôle de cette ligne (ou cette colonne) : $(\sum_{k=1}^{p} R_{i,k} - R_{i,p+1})$;
2. Une seule ligne i est fautive, alors la somme de contrôle de cette ligne $R^t_{i,p+1}$ est erronée. Elle peut être corrigée, mais ne fait pas partie de l'information utile du résultat ;
3. Une seule colonne j est fautive, alors la somme de contrôle de cette colonne $R^t_{m+1,j}$ est erronée ;
4. Plusieurs lignes ou colonnes sont fautives. Dans ce cas, il y a plus d'une erreur et ce mécanisme, bien que suffisant pour les détecter, est insuffisant pour les corriger.

Des faux négatifs peuvent survenir, par exemple si quatre erreurs s'annulant ont lieu à l'intersection de deux colonnes et deux lignes.

En pratique, dans le cadre de calculs sur les entiers, il faut utiliser l'arithmétique modulaire afin d'éviter un dépassement inattendu pouvant conduire l'invariant à être erroné. Avec des mots de taille r bits, tous les calculs de somme peuvent être réalisé modulo 2^r. Les propriétés vues restent vraies. Dans le cadre de calculs sur des nombres à virgules flottantes, l'égalité utilisée dans l'invariant doit être remplacée par une distance comparée à un seuil. En effet, des erreurs d'arrondi peuvent provenir du changement dans l'ordre des opérations et une tolérance est nécessaire.

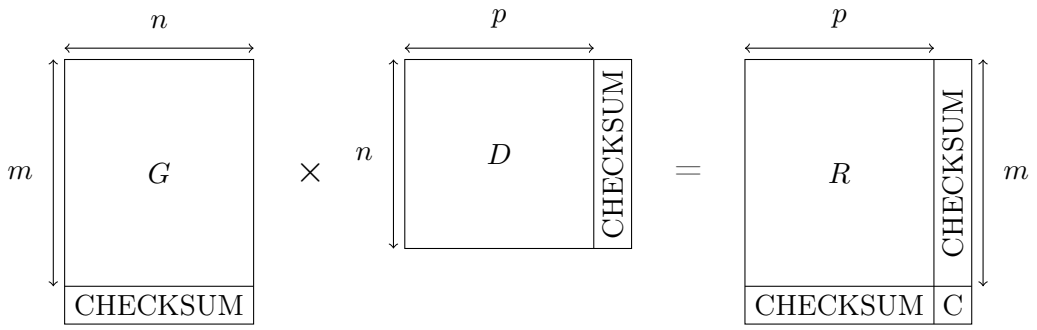

FIGURE 6 – Multiplication de deux matrices encodées

Le coût de cette méthode est simplement dû à l'augmentation de la taille des matrices traitées ainsi que la vérification de l'invariant dans le résultat. La matrice résultat encodée, R^t, a $p+m+1$ valeurs en plus que l'information utile R.

Dans le cas étudié par Huang et Abraham, les multiplications de matrices sont effectuées sur une matrice de processeurs connectés (*mesh-connected processor array*). Celle-ci peut donc être réalisée avec n^2 processeurs en n unités de temps [8]. La tolérance aux fautes peut être réalisée avec l'ajout de $2n+1$ processeurs et nécessite $2 \times k \times log_2(n)$ unités de temps supplémentaires. k est le ratio entre le temps nécessaire pour réaliser une addition et celui pour une multiplication. Le ratio de redondance est donc $2/n$ et $2 \times k \times log^2(n)/n$, respectivement pour le nombre de processeurs et le temps. En comparaison, la solution basique TMR a pour ratio matériel 3.

On peut noter que les ratios de redondance tendent asymptotiquement vers zéro pour des grandes matrices. Cependant, cette solution permettant de corriger une seule erreur, il est préférable de partitionner les matrices multipliées. Le même mécanisme de sommes de contrôle peut ainsi être appliquée à chaque partition et plusieurs erreurs peuvent être corrigées (une par partition).

Dans un cas plus classique, avec une multiplication matricielle (naïve) en temps $O(n^3)$, l'impact de cette méthode est tout de même très inférieur à la méthode TMR.

4.4.2 Produit de convolution

Cette section présente un travail préliminaire réalisé dans le cadre de ce stage. Ce travail établit un schéma de tolérance aux fautes au niveau algorithmique applicable au produit de convolution.

Le produit de convolution unidimensionnel discret, noté $*$, est un opérateur binaire défini pour deux fonctions f et x :

$$y[t] = (f * x)[t] = \sum_{k=-\infty}^{\infty} f[k]x[t-k]$$

Si la fonction f a un support fini $[-M, M]$, alors :

$$y[t] = (f * x)[t] = \sum_{k=-M}^{M} f[k]x[t-k]$$

Souvent, comme par exemple en traitement d'image ou en traitement du signal, l'une des opérandes est constante et l'autre correspond aux données traitées. L'opérande constante est appelée filtre ou

noyau de convolution. L'autre opérande est alors le signal d'entrée, et le résultat de l'opération le signal de sortie. $N = 2M + 1$ est appelé l'ordre du filtre. L'opération est commutative, associative et distributive pour l'addition.

On peut considérer le produit de convolution comme une multiplication matricielle, comme montré sur la figure 7. L'opérande gauche correspond au filtre f de taille N, dans une matrice de Toeplitz : ses diagonales sont constantes. La première ligne de la matrice contient les coefficients du filtre, du dernier (N) au premier (1), puis des zéros. Chacune des lignes suivantes contiennent les mêmes valeurs, mais décalées vers la droite de manière circulaire. Toutes les lignes de la matrice contiennent donc les mêmes valeurs décalées d'un pas de 1 par ligne (représenté par les barres obliques dans la figure). L'opérande droite est le vecteur de données filtrées, dans l'ordre. Il contient n valeurs. Le vecteur résultat est le résultat de la convolution, de $y[1]$ à $y[n - N + 1]$. Il contient $n - N + 1$ valeurs.

Comme expliqué dans la section 4.4.1, si l'on rajoute une ligne de sommes de contrôle à l'opérande gauche, la matrice résultat aura une ligne de sommes de contrôle. Ici, il ne s'agit que d'une seule valeur car l'opérande droite est un vecteur. En revanche, on n'ajoute pas de colonne de sommes de contrôle à l'opérande droite. Comme cette opérande est un vecteur, cela reviendrait à simplement dupliquer les calculs. La vérification de la somme de contrôle dans le résultat permet simplement de détecter une erreur, pas de la corriger.

La régularité des opérandes de ce produit matriciel peut profiter à son calcul ainsi qu'à la vérification des résultats. En effet, comme la matrice est multipliée par un vecteur, toutes les valeurs d'une colonne de la matrice sont multipliées par la même valeur provenant du vecteur. La somme des colonnes de la matrice de Toeplitz (les sommes de contrôle) sont, pour la partie centrale de la matrice dans laquelle chaque colonne contient tous les coefficients (notée A sur la figure), toutes égales. Cela signifie que, dans la somme de contrôle finale, les valeurs de la partie centrale du vecteur (de $N+1$ à $n-N$ compris) sont toutes multipliées par la même constante : la somme des coefficients du filtre.

Cette somme de contrôle est donc égale à :

$$\sum_{i=1}^{N} \left(x[i] \sum_{k=N}^{i} f[k] \right) + \sum_{i=N+1}^{n-N} \left(x[i] \sum_{k=1}^{N} f[k] \right) + \sum_{i=1}^{N} \left(x[n-N+i] \sum_{k=1}^{i} f[k] \right)$$

Si on note σ la somme des coefficients du filtre, on a :

$$\sum_{i=1}^{N} \left(x[i] \sum_{k=N}^{i} f[k] \right) + \sigma \sum_{i=N+1}^{n-N} x[i] + \sum_{i=1}^{N} \left(x[n-N+i] \sum_{k=1}^{i} f[k] \right)$$

Le premier terme correspond à la partie notée A sur la figure, le deuxième à la partie B et le troisième à la partie C. Selon les valeurs de N et n, la partie B est plus ou moins grande.

Cette particularité permet de calculer la somme de contrôle plus efficacement sur la partie B qu'avec une multiplication matricielle.

Cette méthode permet donc de détecter une erreur dans un produit de convolution assez efficacement, mais pas de les détecter. Comme dans d'autres schémas de tolérance aux fautes au niveau algorithmique [16], le calcul doit être renouvelé pour corriger l'erreur.

Il faut éviter de renouveler le calcul de convolution entièrement car les performances seraient alors très faibles en cas d'erreur. Une solution est de fractionner le calcul de convolution. Ainsi,

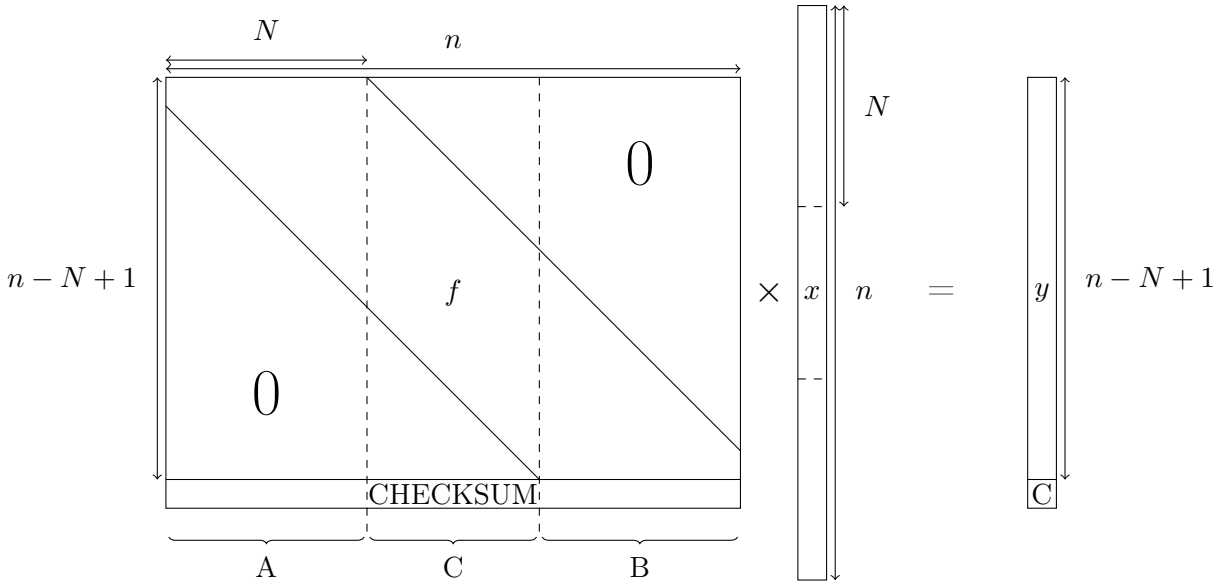

FIGURE 7 – Produit de convolution représenté comme une multiplication matricielle

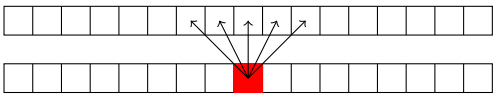

FIGURE 8 – Dépendance de données dans un calcul de convolution pour la valeur représentée en rouge pour un filtre d'ordre 5

lorsqu'une erreur est détectée dans une fraction, seule cette fraction doit être recalculée. Cependant, fractionner les convolutions provoque des calculs redondants. En effet, dans le cadre d'un filtre d'ordre fini, chaque nouvelle valeur du signal est calculée à partir de son ancienne valeur est des anciennes valeurs voisines. La figure 8 illustre ces dépendances. Pour calculer les extrémités du vecteur résultat, le vecteur opérande doit donc être plus grand. La redondance est nécessaire pour calculer les extrémités des fractions. Il y a donc un compromis entre le temps de calcul nécessaire lorsqu'une erreur est détectée et la redondance des calculs. Si les fractions sont larges, recalculer une fraction quand une erreur de calcul est détectée coûte cher. Si elles sont plus petites, il y a plus de redondances. Zhou et al. [21] proposent une solution à ce type de compromis.

Cette méthode peut être étendue aux convolutions 2D et 3D. Elle peut donc être utilisée dans une implémentation de CNN dont les calculs peuvent être erronés, comme dans le cas de la spéculation temporelle.

La section suivante présente les objectifs de ce stage, basé sur les concepts présentés dans les sections précédentes.

5 Objectifs du stage et contributions attendues

L'objectif majeur de ce stage est d'explorer une nouvelle approche pour améliorer l'efficacité des accélérateurs matériels pour réseaux de neurones convolutifs. La cible d'expérimentation est donc

une plateforme FPGA. La plateforme sera programmée grâce à un outil de synthèse de haut niveau. Ces outils permettent de générer une description matérielle à partir d'un algorithme décrit à un niveau plus abstrait, par exemple en langage C.

L'idée principale est d'améliorer l'efficacité énergétique de l'accélérateur matériel grâce à la spéculation temporelle. Cependant, celle-ci génère des erreurs sporadiques de grande amplitude. Les CNN étant résistants au bruit mais pas aux erreurs de grande amplitude, celles-ci doivent être détectées et corrigées. L'approche choisie pour cela est la tolérance aux fautes au niveau algorithmique. Cette méthode permet, lorsqu'elle est applicable, d'obtenir une tolérance aux fautes avec un surcout faible.

Comme indiqué dans la section 4.4.2, un tel schéma de tolérance aux fautes pour les convolutions, opération élémentaire utilisée dans les CNN, existe. Ce schéma est plus simple que des travaux antérieurs (e.g. [14]) et n'a pas été, à notre connaissance, déjà traité.

Les CNN ont recours à de nombreuses convolutions avec des paramètres différents : nombre de dimensions, taille des données traitées, etc. Si les premiers résultats sont concluants, nous généraliserons l'approche grâce à un compilateur source à source. Combiné avec un outil de synthèse de haut niveau, il permettra de générer automatiquement la tolérance aux fautes en fonction des paramètres de chaque convolution. Cette automatisation permettra d'explorer plus facilement les différentes possibilités d'amélioration : pavage de boucles, compromis entre la précision de la détection d'erreur et le surcout, etc.

Références

[1] George Bosilca, Aurelien Bouteiller, Thomas Herault, Yves Robert, and Jack Dongarra. Composing resilience techniques : Abft, periodic and incremental checkpointing. *International Journal of Networking and Computing*, 5(1) :2–25, 2015.

[2] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience : 2014 update. *Supercomputing frontiers and innovations*, 1(1) :5–28, 2014.

[3] Zizhong Chen. Online-abft : An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *ACM SIGPLAN Notices*, volume 48, pages 167–176. ACM, 2013.

[4] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *International Conference on Artificial Neural Networks*, pages 281–290. Springer, 2014.

[5] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, et al. Razor : A low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 7–18. IEEE, 2003.

[6] Clément Farabet, Berin Martini, Polina Akselrod, Selçuk Talay, Yann LeCun, and Eugenio Culurciello. Hardware accelerated convolutional neural networks for synthetic vision systems. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 257–260. IEEE, 2010.

[7] Benjamin Gojman, Sirisha Nalmela, Nikil Mehta, Nicholas Howarth, and André DeHon. Groklab : 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*, pages 81–90. ACM, 2013.

[8] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6) :518–528, 1984.

[9] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv :1408.5882*, 2014.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[11] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition : A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1) :98–113, 1997.

[12] Rengarajan Ragavan, Cedric Killian, and Olivier Sentieys. Adaptive overclocking and error correction based on dynamic speculation window. In *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, pages 325–330. IEEE, 2016.

[13] A. L. Narasimha Reddy and Prithviraj Banerjee. Algorithm-based fault detection for signal processing applications. *IEEE Transactions on Computers*, 39(10) :1304–1308, 1990.

[14] G Robert Redinbo. System level reliability in convolution computations. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(8) :1241–1252, 1989.

[15] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. Swift : Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005.

[16] Amber Roy-Chowdhury, Nikolas Bellas, and Prithviraj Banerjee. Algorithm-based error-detection schemes for iterative solution of partial differential equations. *IEEE Transactions on Computers*, 45(4) :394–407, 1996.

[17] John Sartori and Rakesh Kumar. Compiling for energy efficiency on timing speculative processors. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1297–1304. IEEE, 2012.

[18] Kan Shi, David Boland, and George A Constantinides. Accuracy-performance tradeoffs on an fpga through overclocking. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 29–36. IEEE, 2013.

[19] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587) :484–489, 2016.

[20] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[21] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H Kuhn, Yang Ni, and David Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 207–218. ACM, 2012.