Titre: Title:	Implémentation évolutive à faible complexité de circuits de multiplication par matrices constantes
Auteur: Author:	Aymen-Alaeddine Zeghaida
Date:	2025
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Zeghaida, A.-A. (2025). Implémentation évolutive à faible complexité de circuits de multiplication par matrices constantes [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/68207/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/68207/
Directeurs de recherche: Advisors:	Jean Pierre David, & J. M. Pierre Langlois
Programme: Program:	Génie électrique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Implémentation évolutive à faible complexité de circuits de multiplication par matrices constantes

AYMEN-ALAEDDINE ZEGHAIDA
Département de génie éléctrique

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie éléctrique

Août 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Implémentation évolutive à faible complexité de circuits de multiplication par matrices constantes

présenté par **Aymen-Alaeddine ZEGHAIDA**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

François LEDUC-PRIMEAU, président
Jean Pierre DAVID, membre et directeur de recherche
Pierre LANGLOIS, membre et codirecteur de recherche
Tarek OULD-BACHIR, membre

DÉDICACE

Ce mémoire est dédié à ma famille et à tous ceux qui ont rendu ce parcours possible.
À quiconque choisit de poursuivre le savoir avec curiosité, même dans l'adversité.
"Per aspera, ad astra... "

REMERCIEMENTS

Je tiens à exprimer ma profonde gratitude à mon directeur de recherche, Prof. Jean-Pierre David, pour sa confiance, sa rigueur et la générosité avec laquelle il a partagé son savoir. Son expertise, sa clarté conceptuelle et sa capacité à transformer des problèmes complexes en idées maîtrisables ont profondément marqué mon approche. Sous sa direction, j'ai appris à définir des objectifs précis, à développer des solutions par itérations, et à progresser sans craindre l'imperfection. Il rappelait souvent que «le mieux est l'ennemi du bien» - un principe simple, mais déterminant, qui m'a permis de débloquer bien des étapes. Son encadrement a été une véritable école, intellectuelle autant que personnelle.

Je remercie également Prof. Pierre Langlois, mon codirecteur, pour son exigence méthodologique et sa grande disponibilité. Ses commentaires détaillés et ses conseils avisés ont contribué de manière décisive à la qualité de ce travail. À travers son engagement et sa rigueur, il m'a poussé à affiner mes idées, à structurer mes démarches et à maintenir un haut niveau d'exigence. Je le remercie, avec Monsieur David, pour le soutien financier apporté au projet.

Je remercie tout particulièrement Dinesh pour son aide généreuse dans l'implémentation de l'algorithme et ses retours pertinents, qui m'ont permis d'améliorer efficacement mon travail.

Ce parcours aurait été bien plus difficile sans la présence de mes collègues de laboratoire et amis. À mes camarades de laboratoire Loïc et Fahimeh, pour leur camaraderie dans le quotidien du laboratoire. À Charles, Yunjo, Karou et Mariem, avec qui je partage beaucoup de moment agréables. Leur bonne humeur m'a permis de garder le cap.

Je suis profondément reconnaissant envers ma famille : à ma mère Saida, pour son courage, sa patience et ses sacrifices silencieux - j'espère t'avoir rendu un peu de ta fierté ; à ma sœur Hind, pour sa présence et son soutien émotionnel indéfectible, toujours capable d'alléger mes journées et d'apporter de la bonne humeur, même dans les moments les plus difficiles ; à mon frère Imad, dont les conseils avisés et l'esprit aventurier restent pour moi une source d'inspiration. Je lui souhaite un avenir à la hauteur de ses rêves.

À la mémoire de mon cher père, Mohammed Salah, qui vivra à jamais dans mon cœur. Son souvenir m'accompagne à chaque étape.

RÉSUMÉ

La multiplication d'une matrice constante par un vecteur constitue une opération de base dans de nombreuses applications embarquées et en temps réel, qui reposent sur l'accélération matérielle, notamment en traitement du signal numérique, en intelligence artificielle embarquée et en commande en temps réel.

Sur FPGA, les approches classiques s'appuient sur des multiplieurs généralistes, coûteux en ressources logiques et peu efficaces sur le plan énergétique. Ce surcoût devient particulièrement contraignant dans les scénarios de calcul en périphérie, où les contraintes de surface, de puissance et de latence sont strictement encadrées.

Cette recherche propose un algorithme évolutif à faible complexité, conçu pour éviter l'usage de multiplieurs matériels en exploitant l'arithmétique binaire fondée sur les décalages et les additions, ainsi que la réutilisation de résultats intermédiaires. La méthode repose sur une factorisation récursive des sous-expressions communes entre les lignes de la matrice, ce qui permet de réduire le nombre total d'additions et de produire un graphe de calcul de profondeur minimale.

Trois implémentations cibles sur circuit logique programmable sont évaluées à partir de cet algorithme : une version purement combinatoire, une version segmentée en étapes (pipeline), et une version segmentée et regroupée. Les résultats expérimentaux obtenus sur les circuits Xilinx Zynq-7000 montrent des gains substantiels en termes de LUT pour des matrices allant de 5×5 à 100×100, avec des largeurs de mots de 6 à 16 bits.

Comparée aux approches exactes issues de la littérature, l'approche proposée permet d'optimiser des matrices constantes de dimensions allant jusqu'à 100×100, soit un ordre de grandeur supérieur aux matrices traitées par les méthodes exactes, généralement limitées à de petites tailles en raison de leur complexité exponentielle.

Comparée aux architectures de référence reposant sur des multiplieurs, la version combinatoire permet une réduction allant jusqu'à 5,7× de l'utilisation des LUT et une diminution moyenne de la latence de 2,8×. La version segmentée atteint des fréquences d'horloge allant jusqu'à 301 MHz tout en divisant par deux le nombre de niveaux logiques, tandis que l'architecture segmentée et regroupée réduit la latence totale de 26,7 % à 50 %, au prix d'une baisse modérée entre 17 % et 30 % de la fréquence maximale.

Une étude de cas sur une couche de réseau de neurones quantifié vient valider l'approche dans un contexte d'application réel. Dans ce cadre, l'intégration du facteur d'échelle directe-

ment dans les poids matriciels transfert une partie du coût computationnel vers une phase de prétraitement effectuée en amont avant l'addition des biais, ce qui entraîne aussi un élargissement des opérandes. Néanmoins, une approximation du facteur d'échelle par une valeur à virgule fixe permet de réduire cette surcharge. Lorsque nous appliquons notre algorithme a la matrice constante intégrant le facteur d'échelle, les gains sont comparables à ceux observés sur les matrices synthétiques de dimensions comprises entre 5×5 et 100×100.

Dans l'ensemble, la méthode proposée rend possible une implémentation efficace de multiplications constantes sur circuit logique programmable, adaptée aux systèmes à faible consommation et à haut débit.

ABSTRACT

Constant matrix–vector multiplication is a core operation in many embedded and real-time applications that rely on hardware acceleration, particularly in digital signal processing, embedded artificial intelligence, and control systems.

On FPGA, traditional methods depend on general-purpose multipliers, which are costly in logic resources and energy-inefficient. This overhead becomes especially problematic in edge computing scenarios, where constraints on area, power, and latency are tightly bounded.

This work introduces a scalable, low-complexity algorithm designed to avoid hardware multipliers by leveraging binary arithmetic based on shifts and additions, along with the reuse of intermediate results. The method relies on a recursive factorization of common subexpressions across matrix rows, effectively reducing the total number of additions and generating a computation graph with minimal depth.

Based on this algorithm, three FPGA-targeted hardware implementations are proposed: a purely combinational architecture, a pipelined architecture, and a pipeline-aggregated variant. These designs were evaluated on Xilinx Zynq-7000 devices, showing significant savings in logic utilization for matrices ranging from 5×5 to 100×100 and operand widths between 6 and 16 bits.

Compared to exact methods from the literature, which are typically limited to small matrices due to exponential complexity, the proposed approach successfully optimizes constant matrices up to 100×100, extending scalability by an order of magnitude. When compared against multiplier-based reference architectures, the combinational design achieves up to 5.7× reduction in LUT usage and an average latency decrease of 2.8×. The pipelined version reaches clock frequencies up to 301 MHz while halving logic depth, and the pipeline-aggregated architecture further reduces total latency by 26.7% to 50%, with a moderate frequency drop of 17% to 30%.

To validate the approach in a real-world scenario, a case study was conducted on a quantized neural network layer. In this context, integrating the scaling factor directly into the matrix weights shifts part of the computational cost to a preprocessing stage performed prior to the addition of biases, which increases operand widths. Nevertheless, approximating the scaling factor with a fixed-point value reduces this overhead. When we apply our algorithm to the constant matrix incorporating the scaling factor, we observe gains comparable to those obtained on synthetic matrices ranging from 5×5 to 100×100.

Overall, the empirical results suggest that the proposed method can enable efficient implementation of constant multiplications on FPGA devices, making it suitable for high-throughput and low-power embedded systems.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

ALAP	As Late As Possible (Aussi tard que possible)
BFS	Breadth-first search (Parcours en largeur)
BRAM	Block Random Access Memory (Bloc Mémoire)
CMM	Constant-Matrix Multiplication (Multiplication par matrice constante)
CSD	Canonic-Signed Digit (Chiffres canoniques signés)
CSE	Common Sub Expressions (Sous-expressions communes)
DAG	Directed Acyclic Graph (Graphe orienté acyclique)
DNN	Deep Neural Network (Réseau de neurones profond)
DFS	Depth-first search (Parcours en profondeur)
FF	Flip-Flop (Bascule)
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit (Processeurs graphiques)
HND	Hidden Nonzero Digits (bits non nuls cachés)
ILP	Integer Linear Programming (Programmation linéaire en nombres entiers)
IoT	Internet Of Things (Internet des Objets)
LSB	Least Significant Bit(Bit de poids faible)
LUT	Look-Up Table (Table de correspondance)
MCM	Multiple-Constants Multiplication (Multiplication par plusieurs constantes)
MSB	Most Signifcant Bit (Bit de poids fort)
RPAG	Reduced Pipelined Adder Graph (Graphe additionneurs pipeliné réduit)
RTL	Register-Transfer Level (niveau de transfert de registre)
SCM	Single Constant Multiplication (Multiplication par une constante)
SOP	Sum Of Products (Somme de produits)

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Dans un monde numérique en constante évolution, l'opération fondamentale de multiplication se trouve au cœur des technologies qui façonnent notre quotidien. Des applications grand public aux infrastructures critiques, cette opération mathématique élémentaire sous-tend le fonctionnement de systèmes toujours plus complexes. L'omniprésence des calculs intensifs dans notre société soulève cependant d'importantes préoccupations environnementales et énergétiques, alors que les centres de données et les infrastructures numériques consomment une part croissante des ressources mondiales.

1.1 Contexte

L'essor fulgurant de l'intelligence artificielle ces dernières années a mis en lumière l'importance critique des opérations mathématiques fondamentales, particulièrement les multiplications matricielles, qui reposent à leur tour sur de simples multiplications et additions. Ces opérations constituent la pierre angulaire de nombreuses applications contemporaines, des réseaux de neurones profonds qui consomment d'énormes ressources en effectuant des multiplications tenseur par tenseur, à la cryptographie (notamment les cryptomonnaies), en passant par le traitement de signal et d'image.

1.1.1 Applications émergentes énergivores

L'Internet des Objets (IoT) représente l'un des domaines où cette optimisation est cruciale. Les appareils IoT, caractérisés par leurs ressources limitées en énergie et en capacité de calcul, nécessitent des multiplications ultra-efficaces pour le traitement de données en périphérie (*edge computing*) [3]. Les capteurs intelligents déployés pour la surveillance environnementale doivent effectuer localement des opérations de filtrage et d'analyse qui reposent fondamentalement sur des multiplications [4]. De même, les dispositifs médicaux portables analysant en temps réel les signaux physiologiques et les systèmes de maison intelligente intégrant des algorithmes de reconnaissance vocale ou gestuelle bénéficieraient tous d'une consommation énergétique réduite pour ces opérations fondamentales [5].

Les véhicules autonomes constituent un autre domaine d'application critique où l'efficacité des multiplications est déterminante. Ces systèmes sophistiqués requièrent des calculs en temps réel pour la perception environnementale (traitement d'images et de signaux LIDAR), la cartographie dynamique et la prise de décision intelligente. Ces opérations s'appuient

principalement sur des multiplications matricielles, souvent avec des contraintes énergétiques strictes pour prolonger l'autonomie des batteries [6, 7]. Une amélioration de l'efficacité de ces opérations se traduirait directement par une extension de l'autonomie et une réduction des coûts opérationnels des flottes de véhicules autonomes [8].

La réalité augmentée et virtuelle sur appareils mobiles représente également un cas d'usage exigeant, nécessitant d'intenses calculs de traitement d'image, de reconnaissance d'objets et de rendu 3D avec une puissance limitée [9]. Les drones et la robotique mobile, dont l'adoption s'accélère dans des secteurs comme l'agriculture, la logistique et la sécurité, doivent maximiser l'efficacité des calculs pour la navigation et le contrôle tout en préservant la durée de vie de leur batterie [10, 11]. Dans tous ces contextes, des algorithmes de multiplication optimisés peuvent significativement améliorer les performances tout en réduisant la consommation énergétique, favorisant ainsi l'adoption plus large de ces technologies [12].

La multiplication par constantes, en particulier, représente une opération récurrente dans diverses applications telles que les filtres numériques et les couches de réseaux neuronaux, dont les circuits électriques sont aujourd'hui parmi les plus grands consommateurs d'opérations de multiplication.

Selon un rapport récent du Laboratoire National Lawrence Berkeley [13], les centres de données ont consommé environ 4,4% de l'électricité totale aux États-Unis en 2023, avec une projection atteignant entre 6,7% et 12% d'ici 2028. Plus précisément, leur consommation électrique est passée de 58 TWh en 2014 à 176 TWh en 2023, avec une estimation d'augmentation entre 325 et 580 TWh d'ici 2028 [13]. À l'échelle mondiale, les centres de données représentent actuellement 1,5% de la consommation électrique totale, un chiffre qui pourrait atteindre 3% d'ici la fin de la décennie [14], ce qui équivaut à la consommation actuelle du Japon. Cette croissance exponentielle s'accompagne d'une augmentation préoccupante des émissions de dioxyde de carbone, qui pourraient plus que doubler entre 2022 et 2030 [14].

Au-delà des préoccupations environnementales, l'impact économique de l'optimisation des multiplications est considérable. L'infrastructure numérique mondiale représente un investissement colossal, avec un marché des centres de données évalué à plus de 347.6 milliards de dollars en 2024 [15]. La consommation énergétique constitue jusqu'à 70% des coûts opérationnels de ces infrastructures [16], créant une pression financière significative sur les entreprises technologiques.

L'optimisation des multiplications présente aussi des enjeux particuliers dans le domaine des filtres numériques, dont la complexité est principalement déterminée par le nombre d'opérations de multiplication. La recherche s'est donc concentrée sur la minimisation de la complexité des blocs multiplieurs qui calculent les multiplications par coefficients constants re-

quises dans ces filtres. Bien que la complexité de ces blocs soit considérablement réduite grâce à des techniques efficaces, telles que la décomposition des multiplications en opérations d'additions et de décalage et le partage des sous-expressions communes, des améliorations substantielles restent possibles.

Face à ces défis environnementaux et économiques majeurs, l'optimisation des opérations mathématiques fondamentales, comme la multiplication, revêt une importance capitale. L'évolution des algorithmes de multiplication a marqué des jalons significatifs dans l'histoire de l'informatique.

1.1.2 Algorithmes de multiplication

L'algorithme de Karatsuba en 1962 fut le premier à dépasser l'efficacité asymptotique de la méthode quadratique enseignée a l'école en décomposant les grands nombres en plus petits nombres pour réduire le nombre de multiplications. De l'algorithme de Schönhage–Strassen en 1971 [17], a l'amélioration de Coppersmith-Winograd en 1990 [18], la quête d'efficacité dans les multiplications continue de stimuler l'innovation. L'algorithme de Karatsuba réduit la multiplication de deux nombres de n chiffres à trois multiplications de nombres de n/2 chiffres, illustrant l'importance fondamentale de la réutilisation et de la décomposition pour l'optimisation des calculs.

Depuis les années 1990, l'optimisation des multiplieurs à coefficients constants a suscité un intérêt croissant, notamment dans les domaines du traitement du signal numérique (DSP) et des systèmes embarqués. L'objectif principal était de réduire la complexité matérielle et la consommation énergétique en évitant les multiplieurs coûteux.

En 2006, Kinane, Muresan et O'Connor ont proposé une méthode d'optimisation matérielle pour la multiplication par matrices constantes en utilisant des algorithmes génétiques. Cette approche visait à concevoir des implémentations sans-multiplieurs (*multiplierless*) efficaces en termes de surface et de consommation d'énergie, en explorant un vaste espace de solutions même pour des problèmes de petite taille [19].

En 2007, Hosangadi et al. ont introduit des méthodes algébriques pour optimiser les multiplications constantes dans les systèmes linéaires. Leur approche consistait à éliminer les sous-expressions communes en utilisant des techniques de synthèse logique multi-niveaux, telles que le "*rectangle covering*" et le "*fast extract*" (FX), adaptées à l'optimisation des expressions arithmétiques linéaires [20].

Les travaux de Hosangadi et al. ont été suivis par plusieurs contributions significatives dans le domaine des multiplications par constantes. En 2007, Voronenko et Püschel ont proposé

un algorithme d'optimisation complet pour les multiplications par constantes multiples, améliorant considérablement les méthodes précédentes en réduisant le coût des implémentations matérielles pour des ensembles de constantes arbitraires [21].

1.1.3 Architecture des FPGA modernes

L'efficacité énergétique constitue un paramètre critique dans la conception des systèmes de calcul modernes. Une analyse comparative des architectures matérielles disponibles permet d'identifier les solutions optimales pour l'implémentation d'algorithmes de multiplication matricielle, particulièrement dans le contexte des multiplications par constantes.

Les plateformes matérielles diffèrent grandement en termes de parallélisme, de flexibilité et d'efficacité énergétique [22]. Les processeurs graphiques (GPU) offrent un parallélisme massif avec des milliers de cœurs flottants optimisés pour les opérations de matrice génériques, mais ils s'appuient sur des circuits de multiplication coûteux en énergie pour chaque opération (notamment en FP32) [23]. Ils ne sont pas spécialement conçus pour exploiter des cas où les coefficients sont fixes, car chaque multiplication est traitée comme une opération générale. L'architecture des GPU est optimisée pour l'exécution simultanée d'opérations identiques sur des ensembles de données différents. Cette configuration, bien qu'adaptée aux calculs tensoriels génériques, présente des limitations inhérentes pour les multiplications par constantes. Le modèle d'exécution SIMD (*Single Instruction Multiple Data*) des GPU contraint l'utilisation d'unités de calcul identiques pour toutes les opérations, empêchant ainsi les optimisations spécifiques nécessaires pour exploiter les propriétés des coefficients constants. En revanche, les CPU et les ASIC embarqués disposent de registres, d'unités d'addition et de décalage rapides (*barrel shifters*). Un coefficient constant peut alors souvent être implémenté par une suite d'additions et de décalages dans le code assembleur, sans faire appel à une multiplication matérielle dédiée. Cette approche « logicielle » profite du jeu d'instructions et des optimisations du compilateur pour coder par exemple « multiplier par 5 » comme « décaler à gauche de 2 bits puis ajouter » plus efficacement. Ainsi, sur CPU/ASIC, la liste d'additions partagées peut être directement traduite en instructions élémentaires. Par contraste, les GPU utilisent généralement leur pipeline fixe de multiplication flottante, ce qui les rend moins adaptés.

Les FPGA (*Field-Programmable Gate Array*, ou circuits logiques programmables) constituent un compromis particulier : ils combinent une reprogrammabilité fine du matériel avec une exécution directe en dur (câblée) des calculs [22]. En pratique, on peut y « câbler » les additions intermédiaires partagées pour exploiter pleinement le concept sans multiplieurs. Les FPGA permettent de réaliser des traitements personnalisés (pipeline, mémoire locale)

pour chaque application, obtenant ainsi souvent une latence inférieure et une bien meilleure efficacité énergétique qu'un CPU ou un GPU généraliste pour des tâches spécifiques [22]. Leur nature reconfigurable les rend particulièrement adaptés aux algorithmes sur-mesure comme une multiplication matricielle à coefficients constants optimisée. De plus, les FPGA intègrent de larges interfaces d'entrée/sortie, facilitant leur usage dans les systèmes embarqués et l'IoT [22].

1.1.4 Avantages énergétiques d'une approche sans multiplieurs

L'intérêt majeur de remplacer les multiplications par des additions réside dans l'économie d'énergie. Horowitz [1] fournit des ordres de grandeur : en technologie 45 nm, une multiplication FP32 consomme environ 3,7 pJ contre seulement 0,1 pJ pour une addition de deux entiers 32 bits. Autrement dit, une multiplication FP32 coûte environ 37 fois plus d'énergie qu'une addition entière 32 bits. Le tableau ci-dessous résume ces coûts pour différentes précisions :

Opération	**Entier 8 bits**	**Entier 32 bits**	**FP 16 bits**	**FP 32 bits**
Addition	0,03 pJ	0,1 pJ	0,4 pJ	0,9 pJ
Multiplication	0,2 pJ	3,1 pJ	1,1 pJ	3,7 pJ

Tableau 1.1 Coût énergétique d'opérations arithmétiques (d'après [1]).

Ces écarts renforcent le principe de conception sans multiplieur : chaque opération remplacée par une addition économe réduit significativement la consommation globale. Sur un CPU ou ASIC embarqué, la suite d'additions partagées peut être implémentée sous forme d'instructions (par exemple en assembleur), ce qui évite l'appel à l'unité multiplicative. Toutefois, sur FPGA l'avantage est encore plus marqué car on peut « câbler » ces additions directement dans la logique reprogrammable. Cela évite d'activer les DSP (plus énergivores) et permet de répartir les calculs simples sur de nombreuses LUT/LE à faible coût par opération. En pratique, concevoir l'algorithme de multiplication matricielle à coefficient constant sans multiplieurs dédiés offre une meilleure efficacité énergétique et une grande flexibilité de mise en œuvre (les coefficients fixes sont incarnés dans la structure du circuit).

Dans les domaines d'application avec des contraintes spécifiques (IoT, systèmes embarqués, traitement en bord de réseau), la combinaison de ressources limitées (énergie, surface silicium) et de forts besoins de calcul rend ce genre d'optimisation particulièrement pertinente. Les FPGA sont justement privilégiés dans ces contextes pour leur efficacité énergétique et leur adaptabilité. Leur latence réduite et leur consommation inférieure aux CPU/GPU les

rendent adaptés aux capteurs, objets connectés ou applications en périphérie où chaque microjoule compte. Par exemple, un FPGA peut être directement relié à des capteurs externes grâce à ses interfaces variées, et exécuter localement la multiplication matricielle optimisée en économisant jusqu'à 95% d'énergie par rapport à une solution classique utilisant des multiplications flottantes. De fait, dans ces écosystèmes, la valeur de développer un algorithme *multiplierless* se traduit par des gains importants en autonomie et en performance globale.

1.2 Description du problème

Dans le contexte des réseaux de neurones profonds, les opérations de multiplication sont particulièrement intensives, notamment lors des calculs matriciels qui constituent l'essentiel du traitement. Ces opérations représentent non seulement un défi en termes de performance mais également une source majeure de consommation énergétique. Les architectures matérielles actuelles, bien qu'optimisées, atteignent leurs limites face à l'accroissement de la taille et de la complexité des modèles.

Un défi supplémentaire provient de la nécessité d'équilibrer précision et efficacité. Les représentations à haute précision (virgule flottante 32 bits) offrent une excellente fidélité de calcul mais consomment jusqu'à 18,5 fois plus d'énergie et 27,3 fois plus de surface que les implémentations à virgule fixe 8 bits [24]. Cependant, les représentations avec peu de bits requièrent des stratégies d'approximation judicieuses pour maintenir la précision des résultats. L'approximation des entrées en valeurs de puissances de 2, permettant de remplacer les multiplications par de simples décalages, représente une piste prometteuse mais qui doit être explorée avec rigueur.

Le problème principal réside dans l'implémentation efficace des multiplications par constantes multiples (*Multiple Constant Matrix Multiplications*), où un même vecteur d'entrée doit être multiplié par plusieurs constantes différentes. Les approches conventionnelles, utilisant des multiplieurs dédiés, s'avèrent souvent coûteuses en ressources matérielles et en énergie. La littérature sur ce sujet a considérablement évolué entre 1991 et 2007, avec des avancées significatives dans l'optimisation de ces opérations, mais des limitations persistent, particulièrement pour les matrices de grande taille.

1.3 Objectifs de recherche

Ce mémoire vise à développer et évaluer un nouvel algorithme pour les multiplications matricielles par constantes qui réduit significativement l'utilisation des ressources matérielles (tables de correspondance et bascules) et la consommation énergétique. Les objectifs spéci-

fiques de cette recherche sont :

1. Caractériser un algorithme permettant d'effectuer des opérations de multiplication sans recourir à des multiplieurs dédiés et augmenter la réutilisation des résultats intermédiaires, des sous-expressions communes et des termes de multiplication.
2. Proposer et comparer différentes implémentations matérielles de l'algorithme et en faire l'étude.
3. Évaluer l'évolutivité de l'algorithme proposé en fonction de la taille des matrices, a l'aide d'implémentations matérielle sur FPGA, permettant ainsi de démontrer concrètement les gains en termes de ressources utilisées et de consommation énergétique dans un contexte d'application réel.
4. Valider l'approche dans un contexte réel en implémentant une couche provenant d'un modèle de réseau neuronal quantifié issu de la littérature.
5. Explorer l'impact de l'approximation a virgule fixe du facteur d'échelle dans le but de remplacer la multiplication a virgule flottante par de l'arithmétique entière.

1.4 Contributions

Les principales contributions de ce travail de recherche sont :

1. Publication d'une communication scientifique à la conférence *IEEE Midwest Symposium on Circuits and Systems (MWSCAS) 2024* [25]. Il est à noter que l'implémentation Python de l'algorithme 2 a été réalisée par Dinesh Daultani.
2. Générer et caractériser trois implémentations FPGA distinctes de l'algorithme proposé : une version combinatoire et deux versions pipelinées, chacune présentant des compromis spécifiques en termes de latence, débit et utilisation des ressources.
3. Une méthodologie d'approximation pour les réseaux de neurones basée sur une représentation à virgule fixe, optimisant davantage l'efficacité des calculs tout en maintenant une précision acceptable.
4. Une analyse comparative détaillée des performances de l'algorithme proposé par rapport aux méthodes existantes, en termes d'utilisation des ressources matérielles, de consommation énergétique et de scalabilité.
5. Une étude de cas appliquée démontrant l'efficacité de l'approche dans le contexte d'un réseau neuronal réel.

1.5 Plan du mémoire

La structure de ce mémoire est organisée en six chapitres.

1. Le chapitre 2 est consacré à une revue de la littérature, dans laquelle sont explorées les approches existantes pour l'optimisation de la multiplication par plusieurs constantes, les différentes catégories d'algorithmes, ainsi que les architectures matérielles adaptées.
2. Le chapitre 3 détaille l'algorithme multiplieur proposé, en commençant par l'optimisation des opérations binaires, suivi d'une description complète de l'algorithme, d'un exemple illustratif, puis de ses implémentations en Python et SystemVerilog, incluant les versions combinatoires et pipelinées.
3. Le chapitre 4 présente les expériences réalisées sur du matériel reconfigurable et les résultats obtenus. D'abord une évaluation sur des cas génériques avec une analyse de performances sur FPGA, puis une étude de cas appliquée à un réseau de neurones, incluant une extension de cette méthode avec approximation des poids.
4. Ensuite, une brève discussion est donnée dans le chapitre 5.
5. La fin du mémoire synthétise les contributions et met en évidence les limites de l'approche en ouvrant sur des perspectives d'amélioration.

CHAPITRE 2 REVUE DE LITTÉRATURE

2.1 Optimisation de la multiplication par plusieurs constantes

Le problème de la multiplication par plusieurs constantes (*Multiple Constants Multiplication*, MCM) a largement été exploré dans le domaine des systèmes linéaires tels que les filtres numériques, depuis au moins trois décennies [26].

La résolution de ce problème est particulièrement pertinente pour l'implémentation matérielle des filtres numériques à réponse impulsionnelle finie (Fig. 2.1) [26] [27], mais aussi pour les transformations linéaires de signaux telles que la transformée de Fourier discrète (DFT) ou la transformée en cosinus discrète (DCT), qui reposent sur des produits matrice-vecteur avec une matrice fixe [21].

En pratique, la réponse impulsionnelle d'un filtre FIR est un tableau de constantes, qui est convolué avec le signal d'entrée pour produire un signal transformé en sortie (Fig. 2.1). Le résultat implique alors une somme de produits (*sum of products*, SOP), telle que l'équation 2.1.

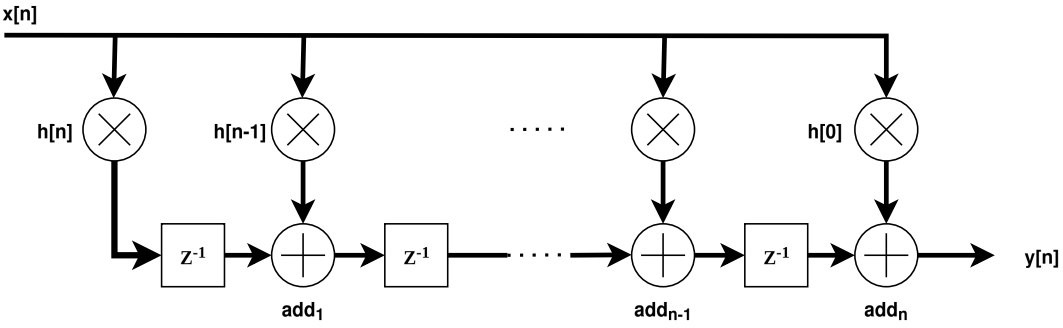


FIGURE 2.1 Filtre FIR, forme transposée

$$y[n] = b_0x[n] + b_1x[n-1] + \cdots + b_Nx[n-N] = \sum_{i=0}^{N} b_ix[n-i] \tag{2.1}$$

Les blocs multiplieurs capables de réaliser cette opération sont des circuits complexes qui nécessitent de grandes surfaces de silicium dans les puces, consomment beaucoup d'énergie et entraînent de longs délais. En revanche, les multiplications par des constantes peuvent être réalisées avec un ensemble réduit d'opérations arithmétiques : additions, soustractions et décalages binaires.

Cette implémentation à faible complexité a deux avantages principaux [28] : (i) Elle est moins coûteuse en termes de ressources matérielles qu'un multiplieur général, tel que le DSP48 qu'on retrouve couramment dans les circuits reconfigurables de Xilinx. (ii) Les constantes à multiplier dans l'opération MCM sont déterminées à l'avance par les algorithmes DSP. Par conséquent, la flexibilité d'un multiplieur général complet n'est pas requise pour la mise en œuvre de cette opération. Ces opérations d'addition, soustraction et décalage sont organisées dans un graphe orienté acyclique où chaque nœud correspond à un additionneur/soustracteur, et le poids de l'arête représente un décalage de bits. Trouver le nombre minimal de nœuds dans ce graphe donne le nombre minimal d'additionneurs pour réaliser cette multiplication. C'est ce que l'on appelle le problème MCM. Dans le problème d'optimisation MCM, on suppose que les coefficients sont connus et déjà quantifiés dans une représentation en virgule fixe (ou entière).

Dans l'exemple de Fig 2.2, la multiplication $45x$ peut être calculé à l'aide de 3 additionneurs. Cependant, la solution optimale ne nécessite que 2 additionneurs.

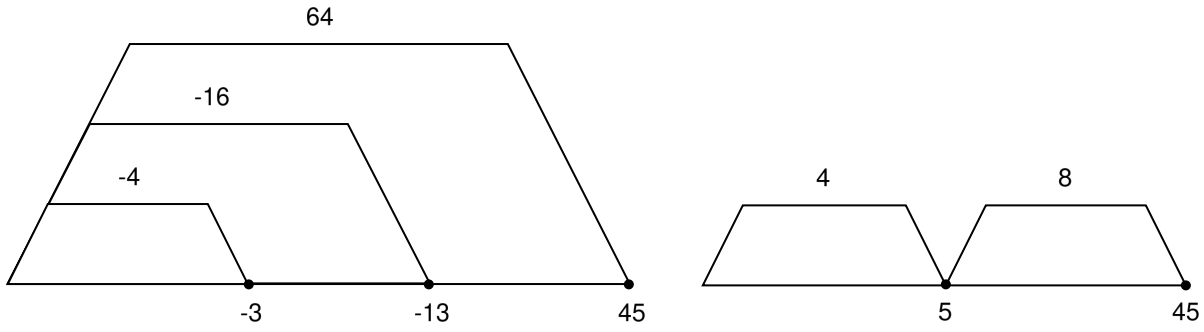


FIGURE 2.2 A gauche, $45x$ calculée en tant que $((x-4x)-16x)+64x$ requiert 3 additionneurs. A droite, calcul optimal : $45x$ calculée en tant que $(8(4x + x) + (4x + x))$, qui est une représentation optimale, ne requiert que 2 additionneurs.

Soit $t \in \mathbb{N}$ une constante codée sur b bits et x l'entrée. La décomposition classique de la multiplication $x \cdot t$ en opérations d'additions et de décalages consiste à traduire chaque bit à `1` de t en un décalage de x et à additionner les termes ainsi décalés. Si k désigne le poids de Hamming de t (nombre de bits à `1`), le coût en additions est proportionnel à k (plus précisément $k-1$ si l'on chaîne les additions).

Par une autre approche, il est possible d'exploiter le complément par rapport à la constante « tout à `1` », avec $c_b = 2^b - 1$.

On écrit alors (éq. 2.2) :

$$x \cdot t = x \cdot c_b - x \cdot (c_b - t) = \underbrace{(x \ll b) - x}_{x \cdot c_b} \; - \; \sum_{i \in \mathcal{Z}} (x \ll i), \tag{2.2}$$

où $\mathcal{Z}$ indexe les bits à `0` de t et $\ll$ note un décalage gauche. Cette forme remplace des additions par des soustractions et fait intervenir le nombre de zéros de t (soit $b - k$). En choisissant, pour chaque t, la meilleure des deux décompositions (par `1` ou par `0`), le coût devient

$$\#\text{adds/subs} \; = \; \min\{k,\; b-k\} \; + \; O(1),$$

d'où un cas moyen (pour un t uniforme sur b) de $b/2 + O(1)$ [21].

Plusieurs travaux ont abordé la meilleure façon de combiner ces opérateurs (addition, soustraction et décalage) pour minimiser le coût et la latence, en s'appuyant généralement sur des algorithmes réutilisant de sous-expressions communes, des algorithmes de recodage, des algorithmes basées sur les graphes, et des algorithmes hybrides [21].

2.1.1 Algorithmes de recodage

Dans cette catégorie, on retrouve notamment les algorithmes tels que la décomposition binaire et la représentation en chiffres canoniques signés (Canonic Signed Digit, CSD), qui utilisent un système de numération ternaire pour réduire le nombre de bits non nuls.

Algorithm 1 Conversion d'un nombre binaire en représentation CSD

Parcourir le nombre binaire de droite à gauche (MSB ← LSB).
Rechercher une séquence de bits égale à `0111`.
Remplacer chaque occurrence de `0111` par $100\bar{1}$.
Répéter l'opération jusqu'à la du nombre.

Le principe de la conversion en format CSD consiste à gérer une liste de constantes à optimiser et à trouver un « motif » qui apparaît plusieurs fois dans l'ensemble des constantes. Un motif est une séquence de chiffres dans l'ensemble $\{-1, 0, 1\}$. Le nombre de bits non nuls dans le motif est appelé son poids. Le format CSD conduit à une implémentation plus efficace de la multiplication en réduisant le nombre de bits non nuls.

Exemple : Soit $93 = (01011101)_2$.
On identifie la séquence 0111 qui correspond à une suite de bits à 1 suivie d'un 0.
Cette séquence est remplacée par sa représentation CSD : $100\bar{1}$.

$$\begin{aligned} 01011101 &\rightarrow 01100\bar{1}01 \quad \text{(remplacement de la première séquence 0111)} \\ &\rightarrow 10\bar{1}00\bar{1}01 \quad \text{(nouveau remplacement si applicable)} \end{aligned}$$

Ce qui correspond à :

$$2^7 - 2^5 - 2^2 + 2^0 = 128 - 32 - 4 + 1 = 93.$$

Comparé à la décomposition classique vue précédemment, la représentation CSD nécessite moins d'additionneurs, car le cas moyen s'améliore à $b/3+O(1)$ (le coût matériel est similaire pour les additionneurs et les soustracteurs, tous les deux étant appelés additionneurs). En moyenne, ce format utilise 33 % de bits non nuls en moins que la représentation binaire.

Bien que la méthode CSD donne généralement de meilleures performances en termes d'efficacité opérationnelle, car elle permet l'utilisation de chiffres négatifs, elle ne garantit pas que la solution trouvée soit la plus optimale.

La performance des algorithmes de recodage dépend fortement de la représentation numérique utilisée, ce qui conduit à des résultats sous-optimaux dans certains scénarios. [21].

2.1.2 Algorithmes basés sur le partage des sous-expressions communes

L'algorithme CSD peut être utilisé pour identifier les opportunités de partage de calculs intermédiaires. La méthode qui consiste à rechercher et éliminer les bits communs à plusieurs constantes, est connue sous le nom de l'élimination des sous-expressions communes (CSE). L'idée des travaux [29] et [30] est d'implémenter un ensemble de multiplications constantes sous forme d'un ensemble d'opérations d'additions et de décalages et d'optimiser celles-ci en fonction des sous-expressions communes.

Ainsi des économies sont obtenues en distribuant les résultats intermédiaires à tous les coefficients qui en dépendent, et en maximisant leurs réutilisation. [29], [27]. Cependant, l'élimination de sous-expressions communes ne peut pas fournir toutes les opportunités de partage possibles en raison de sa dépendance à la représentation des nombres [21] et de l'effet des bits **non nuls cachés** [31].

Les bits non nuls cachés (*Hidden Nonzero Digits* - HND) représentent un phénomène où certains motifs de bits sont "cachés" lors de l'addition de deux nombres en représentation signée. Faust et Chang [31] ont identifié quatre types principaux : *(i)* la collision de chiffres, qui survient lorsque deux chiffres de même signe coïncident à la même position lors de l'addition, formant un seul chiffre retenu à la position supérieure, ce qui rend la position originale

"cachée" ; *(ii)* l'élimination de chiffres (chiffres de signes opposés qui s'annulent) ; *(iii)* la réduction de chiffres contigus, où les chiffres contigus forment un motif réduit de chiffre signé ± 1. Dans ce cas, des séquences telles que $1\bar{1}$ ou $\bar{1}1$ peuvent être réduites en 01 ou $0\bar{1}$; *(iv)* la réduction binaire vers CSD (un motif CSD peut être ajouté à un autre motif pour résulter en une réduction binaire vers CSD, comme dans $101 + 010 = 111 = 100\bar{1}$).

Ces transformations apparaissent dès qu'il existe au moins un degré de liberté (DOF) dans l'arbre d'addition (autrement dit, dès qu'il est possible de réarranger les opérations sans changer la valeur finale). Le nombre de possibilités de partage cachées augmente exponentiellement avec le nombre de DOF du coefficient, défini comme :

$$DOF(C) = 2^{\lceil \log_2 S(C) \rceil} - S(C) \tag{2.3}$$

où $S(C)$ est le nombre de chiffres non-nuls du coefficient C.

Les algorithmes CSE, qui se basent uniquement sur la recherche de motifs récurrents dans une représentation donnée, omettent des solutions potentiellement optimales car ces opportunités de partage sont masquées par les transformations des bits non-nuls cachés. Cette limitation fondamentale des approches CSE a motivé le développement d'algorithmes basés sur les graphes de dépendance (GD), qui recodent les constantes ou se dispensent de représentation explicite.

2.1.3 Algorithmes basés sur les graphes

La représentation en graphe peut être utilisée pour réduire davantage la complexité matérielle. Cette représentation sous forme de graphe a été introduite pour la première fois par Bull et Horrocks dans [26]. Dans cette représentation, chaque sommet du graphe représente un additionneur avec deux entrées et chaque arête représente une multiplication par une puissance de 2, qui peut être implémentée sous forme de décalage binaire.

Ce sont des méthodes ascendante (*bottom-up*) qui construisent itérativement le graphe représentant le bloc multiplieur. La construction du graphe est guidée par une heuristique qui détermine le prochain sommet à ajouter au graphe d'additionneurs. Ces algorithmes ne sont pas limités à une représentation particulière des coefficients ni à une topologie de graphe prédéfinie (comme c'est le cas des algorithmes basés sur les chiffres). Ils produisent généralement des solutions avec le plus faible nombre d'opérations. Des exemples d'algorithmes basés sur les graphes incluent ceux de Bull et Horrocks [26], RAG-n [27], et [21].

2.1.4 Autres méthodes

En plus des algorithmes de recodage, des techniques de partage des sous-expressions et des approches basées sur les graphes, d'autres méthodes ont été explorées pour améliorer l'implémentation matérielle des multiplications par constantes.

Les travaux de Kumm et al. [32] ont introduit l'utilisation des additionneurs à trois opérandes (a + b + c, pris en charge nativement via LUT6 par les FPGA modernes) pour réduire le nombre total d'opérations nécessaires dans les multiplications par constantes multiples (MCM). Ce type d'additionneur ne fait pas appel à une arithmétique en base trois ni à des chiffres -1,0,1. Dans cette approche, la topologie d'addition autorise la sommation de trois termes en une seule étape logique, ce qui diminue la profondeur du graphe d'additionneurs et donc la latence. Les résultats montrent une réduction de 27 % du nombre d'opérations enregistrées par rapport aux circuits optimaux utilisant des additionneurs binaires à deux entrées, ce qui entraîne une diminution de 15,7 % des tranches sur les FPGA Xilinx et de 10,5 % des ALM (*Adaptive Logic Module*) sur les FPGA Altera. De plus, le nombre d'étages de pipeline est réduit, diminuant ainsi la latence globale du système. Bien que la fréquence maximale baisse légèrement (26 % sur Xilinx), les circuits restent très rapides, atteignant plus de 370 MHz sur Virtex 6 et 450 MHz sur Stratix IV. Cette approche optimise l'utilisation des ressources logiques tout en conservant des performances élevées, ce qui en fait une solution idéale pour des applications de traitement du signal numérique nécessitant des calculs rapides et efficaces.

Cependant, cette approche présente certaines limites : aucune méthode optimale pour résoudre le problème MCM avec des additionneurs à trois opérandes n'a été trouvée à ce jour. De plus, l'ajout d'une troisième entrée complexifie la logique combinatoire, allongeant ainsi le chemin critique et pouvant réduire la vitesse des circuits. Par exemple, des expérimentations sur FPGA montrent une réduction de la fréquence d'horloge allant jusqu'à 26 % sur les FPGA Xilinx.

En revanche, pour le cas spécifique de la multiplication par une constante unique (*Single Constant Multiplication*, SCM), des méthodes optimales utilisant des additionneurs à trois opérandes existent. Dans leurs travaux ultérieurs, Kumm et al. [33] ont démontré qu'il est possible de minimiser de manière optimale le nombre d'opérations pour le problème SCM grâce à une exploration exhaustive des motifs d'additionneurs. Ces résultats ne s'appliquent cependant qu'au cas SCM et ne permettent pas de généraliser l'approche aux scénarios MCM, car l'espace de recherche est suffisamment petit [33].

Enfin, un autre facteur à considérer est la consommation accrue des ressources de routage :

bien que les additionneurs à trois opérandes utilisent la même quantité de slices ou d'ALM que les additionneurs à deux entrées, ils nécessitent davantage de connexions locales. Dans des conceptions plus grandes, cela peut saturer les ressources de routage disponibles, rendant certaines slices inutilisables malgré leur disponibilité logique.

2.2 Généralisation du problème de la multiplication par plusieurs constantes en matrice constante

L'opération de multiplication par une matrice constante (*Constant Matrix Multiplication*, CMM) d'une matrice $M \times N$ est une multiplication matrice-vecteur, où chaque sortie correspond au produit scalaire entre une ligne de la matrice de coefficients et le vecteur d'entrée, comme décrit dans l'équation 2.4.

$$\begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,N} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ c_{M,1} & c_{M,2} & \cdots & c_{M,N} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{pmatrix} \tag{2.4}$$

Le problème de la multiplication par une matrice constante (CMM) peut être vu comme une généralisation du problème de la multiplication par plusieurs constantes (*Multiple Constant Multiplication*, MCM). Dans le problème MCM, une seule entrée est multipliée par plusieurs constantes connues. Dans le cas de CMM, plusieurs entrées sont multipliées par une matrice constante, et chaque sortie peut impliquer des combinaisons communes de produits partiels provenant de ces multiplications.

Plus précisément, chaque sortie du produit scalaire entre une ligne de la matrice C et le vecteur d'entrée X est calculée par une somme de produits (Sum of Products, SOP), comme illustré dans l'équation 2.5. Ainsi, la multiplication matrice-vecteur implique M produits scalaires, soit M sommes de produits. Chaque produit dans ces SOP correspond à une multiplication d'une entrée par un coefficient constant, ce qui constitue un sous-problème de type MCM.

$$y_1 = c_{1,1}x_1 + c_{1,2}x_2 + \cdots + c_{1,N}x_N \tag{2.5a}$$
$$y_2 = c_{2,1}x_1 + c_{2,2}x_2 + \cdots + c_{2,N}x_N \tag{2.5b}$$
$$\vdots$$
$$y_M = c_{M,1}x_1 + c_{M,2}x_2 + \cdots + c_{M,N}x_N \tag{2.5c}$$

Par conséquent, on peut voir le problème CMM comme étant constitué de M problèmes MCM interdépendants. La multiplication matrice-vecteur devient alors une suite de M sommes de produits, chaque produit pouvant être optimisé via des techniques MCM. Cela permet d'exploiter la redondance entre les produits partiels pour réduire la complexité matérielle globale.

2.3 Avancées dans les architectures et algorithmes de multiplication efficace de matrices

2.3.1 Approches exactes

Depuis que Dempster et MacLeod [27] ont étendu leur algorithme d'optimisation MCM pour résoudre un problème CMM généralisé, plusieurs auteurs ont à leur tour proposé des méthodes d'optimisation pour le problème CMM. Certaines de ces approches reposent sur une recherche exhaustive de l'espace des solutions.

Par exemple, Kumm et al. [34] ont proposé une approche heuristique qui utilise une recherche en profondeur (DFS) inspirée de l'algorithme du graphe d'addition pipeliné réduit (RPAG) [35] pour trouver le graphe de profondeur minimale en additions. Ils utilisent la programmation linéaire en nombres entiers (ILP, *integer linear programming*) pour obtenir des solutions optimales pour les multiplications constantes. Leurs résultats indiquent que l'ILP peut fournir des configurations optimales avec une réduction de l'utilisation des ressources d'environ 50 % par rapport aux méthodes traditionnelles. Les auteurs ont démontré que leur approche basée sur l'ILP pouvait résoudre des instances du problème CMM avec des matrices de taille 8×8 dans un délai raisonnable, montrant le potentiel pour une multiplication matricielle haute performance dans les implémentations matérielles.

Kinane *et al.* [36] [37] ont considéré toutes les permutations des représentations en chiffres signés des constantes pour construire les sous-termes de sommes de produits (SOP). Les résultats sont ensuite sélectionnés et recombinés par un algorithme génétique pour trouver la

meilleure SOP combinée globale (en termes de nombre d'additions). L'espace des solutions est vaste malgré les efforts de réduction de complexité, et peut donc ne pas convenir pour de grandes matrices.

Aksoy *et al.* [38] [39] ont proposé deux algorithmes distincts pour fournir des solutions exactes au problème MCM : un algorithme de recherche en largeur exacte (BFS) et un algorithme de recherche en profondeur exacte (DFS). L'algorithme BFS explore systématiquement toutes les combinaisons possibles d'opérations en construisant un arbre de recherche exhaustif. Cette méthode garantit une solution optimale en explorant de manière exhaustive l'ensemble de l'espace des solutions. Dans leurs expériences, Aksoy *et al.* ont rapporté que l'algorithme BFS a atteint des solutions optimales avec une réduction moyenne de 45 % du nombre d'opérations par rapport aux méthodes heuristiques existantes.

L'algorithme DFS, quant à lui, repose sur les limites établies par les algorithmes approximatifs pour rechercher des solutions minimales. Il élague efficacement l'espace de recherche en évitant les chemins qui ne sont pas susceptibles de mener à une solution minimale. Les auteurs ont noté que l'algorithme DFS améliorait l'efficacité computationnelle par rapport à l'approche BFS, le rendant plus pratique pour des matrices plus grandes. L'algorithme DFS a réussi à trouver des solutions avec une réduction allant jusqu'à 35 % des opérations pour des matrices de taille 8×8, démontrant son efficacité pour équilibrer précision et demande computationnelle.

2.3.2 Approches approximatives

En contraste avec les méthodes exactes, les approches approximatives sacrifient souvent la précision pour une efficacité améliorée et une utilisation réduite des ressources. Aksoy *et al.* [28] ont exploré diverses techniques d'approximation, en se concentrant sur la réduction du nombre d'opérations nécessaires pour le MCM. Leur algorithme approximatif basé sur les graphes cherche à trouver rapidement une solution proche du minimum en sélectionnant stratégiquement des constantes intermédiaires qui peuvent synthétiser le plus grand nombre de constantes cibles à chaque itération. Cet algorithme utilise une approche heuristique pour réduire considérablement l'espace de recherche, accélérant ainsi le processus de calcul. Cependant, il ne garantit pas que la solution minimale sera trouvée, car il peut converger vers un minimum local.

Les auteurs ont rapporté que cet algorithme pouvait atteindre des solutions à 10 % de l'optimal dans 80 % des cas testés, tout en réduisant le temps de calcul jusqu'à 70 %. Ce compromis entre précision et efficacité est particulièrement intéressant dans les applications où la vitesse est critique.

Lehnert *et al.* [40] ont proposé une méthode novatrice pour réduire les besoins en ressources matérielles en décomposant les matrices de poids en sous-matrices creuses contenant des puissances de deux. Leurs résultats expérimentaux ont indiqué que cette approche pouvait diminuer significativement l'utilisation des ressources. Spécifiquement, ils ont atteint une réduction du nombre de multiplications nécessaires dans leur mise en œuvre FPGA par un facteur de 2× à 6×, démontrant le compromis entre efficacité des ressources et précision. Bien que cette méthode traite efficacement certaines inefficacités computationnelles, elle n'assure pas l'évolutivité ni la précision pour les opérations impliquant de grandes matrices denses.

Une autre contribution notable aux méthodes approximatives est le schéma de quantification *Additive Powers-of-Two* (APoT) proposé par [41]. Cette méthode permet une discrétisation non uniforme efficace des poids dans les réseaux neuronaux, réalisant une réduction significative de la complexité computationnelle. Li *et al.* ont rapporté que la quantification APoT peut réduire le nombre de multiplications jusqu'à 50 % tout en maintenant un niveau élevé de précision des résultats. Cette approche est particulièrement bénéfique dans les scénarios où les ressources matérielles sont limitées, car elle permet l'implémentation de modèles d'apprentissage profond sur des dispositifs contraints en ressources.

Bien que ces approches traitent certaines inefficacités computationnelles des CMM, elles n'assurent pas l'évolutivité ni la précision pour les opérations impliquant de grandes matrices denses.

2.4 Multiplieurs efficaces sur systèmes reconfigurables FPGA

Les FPGA actuels emploient des tables de correspondance (*Look-Up Tables, LUT SRAM*), des bascules (Flip-Flop, FF), des blocs DSP et de la mémoire embarquée (*Block RAM, BRAM*), interconnectés par un réseau programmable.

1. **Éléments logiques de base (LE ou BLE)** : ils combinent typiquement une LUT à 6 entrées suivie d'un bascule (flip-flop) [42]. Par exemple, une architecture Xilinx Zynq-7000 regroupe deux tranches (slices) par CLB : chaque slice contient quatre LUT 6-entrées et huit bascules. Ces LUT peuvent être fractionnées : une LUT6 peut agir comme deux LUT5 indépendantes si utile. Elles offrent aussi des modes spéciaux : on peut configurer une LUT comme petite mémoire (RAM distribuée) de 64 bits, ou comme registre à décalage (SRL32). Les FPGA incluent également des chaînes de retenue rapide (carry chains) entre LUT pour accélérer les additions.
2. **Blocs DSP** : ce sont des blocs ASIC spécialisés intégrés qui implémentent des multiplications et additions à faible coût temporel. Par exemple, un bloc DSP peut effec-

tuer en dur deux multiplications 18×18 bits ou une multiplication 27×27 bits, avec des additions avant/après, et gérer des formats fixes et flottants (par ex. 32 bits, 16 bits, float16) [43, 44]. Les DSP sont très efficaces pour les opérations MAC (multiply-accumulate), mais ils sont coûteux en silicium et consomment plus d'énergie qu'un simple additionneur LUT. Dans une approche sans multiplieur, on préfère utiliser la logique programmable (LUT) pour éviter cette dépense énergétique.

3. **Blocs mémoire (BRAM)** : ce sont de grandes mémoires SRAM internes (typiquement 18–36 Kbits), accessibles en simple ou double port. Elles stockent les données et coefficients intermédiaires. Par exemple, chaque bloc BRAM de 20 Kbits (Xilinx) est double port, avec des options d'organisation de largeur différentes (ex. 1024 × 20 bits, 2048 × 10 bits) [43]. On stocke souvent les matrices de grande taille dans ces BRAM.
4. **Réseau d'interconnexion programmable** : il relie les différents blocs (LUT, DSP, BRAM, E/S) via des multiplexeurs programmables contrôlés par SRAM. Ces multiplexeurs 2 :1 permettent de diriger chaque fil de connexion vers divers chemins. Ils représentent en général plus de 50 % de la surface d'un FPGA [22]. Ce réseau flexible autorise la création de liaisons logiques arbitraires, mais il impose de bien dimensionner les ressources pour éviter les goulots d'étranglement. Dans l'approche sans multiplieurs, les interconnexions sont utilisées pour connecter toutes les additions et partages de sommes intermédiaires nécessaires à l'algorithme.

L'implémentation de multiplieurs optimisés sur du matériel reconfigurable est devenue un point central dans la quête d'une multiplication matricielle efficace. Les avancées récentes ont démontré que les FPGA peuvent améliorer significativement les performances des multiplieurs matrice-vecteur grâce à leurs capacités de traitement parallèle et leurs architectures personnalisables. Kumm *et al.* [32] ont mis en évidence le potentiel des additionneurs ternaires dans les applications MCM, démontrant leur capacité à réduire l'utilisation des ressources tout en maintenant un haut débit. Leur implémentation FPGA a atteint une augmentation de débit de 30 % par rapport aux additionneurs binaires traditionnels, ce qui en fait un choix convaincant pour les applications haute performance.

Lehnert *et al.* [40] ont en outre souligné les avantages des implémentations FPGA pour la multiplication matrice-vecteur, notant que leur architecture proposée convient à une large gamme d'applications, au-delà des réseaux neuronaux artificiels. La flexibilité des FPGA permet l'adaptation des circuits de multiplication pour répondre aux demandes spécifiques de diverses tâches computationnelles, améliorant ainsi l'efficacité globale. Leurs résultats expérimentaux ont montré que ce type d'implémentation pouvait atteindre une accélération de 2,5 fois par rapport aux processeurs DSP conventionnels, démontrant l'efficacité de leur

approche dans des scénarios pratiques.

2.5 Conclusion

Les progrès dans les architectures et les algorithmes de multiplication matricielle efficaces révèlent un riche paysage de recherche axé sur l'optimisation des méthodes exactes et approximatives. Les approches exactes fournissent des solutions rigoureuses qui garantissent l'optimalité mais peuvent avoir des difficultés avec l'evolutivité sur les grandes matrices. L'algorithme BFS, par exemple, a atteint des solutions optimales avec une réduction moyenne de 45 % du nombre d'opérations, tandis que l'algorithme DFS a démontré une réduction allant jusqu'à 35 % des opérations pour des matrices de taille 8×8. À l'inverse, les méthodes approximatives offrent des voies prometteuses pour améliorer l'efficacité, bien que cela se fasse au détriment de la précision, comme le démontrent les résultats de [40] et [41]. L'algorithme approximatif basé sur les graphes a rapporté des solutions à 10 % de l'optimal dans 80 % des cas testés, tout en réduisant le temps de calcul jusqu'à 70 %.

L'état de l'art des multiplieurs optimisés sur du matériel reconfigurable, en particulier les FPGA, souligne l'importance de la flexibilité et de l'adaptabilité pour atteindre une multiplication matricielle haute performance. Les mises en œuvre utilisant des additionneurs ternaires ont démontré une augmentation de débit de 30 %, tandis que Lehnert et al. [40] ont atteint une accélération de 2,5 fois par rapport aux processeurs DSP conventionnels. Ces avancées mettent en évidence le potentiel des FPGA pour améliorer les performances des multiplieurs matrice-vecteur grâce à leurs capacités de traitement parallèle.

Alors que les applications continuent d'exiger des matrices plus grandes et plus complexes, l'exploration continue des méthodes exactes et approximatives sera cruciale pour développer des solutions évolutives qui répondent aux besoins des environnements informatiques modernes. À la lumière de ces progrès, nous avons développé un nouvel algorithme spécialement conçu pour les multiplications matricielles à coefficients constants qui vise à améliorer la scalabilité sans sacrifier la précision. Cet algorithme tire parti des connaissances acquises à partir des méthodes exactes et approximatives, et vise a minimiser l'utilisation des ressources FPGA en réduisant le nombre de cellules logiques nécessaires pour effectuer des multiplications par matrice constante (CMM), tout en exploitant les redondances dans les sous-expressions communes à plusieurs produits et sorties.

CHAPITRE 3 ALGORITHME MULTIPLIEUR OPTIMISÉ

3.1 Multiplication par somme de décalages binaires

L'approche proposée repose sur l'idée de la multiplication par sommation de produits partiels. Soit X, une variable de N bits, et C une constante de M bits. X et C peuvent être exprimés sous le format suivant :

$$X = \sum_{i=0}^{N-1} x_i 2^i, \quad C = \sum_{j=0}^{M-1} c_j 2^j \tag{3.1}$$

Le produit $P = X \times C$ peut être calculé en accumulant leurs produits partiels. Cela implique de multiplier les bits de la variable et de la constante et de les décaler ensuite vers la gauche d'un nombre approprié de positions :

$$P = \sum_{j=0}^{M-1} X c_j 2^j \tag{3.2}$$

Ce principe est la base de l'algorithme proposé. La constante C étant connue à l'avance, les c_i nuls peuvent être ignorés, et la multiplication se réduit à quelques opérations de décalage et d'addition. Par exemple, multiplier une variable X par 193 peut être vu comme la somme de trois versions décalées de X :

$$193X = 128X + 64X + X \tag{3.3}$$

Un avantage évident se présente lorsqu'on considère l'implémentation matérielle des multiplications constantes, car les opérations de décalage fixes sont computationnellement gratuites [26]. Tirer profit des décalages de bits sans coût en remplaçant les multiplications des puissances de deux par une simple opération de décalage offre donc des économies de ressources significatives.

3.2 Description détaillée de l'algorithme proposé

L'algorithme proposé est décrit ci-dessous (Algorithme 2). Il s'inscrit dans la famille des méthodes basées sur les graphes d'additionneurs (cf. 2.1.4). À l'instar de RAG-n et dérivés, il recherche et réutilise des sous-expressions pour partager des additions entre sorties. La

spécificité ici tient au traitement matriciel global : on maximise la réutilisation à travers les colonnes (constantes) et les lignes (sorties), alors que les MCM classiques ciblent un seul vecteur d'entrées ou un seul bloc de constantes. Une description détaillée de son fonctionnement est mentionné plus bas (cf. 3.2.1).

Algorithm 2 Multiplieur optimisé par matrice constante

Input: Matrice constante W (taille M x N) avec poids codés sur B bits, vecteur d'entrées d'activation variable $\mathbf{X}$ (taille N)
Output: Liste des opérations sur un tableau de variables MEM, et vecteur résultat Y (taille M)

1: $binW \leftarrow binarizeMatrix(W, B)$ ▷ Binarisation de la matrice
2: Initialiser MEM avec les valeurs décalées de X
3: $maxCount \leftarrow \max_j \left(\sum_{i=0}^{M-1} binW[i,j] \right)$ ▷ Trouver le nombre maximal de 1 par colonne
4: $col \leftarrow B \times N$
5: **while** maxCount >= 1 **do** ▷ Itération par colonne
6: **for** $j = 0$ à col($binW$) **do**
7: Trouver la colonne k telle que
$\sum_{i=0}^{M-1} binW[i,j] \cdot \sum_{i=0}^{M-1} binW[i,k] = \text{maxCount}$ ▷ Trouver les colonnes similaires
8: **if** k trouvé **then**
9: $binW[:, col] = binW[:, j] \ \cdot \ binW[:, k]$
10: Effacer les 1 correspondants dans les colonnes j et k
11: MEM[col++] $\leftarrow$ MEM[j] + MEM[k]
12: **end if**
13: **end for**
14: $maxCount$--;
15: **end while**
16: Étant donné qu'il ne reste qu'un seul 1 à la position p_k pour la ligne k,
$Y_k = \text{MEM}[p_k]$
17: **return** Y

3.2.1 Initialisation des matrices binaires

Tout d'abord, nous construisons une matrice de poids binaire $binW$ (ligne 1) en utilisant la fonction $binarizeMatrix(W, B)$, basée sur les compléments à deux de W. Cette forme binaire est essentielle pour réaliser des opérations bit à bit dans les étapes ultérieures. Un tableau MEM est également initialisé avec les valeurs décalées des variables (ligne 2). Pendant l'exécution de l'algorithme, MEM et $binW$ seront étendus vers la droite afin de stocker les sous-expressions communes et leurs contributions aux sorties. Chaque colonne subséquente ajoutée à ce tableau correspond à un nouveau nœud n_z contenant la somme de deux nœuds précédemment calculés $n_z = n_x + n_y$, où $x, y < z$.

3.2.2 Réduction des colonnes

Pendant la réduction, *maxCount* représente le nombre maximum de correspondances potentielles entre n'importe quelle paire de colonnes. Sa valeur initiale est le nombre maximum de bits "1" dans toutes les colonnes de *binW* (ligne 3). L'algorithme effectue un balayage itératif de *binW* pour identifier les colonnes ayant le nombre maximum commun. Ceci est réalisé en calculant le produit scalaire entre toutes les paires de colonnes (ligne 7). Lorsque le nombre de "1" communs atteint *maxCount*, une nouvelle colonne est ajoutée à *binW* (ligne 11) contenant les 1 communs, tandis que les "1" correspondants dans les colonnes originales sont effacés (ligne 10). Lorsqu'il n'y a plus de paires de colonnes avec des correspondances *maxCount*, le *maxCount* est décrémenté, et le processus se poursuit jusqu'à ce que *binW* devienne une matrice avec un seul "1" par ligne.

3.2.3 Calcul du résultat

Pour chaque ligne de *binW*, désormais réduite à un seul '1' par ligne, la valeur correspondante dans MEM est attribuée à Y (ligne 16). Chaque résultat peut donc être vu comme le sommet d'un graphe de calcul comportant des additions et des décalages partagés (voir figure 3.1).

Les nœuds (sauf les nœuds feuilles) contiennent l'identifiant du nœud (*node ID*), qui correspond aussi à l'indice de la colonne dans laquelle il apparaît dans le tableau 3.1. Ils contiennent également l'opération (seulement +) ainsi que le cycle de départ. Dans ce graphe, "@1" signifie que ce nœud est créé au cycle 1. Les nœuds feuilles contiennent uniquement le *node ID*.

3.3 Exemple illustratif

Pour illustrer les mécanismes par lesquels l'algorithme proposé optimise une multiplication matrice-vecteur, considérons l'exemple suivant où W représente une matrice constante non signée sur 4 bits et X un vecteur de variables. Cet exemple servira de reference pour le reste de cette section.

$$\mathbf{W} = \begin{pmatrix} 5 & 4 & 9 \\ 1 & 3 & 1 \\ 3 & 7 & 11 \end{pmatrix}, \quad \mathbf{X} = \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} \tag{3.4}$$

Le résultat Y du produit matrice-vecteur est exprimé sous la forme :

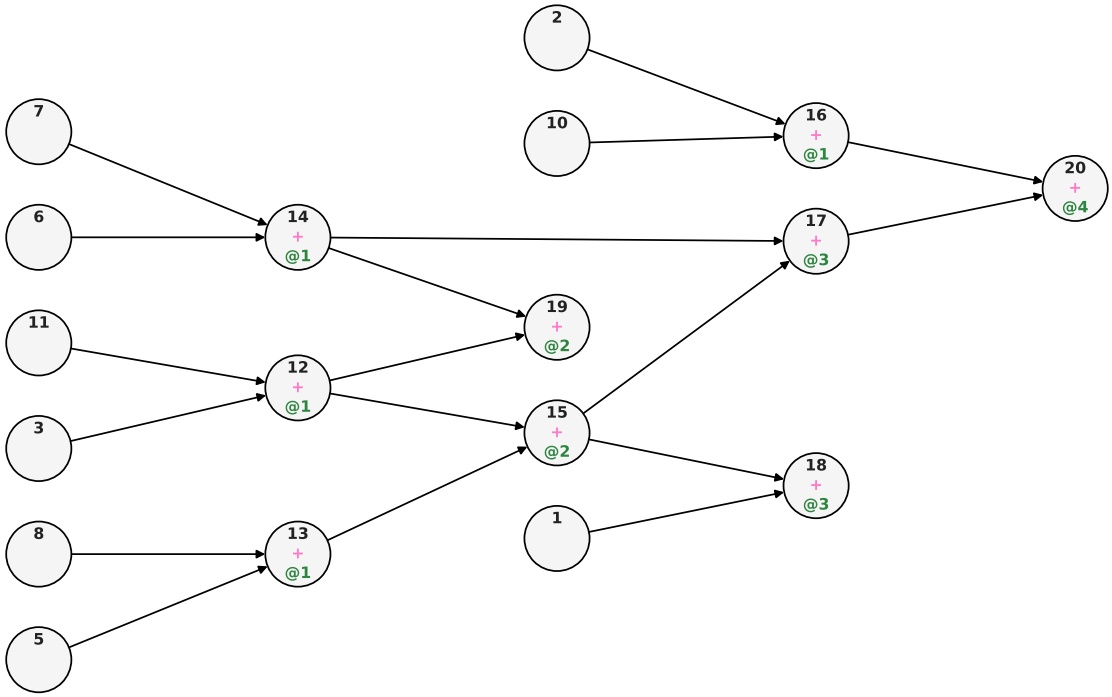


FIGURE 3.1 Graphe des opérations avec addition

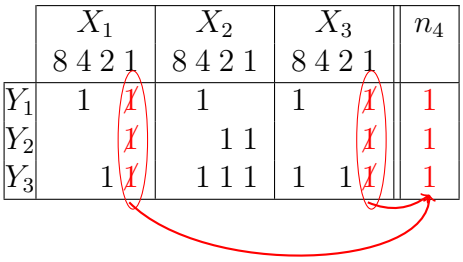
$$\mathbf{Y} = \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \mathbf{WX} = \begin{bmatrix} 5 & 4 & 9 \\ 1 & 3 & 1 \\ 3 & 7 & 11 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} \tag{3.5}$$

L'algorithme proposé repose sur une matrice binaire illustrée dans le tableau 3.1 où les trois premières colonnes sont construites à partir de la forme binaire sur 4 bits de W (les zéros sont omis par souci de clarté). Ainsi, chaque sous-colonne représente la variable associée (X_1, X_2 ou X_3) multipliée par une puissance de 2. Chaque ligne représente un résultat de sortie (Y_1, Y_2 ou Y_3). Les "1" dans les lignes marquent les contributions des variables d'entrée aux résultats de sortie. Par exemple, dans la première ligne : $Y_1 = 4X_1 + X_1 + 4X_2 + 8X_3 + X_3$.

Pour réutiliser les produits partiels communs entre les lignes, l'algorithme proposé recherche les paires de colonnes présentant le nombre maximal de "1" aux mêmes positions. Dans notre exemple, lors de la première itération, ces deux colonnes sont $(1)X_1$ et $(1)X_3$, ce qui signifie que le sous-produit $(1)X_1 + (1)X_3$ est commun aux trois sorties.

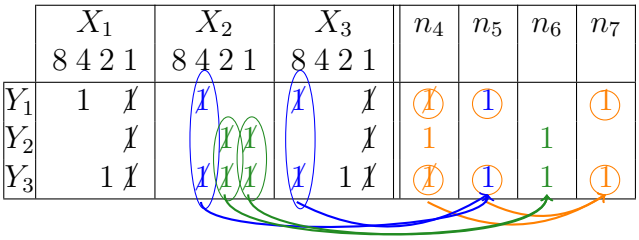
L'algorithme substitue alors l'addition X_1+X_3 par un nouveau nœud (colonne) $n_4 = X_1+X_3$ (Table 3.2), remplaçant ces deux produits partiels dans les lignes du vecteur résultat. La

TABLEAU 3.1 Illustration de l'algorithme proposé. *Étape 1* : Élimination des colonnes contenant le plus grand nombre de '1' (ce qui correspond aux puissances de 2 les plus répandues)



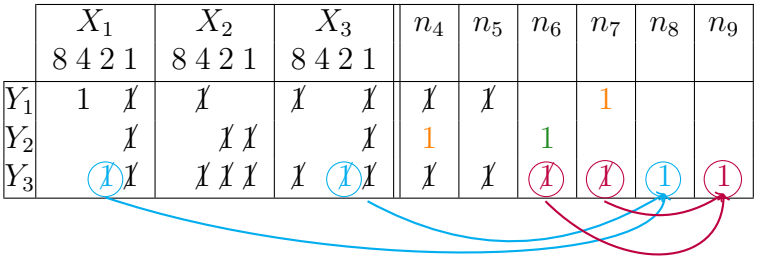
colonne de la nouvelle variable n_4 est ajoutée à droite de la matrice binaire, et les 1 communs dans les colonnes sources $(1)X_1$ et $(1)X_3$ sont déplacés vers la nouvelle colonne. Désormais, l'addition $X_1 + X_3$ ne doit être calculée qu'une seule fois au lieu de trois.

TABLEAU 3.2 *Étape 2* : Passage aux prochaines colonnes contenant le plus de '1'



L'algorithme passe ensuite aux produits partiels les plus communs suivants. Dans le tableau 3.3, il s'agit de $(4)X_2$ et $(8)X_3$ qui apparaissent dans deux lignes. Une nouvelle colonne avec la variable $n_5 = 4X_2 + 8X_3$ est ajoutée à droite de la matrice binaire. Le processus est répété pour n_6, n_7, puis Y_1, Y_2 et Y_3.

TABLEAU 3.3 *Étape 3* : Cette opération continue pour les colonnes suivantes



L'avant dernière étape dans le tableau 3.3 consiste à remplacer, à travers les lignes, les puissances de 2 qui apparaissent plus d'une fois sur pour la meme variable Y_i jusquà'ce qu'aucune autre addition de produits partiels ne peut être remplacée.

TABLEAU 3.4 *Étape 4* : Les dernières colonnes contiennent les résultats finaux Y_1, Y_2 et Y_3. Par souci de clarté, nous montrons uniquement la dernière étape, sans les '1' simplifie

	X_1 8 4 2 1	X_2 8 4 2 1	X_3 8 4 2 1	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}	n_{11}	n_{12}
Y_1	~~1~~						~~1~~			1		
Y_2				~~1~~		~~1~~					1	
Y_3								~~1~~	~~1~~			1

La dernière étape dans le tableau 3.4 consiste à remplacer, à travers les lignes, les puissances de 2 qui apparaissent plus d'une fois pour la même variable Y_i, jusqu'à ce qu'aucune autre addition de produits partiels ne puisse être remplacée. Ainsi, notre tableau réduit ne contient à la fin que des colonnes avec un seul '1'. La liste des substitutions est retenue en mémoire. Les équations en gras montrent les résultats finaux Y_1, Y_2 et Y_3.

$$n_4 = X_1 + X_3 \tag{3.6}$$
$$n_5 = 4X_2 + 8X_3 \tag{3.7}$$
$$n_6 = 2X_2 + X_2 \tag{3.8}$$
$$n_7 = n_4 + n_5 \tag{3.9}$$
$$n_8 = 2X_1 + 2X_3 \tag{3.10}$$
$$n_9 = n_6 + n_7 \tag{3.11}$$
$$n_{10} = 4X_1 + n_7 \tag{3.12}$$
$$n_{11} = n_4 + n_6 \tag{3.13}$$
$$n_{12} = n_8 + n_9 \tag{3.14}$$
$$\mathbf{Y_1 = n_{10}} \tag{3.15}$$
$$\mathbf{Y_2 = n_{11}} \tag{3.16}$$
$$\mathbf{Y_3 = n_{12}} \tag{3.17}$$

3.4 Implémentation Python

Cette implémentation a été réalisée en Python par Dinesh Daultani dans le cadre d'un stage de recherche et constitue sa contribution à l'article [25]. Cette implémentation génère en sortie une liste d'opérations d'addition, qui sera nommée par la suite *OperationsArray*. Elle est nommée ainsi car le premier indice de chaque ligne correspond à l'indice d'un bit dans la

matrice de poids initiale.

OperationsArray est un tableau bidimensionnel qui contient des informations pour chaque ligne de la matrice, ainsi que des opérations à effectuer sur ces éléments. Il encode plus particulièrement la séquence d'opérations nécessaires pour calculer le vecteur de sortie. Chaque ligne du tableau contient :

— La première colonne représente l'indice du bit dans la mémoire (ou la première opérande d'une addition lorsque l'opération est `+`).

— La deuxième colonne contient l'indice de l'entrée (ou de la valeur à opérer), ou la deuxième opérande d'une addition lorsque l'opération est `+`.

— La troisième colonne indique la largeur en bits de l'élément si le parametre est `'«'`, ou l'indice de la nouvelle colonne (*Node ID*) où stocker le résultat de l'addition lorsque l'opération est `+`.

— La quatrième colonne spécifie l'opération à réaliser (par exemple, décalage à gauche `«`, décalage à droite `»`, addition `+`, ou `=` lorsqu'il s'agit de la sortie finale).

```
OperationsArray =
[[3, 3, 4, 'parameters'],
 [0, 0, 3, '<<'],
 [1, 0, 2, '<<'],
 [2, 0, 1, '<<'],
 [3, 0, 0, '<<'],
 [4, 1, 3, '<<'],
 [5, 1, 2, '<<'],
 [6, 1, 1, '<<'],
 [7, 1, 0, '<<'],
 [8, 2, 3, '<<'],
 [9, 2, 2, '<<'],
 [10, 2, 1, '<<'],
 [11, 2, 0, '<<'],
 [3, 11, 12, '+'],
 [5, 8, 13, '+'],
 [6, 7, 14, '+'],
 [12, 13, 15, '+'],
 [2, 10, 16, '+'],
 [14, 15, 17, '+'],
 [1, 15, 18, '+'],
 [12, 14, 19, '+'],
```

```
23  [16, 17, 20, '+'],
24  [0, 18, -1, '='],
25  [1, 19, -1, '='],
26  [2, 20, -1, '=']]
```

Extrait de code 3.1 Tableau d'operations *OperationsArray* basé sur l'exemple précedent sous forme de liste Python

La sortie d'exécution 3.1 encode, dans un format explicite, toutes les opérations nécessaires pour réaliser la multiplication et calculer le résultat. Ce format est décodé selon la colonne "*parameters*" (4^{e} élément de chaque ligne) de chaque élément de *OperationsArray*. Les lignes colorées reflètent les colonnes des tableaux précédents (tableaux 3.1, 3.2 et 3.3).

La phase de test consiste à créer une matrice aléatoire et à effectuer une multiplication matrice-vecteur avec plusieurs vecteurs d'entrée générés de manière aléatoire. Ces vecteurs d'entrée sont multipliés avec la matrice de poids définie par *OperationsArray*. Les matrices de test sont générées en définissant des tailles de matrices spécifiques (par exemple, 5×5) et des largeurs de bits. La matrice de poids est créée avec des valeurs entières aléatoires dans une plage définie par la largeur de bits spécifiée. Pour chaque cas de test, le tableau *OperationsArray* est généré, puis la multiplication optimisée est effectuée en utilisant une fonction décodant cette liste. La sortie attendue est ensuite comparée au résultat de la multiplication matrice-vecteur afin de garantir la justesse de l'algorithme.

Le tableau *OperationsArray* décrit à l'étape 4 (voir 3.4) constitue une modélisation du graphe computationnel. Celui-ci est implémenté en logiciel sous la forme d'une liste d'adjacences, où le poids de chaque sommet (nœud) représente l'indice (*Node ID*) d'une colonne dans le tableau. Chaque nœud (à l'exception des nœuds feuilles de l'arbre) correspond à une opération d'addition (+). Cette formalisation offre la possibilité de générer, de manière systématique, des modules multiplieurs en SystemVerilog pour différentes configurations de matrices/vecteurs.

3.5 Implémentations SystemVerilog

3.5.1 Architecture combinatoire

La création d'un module multiplieur en SystemVerilog peut désormais être réalisée grâce à la liste des opérations *OperationsArray*, qui contient toutes les informations nécessaires (additions et paires d'opérandes) pour permettre une implémentation matérielle combinatoire du circuit. Le diagramme ci-dessous (Fig. 3.2) en est un exemple. Ce circuit réalisé par le

script est une première implémentation purement combinatoire sans registres du multiplieur.

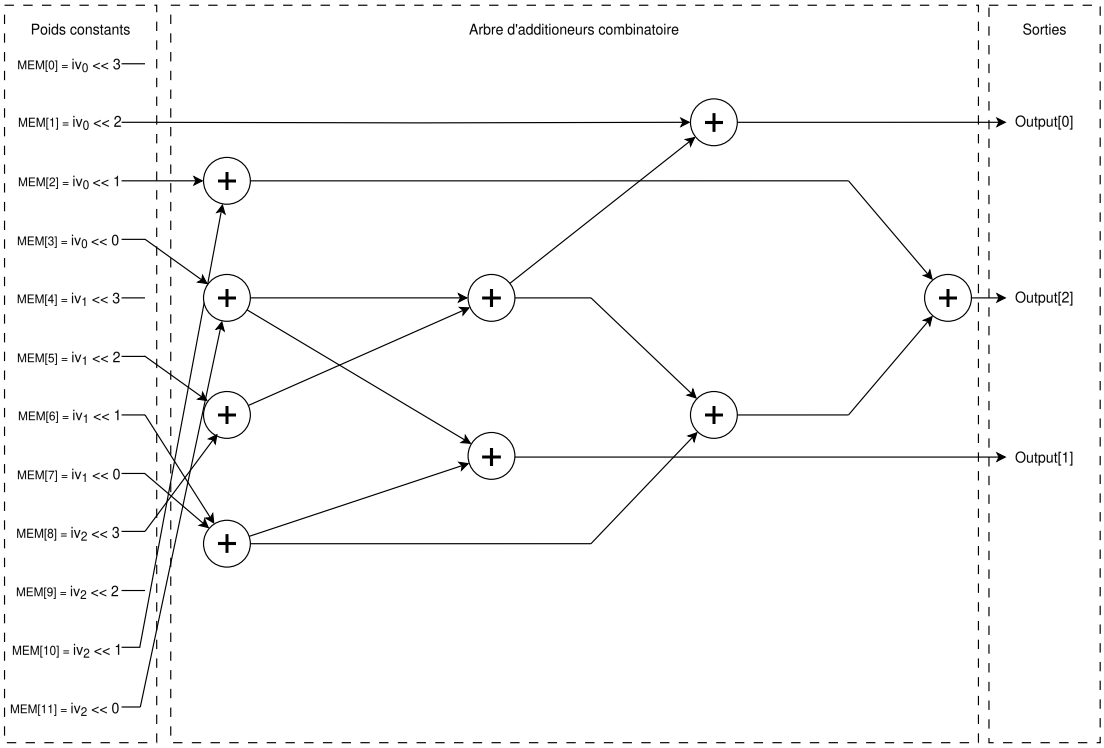


FIGURE 3.2 Graphe d'un multiplieur combinatoire (basé sur l'exemple initial 3.4)

Un générateur de modules en `SystemVerilog` a été développé pour traduire automatiquement la structure de données `OperationsArray` en une description RTL (voir algorithme 3). Ce script convertit les opérations définies dans l'implémentation de référence en `Python` en un ensemble d'assignations combinatoires, reflétant fidèlement la sémantique de la liste `OperationsArray`.

Le script prend en paramètre cette structure, la largeur des coefficients (par défaut fixée à 6 bits), ainsi que le nom du module. En sortie, il produit un fichier `SystemVerilog` capable d'exécuter la séquence d'opérations spécifiée. Les entrées et sorties sont les suivantes :

Entrées :

- **`OperationsArray`**.
- **`input_bitwidth`** (optionnel) : La largeur en bits des vecteurs d'entrée (par défaut à 6 bits).
- **`filename`** (optionnel) : Un suffixe pour nommer le fichier généré.

Sorties :

Algorithm 3 Script de génération de module SystemVerilog pour multiplication matricielle

procedure GenerateSystemVerilogModule(*OperationsArray*, *input_bitwidth*, *filename*)
    /* Calcul des paramètres du module */
    $rows, cols, weight\_bitwidth \leftarrow OperationsArray[0][0:3]$
    $output\_bitwidth \leftarrow weight\_bitwidth + input\_bitwidth + \lceil \log_2(cols - 1) \rceil$
    $mem\_size \leftarrow \max\{row[2] \mid row \in OperationsArray\}$
    $added\_statements \leftarrow$ ""
    /*Décodage de la liste d'opérations */
    for *row* **in** $OperationsArray[1:]$ **do**
        if $row[2] \neq -1$ **then**
            if $row[3]$ = "«" **then**
                $added\_statements \leftarrow added\_statements+$
                    "assign MEM[$row[0]$] = input_vector[$row[1]$] « $row[2]$;"
            else if $row[3]$ = "-«" **then**
                $added\_statements \leftarrow added\_statements+$
                    "assign MEM[$row[0]$] = -(input_vector[$row[1]$] « $row[2]$) ;"
            else if $row[3]$ = "+" **then**
                $added\_statements \leftarrow added\_statements+$
                    "assign MEM[$row[2]$] = MEM[$row[0]$] + MEM[$row[1]$] ;"
            else
                $added\_statements \leftarrow added\_statements+$
                    "assign output_vector[$row[0]$] = MEM[$row[1]$] ;"
            end if
        end if
    end for
    /*Déclarations de l'en-tête du module*/
    Specialiser le gabarit en remplacant les parametres du module par leurs valeurs $rows, cols, weight\_bitwidth, mem\_size, input\_bitwidths, output\_bitwidths$
    /*Écriture du fichier */
    Création du fichier avec les signaux déclaré
end procedure

— Un fichier SystemVerilog qui contient un module optimisé pour effectuer des multiplications matricielles. Le fichier est sauvegardé et porte un nom basé sur les dimensions de la matrice et d'autres paramètres.

— La largeur des poids (`weight_bitwidth`), extraite de l'élément de `OperationsArray`.

Le module ainsi généré peut ensuite être synthétisé pour produire une *netlist*, c'est-à-dire la liste des interconnexions d'un circuit implémentable sur FPGA, à partir de laquelle on peut estimer le coût en ressources logiques et les performances en termes de débit. Une explication détaillée des étapes effectuées par le script est présentée ci-dessous :

(a) Extraction des paramètres : L'algorithme commence par extraire les dimensions de la matrice et la largeur en bits des poids depuis la première ligne de `OperationsArray` :

- `rows1` et `cols1` sont respectivement le nombre de lignes et de colonnes de la matrice (extraits des deux premières valeurs de la première ligne).
- `weight_bitwidth` est la largeur en bits des poids (troisième valeur de la première ligne).

Ensuite, l'algorithme calcule la largeur en bits de la sortie (`output_bitwidth`). Cette valeur est égale à la somme de la largeur des poids (`weight_bitwidth`), de la largeur des entrées (`input_bitwidth`) et d'une valeur supplémentaire, calculée en fonction du nombre de colonnes de la matrice (`cols1`), représentant l'indexation binaire de la matrice.

(b) Calcul de la taille du tableau de signaux : Le nombre de signaux nécessaires (`mem_size`) est déterminé en prenant la valeur maximale de la troisième colonne de chaque ligne dans `OperationsArray`. Cette valeur représente le plus grand indice utilisé pour stocker un résultat intermédiaire. Ces résultats ne sont pas conservés dans une mémoire au sens logiciel du terme, mais sont directement véhiculés par des signaux (`wire`) dans une implémentation SystemVerilog purement combinatoire. Ainsi, `mem_size` correspond au nombre total de lignes de signal requis pour représenter l'ensemble des calculs intermédiaires dans le circuit.

(c) Création des assignations pour les opérations : L'algorithme parcourt les lignes de `OperationsArray` (à partir de la deuxième ligne) et génère des assignations SystemVerilog en fonction des opérations spécifiées dans la quatrième colonne :

- **Opérations de décalage à gauche (`«`)** : si l'opération est un décalage à gauche, l'algorithme crée une assignation Verilog pour appliquer ce décalage aux valeurs des entrées et les stocker dans la mémoire (`MEM`).
- **Opérations de décalage à gauche et inversion de signe (`-«`)** : si l'opération est un décalage à gauche négatif, l'algorithme effectue d'abord le décalage, puis applique un signe négatif à la valeur avant de l'assigner à la mémoire. Cette opération est réservé aux bits de poids fort, pour une représentation en complément a deux.
- **Opérations d'addition (`+`)** : si l'opération est une addition, l'algorithme additionne les valeurs de mémoire aux indices spécifiés et assigne le résultat à un nouvel emplacement dans la mémoire.

Si une ligne contient `"-1"` dans la troisième colonne, le nœud est assigné à `output_vector`, la sortie finale.

(d) Création du module SystemVerilog : Une fois que toutes les assignations sont générées, l'algorithme crée un module SystemVerilog en ajoutant les déclarations nécessaires :

— Il définit les paramètres du module, tels que le nombre de lignes `ROWS`, le nombre de colonnes `COLS`, la taille de la mémoire `MEM_SIZE`, la largeur `input_bit_width` des entrées et la largeur de la sortie `output_bit_width`.

— Il déclare les vecteurs d'entrée et de sortie, en précisant leur largeur en bits.

— Les assignations précédemment générées sont ajoutées au corps du module.

(e) Écriture du module SystemVerilog : Le code SystemVerilog généré est écrit dans un fichier avec l'extension `.sv`. La trace pour l'exemple précédent se trouve dans l'annexe A.

Bien que directe, l'architecture purement combinatoire s'avère rapidement impraticable pour des matrices de grande taille en raison de sa latence intrinsèquement élevée. Le calcul du produit scalaire implique de multiples additions de valeurs décalées, ce qui génère un chemin critique qui croît avec la taille de la matrice et la précision des coefficients. Cela limite sévèrement la fréquence d'horloge maximale. Cette croissance de la latence, combinée aux ressources limitées du FPGA, souligne la nécessité d'une approche capable de respecter des contraintes temporelles plus rigoureuses. Pour contourner ces limites, une conception **pipelinée** a été élaborée.

3.5.2 Architecture en arbre d'additionneurs pipeliné

L'idée fondamentale du pipeline consiste à décomposer le calcul en étapes plus petites et séquentielles, chacune effectuant une portion du travail, en alternant entre additionneurs et registres. En insérant des registres entre ces étapes, nous garantissons que les résultats intermédiaires sont stockés et disponibles dans les cycles d'horloge suivants, permettant à l'architecture de traiter de nouvelles entrées en continu. Cette approche réduit drastiquement le chemin critique et permet à la conception de traiter des dimensions matricielles plus importantes sans compromettre la fréquence d'horloge.

En décomposant le calcul en étapes, en insérant des registres et en gérant soigneusement la durée de vie des résultats, nous avons créé une architecture matérielle qui équilibre le débit, l'utilisation des ressources et les performances temporelles.

La méthodologie décrite ici, tout en combinant l'analyse des dépendances, le suivi des cycles et la génération automatisée de code SystemVerilog, fournit un cadre robuste pour concevoir des accélérateurs FPGA évolutifs.

Ce qui suit est une explication détaillée de la manière dont nous avons imposé le pipeli-

nage dans Vivado, contrôlé le cycle de vie des résultats intermédiaires et généré du code SystemVerilog optimisé pour implémenter la multiplication matrice-vecteur :

(a) Comptage des cycles et gestion de la durée de vie : Pour implémenter le pipelinage, la structure du tableau *OperationsArray* est étendue afin de mémoriser le premier et le dernier cycle pendant lesquels un résultat est valide. Ce suivi des cycles est essentiel pour contrôler à quel moment les résultats sont stockés dans les registres et quand ils peuvent être supprimés. La fonction `cycleCount` (Annexe B) augmente le tableau d'indices avec des informations de cycle. Pour chaque opération, elle détermine quand le résultat sera disponible et pendant combien de cycles d'horloge il doit être conservé :

1. **Opérations de décalage :** Elles sont immédiates, donc elles commencent et se terminent au cycle 0.
2. **Additions :** Le résultat devient disponible un cycle après le dernier des premiers cycles des opérandes. Le dernier cycle est défini comme la dernière étape où le résultat est utilisé.
3. **Sorties finales :** Le résultat de la dernière addition dans la chaîne est transmis au vecteur de sortie.

En suivant soigneusement les premiers et derniers cycles, nous éliminons l'utilisation inutile de registres. Les valeurs qui ne sont plus nécessaires ne sont pas reportées. Le nouveau format du tableau *OperationsArray* est le suivant (extrait de code 3.2).

```
[[3, 3, 4, 'parameters', 'FirstCycle', 'LastCycle'],
 [0, 0, 3, '<<', 0, -1],
 [1, 0, 2, '<<', 0, 2],
 [2, 0, 1, '<<', 0, 0],
 [3, 0, 0, '<<', 0, 0],
 [4, 1, 3, '<<', 0, -1],
 [5, 1, 2, '<<', 0, 0],
 [6, 1, 1, '<<', 0, 0],
 [7, 1, 0, '<<', 0, 0],
 [8, 2, 3, '<<', 0, 0],
 [9, 2, 2, '<<', 0, -1],
 [10, 2, 1, '<<', 0, 0],
 [11, 2, 0, '<<', 0, 0],
 [12, 11, 3, '+', 1, 1],
 [13, 8, 5, '+', 1, 1],
 [14, 7, 6, '+', 1, 2],
 [15, 13, 12, '+', 2, 2],
 [16, 10, 2, '+', 1, 3],
 [17, 15, 14, '+', 3, 3],
```

```
20  [18, 15, 1, '+', 3, 4],
21  [19, 14, 12, '+', 2, 4],
22  [20, 17, 16, '+', 4, 4],
23  [0, 18, -1, '=', 4, 4],
24  [1, 19, -1, '=', 4, 4],
25  [2, 20, -1, '=', 4, 4]]
```

Extrait de code 3.2 Tableau d'opérations *OperationsArray* augmenté par les cycles

La figure 3.3 illustre le graphe de dépendances orienté des opérations d'additions extraites du flot de calcul planifié sur un pipeline matériel basé sur l'exemple 3.4. Chaque nœud représente une opération `+`, annotée avec ses cycles d'activation `(FirstCycle, LastCycle)`, et les arêtes encodent les contraintes de précédence. Les sommets sont regroupés par cycle d'exécution et ordonnés localement pour réduire les croisements d'arêtes et améliorer la lisibilité du graphe. L'espacement vertical est proportionnel à la profondeur des dépendances, ce qui rend visible les chemin critique et la structure de l'arbre de réduction. La séparation verticale indique le cycle de création du nœud (qui représente le *timing* de l'opération). Un exemple de l'ordonnancement des opérations se trouve en annexe (Annexe E).

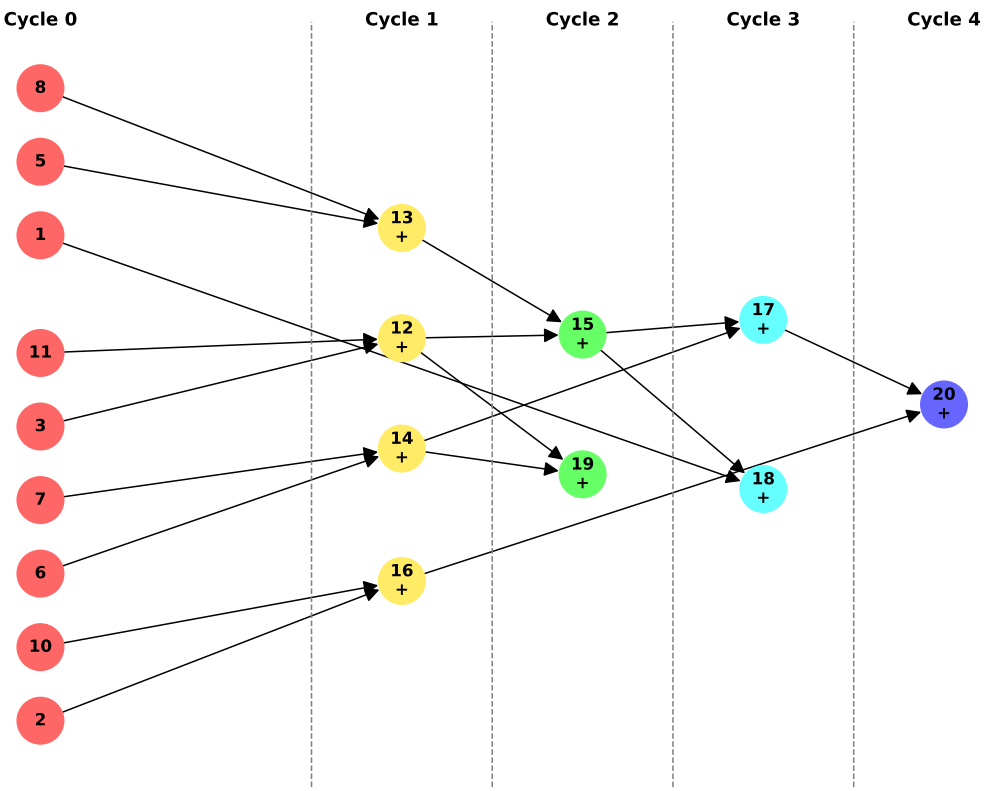


FIGURE 3.3 Graphe de dépendances d'un multiplieur pipeliné (basé sur l'exemple 3.4)

(b) Génération du module SystemVerilog basée sur les étages de pipeline : Une fois les informations de cycle ajoutées, le tableau d'indices *OperationsArray* augmenté est traduit en code SystemVerilog. Le calcul est divisé en étapes, chacune correspondant à un cycle d'horloge. La fonction regroupe les opérations par leur cycle de départ, génère les opérations arithmétiques requises et insère des registres pour stocker les résultats intermédiaires.

Pour chaque étape :

— **Les nouveaux résultats intermédiaires sont calculés et stockés**

— **Les valeurs encore vivantes sont propagées**

— **Les déclarations de registres et les réinitialisations sont gérées**

— **Les sorties sont assignées une fois le calcul terminé**

Le code SystemVerilog généré reflète la structure pipelinée (Annexe C) : chaque étape termine son opération en un cycle, avec des résultats intermédiaires transmis à l'étape suivante via des registres. Le résultat final apparaît à la sortie après la dernière étape, et de nouvelles entrées peuvent commencer à être traitées immédiatement, maximisant ainsi le débit. Le module final suit l'architecture illustrée à la figure 3.4.

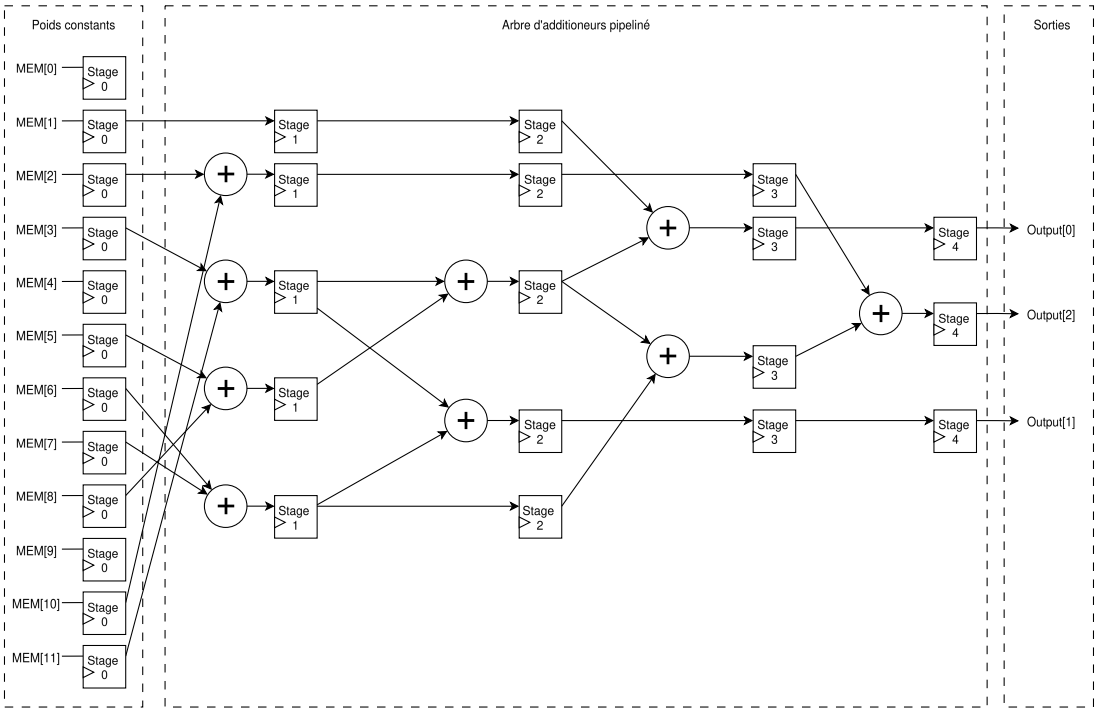


FIGURE 3.4 Graphe d'un multiplieur pipeliné (basé sur l'exemple 3.4)

Cette conception pipelinée améliore significativement l'évolutivité. Le délai n'est plus contraint par la taille de la matrice, car chaque étape a une complexité limitée. Des matrices plus

grandes se traduisent simplement par un plus grand nombre d'étapes, sans allonger le chemin critique. Cette structure permet donc de gérer des matrices de grande taille sans compromettre les performances, ce qui la rend bien adaptée aux applications en temps réel ou à haut débit.

En revanche, cette amélioration des performances temporelles s'accompagne généralement d'une augmentation de l'utilisation des registres, notamment pour stocker les résultats intermédiaires entre chaque étage de pipeline. Toutefois, en suivant soigneusement les durées de vie des signaux et en supprimant les résultats devenus inutiles, il est possible d'agréger certains de ces étages et de limiter la croissance des ressources nécessaires. Cela est particulièrement important dans les implémentations sur FPGA, où les ressources logiques et les registres sont limités.

3.5.3 Agrégation des étages de pipeline

Les architectures pipelinées divisent le traitement en étapes successives, synchronisées par des registres. Lorsqu'il est possible d'effectuer plus d'une addition avant le prochain front d'horloge, il peut être bénéfique d'ajouter un additionneur supplémentaire par étage, afin de réduire le nombre de registres nécessaires en fonction de la fréquence utilisée (voir Fig. 3.5).

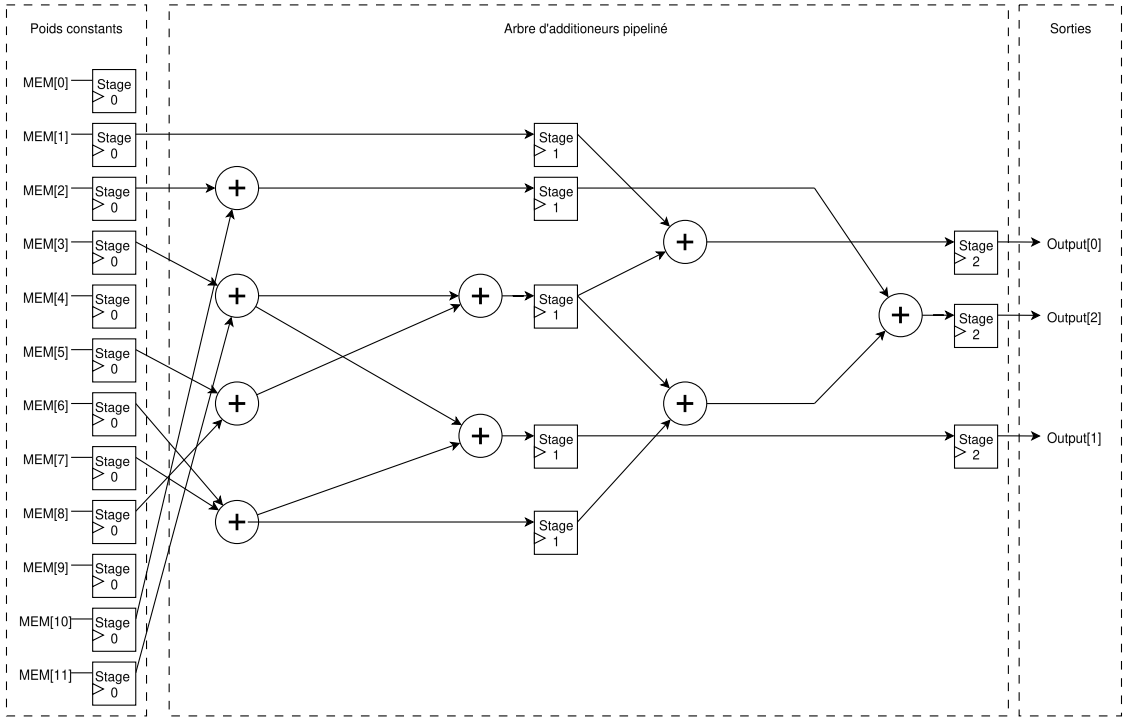


FIGURE 3.5 Agrégation des étages d'additionneurs.

À l'instar de l'approche précédente, la méthode suivante repose sur une segmentation partielle du calcul du résultat. Cette optimisation de la méthode pipelinée repose sur l'agrégation d'étages d'additionneurs alternés.

Pour une fréquence de 100 MHz, il est possible d'atteindre plus d'un additionneur par étage de pipeline, ce qui permet de réduire le nombre d'étages de registres requis pour calculer le résultat final.

Cette méthode peut être interprétée comme une forme de contraction d'arbre de dépendances, où les cycles impairs (contenant les opérations d'addition) sont fusionnés dans les cycles pairs qui leur succèdent. Cette stratégie, inspirée de l'algorithme de contraction d'arbres *rake and compress*.

Chaque niveau du graphe de dépendances est associé à un cycle d'horloge. Les opérations réalisées aux cycles impairs sont fusionnées dans les cycles pairs suivants, ce qui entraîne :

- la suppression des cycles impairs, qui sont absorbés dans les cycles pairs suivants ;
- la simplification du graphe, par réduction du nombre total d'étapes ;
- la substitution directe (*inline*) d'expressions intermédiaires dans l'occurrence suivante (par exemple, remplacer 17 dans $20 = 17 + 16$ par sa définition $17 = 15 + 14$) ;
- la propagation de copies (par réassignation) des signaux utiles pour les maintenir en vie jusqu'à leur dernière utilisation ;
- la mémorisation locale des résultats intermédiaires pour éviter leur recalcul.

Cette agrégation partielle réduit la latence totale et le coût en registres sans compromettre la fréquence cible, tant que les expressions résultantes respectent les contraintes temporelles de propagation combinatoire. Par exemple, une opération comme (a + b) + c initialement réalisée sur deux cycles (avec un registre entre les deux additions) peut être réécrite en un seul cycle en substituant directement a + b dans le second additionneur, à condition que le chemin critique reste dans les limites imposées par la fréquence. Un exemple de ce type d'ordonnancement est fournit dans l'Annexe F.

En pratique, cette logique suit une planification ALAP (*As Late As Possible*), où chaque signal est retardée jusqu'au cycle immédiatement précédant son utilisation, minimisant ainsi le nombre de cycles actifs tout en assurant la disponibilité des résultats. Cette transformation *bottom-up* ne s'effectue pas de manière récursive mais uniquement un niveau au-dessus à chaque étape, évitant la propagation incontrôlée des expressions.

Le dernier étage impair, une fois fusionné, devient la couche de sortie (*Output*) dans le circuit généré. Les nœuds de cette couche sont directement assignés à la sortie, sans passer par des registres supplémentaires. Le module est présenté en annexe (Annexe D).

Cette agrégation ne vise pas à réduire nécessairement l'utilisation des tables de correspondance, mais à optimiser la profondeur logique et à raccourcir le chemin critique. Elle permet donc un meilleur ordonnancement et une génération de code synthétisable respectant les contraintes temporelles, sans augmenter inutilement l'utilisation des registres.

3.5.4 Conclusion

Ce chapitre a présenté un algorithme original de multiplication par matrice à coefficients constants, conçu pour une implémentation efficace sur FPGA sans multiplieurs DSP dédiés. Cet algorithme simplifie le flot de données grâce à deux mécanismes : (i) une construction progressive de produits partiels à partir de la représentation binaire des coefficients, et (ii) la factorisation et la réutilisation systématique des sous-expressions communes, permettant une simplification des produits partiels et du flot de données.

L'implémentation en Python permet de générer le graphe computationnel et d'en extraire la séquence d'additions équivalente à la multiplication. L'ensemble du flot de conception a été automatisé. Cette séquence est ensuite transcrite en modules SystemVerilog synthétisables, structurés de manière à s'adapter à des tailles de matrices arbitraires.

À partir d'une même représentation intermédiaire du graphe de calcul (dans un format similaire à une liste d'adjacence), trois architectures distinctes sont générées : une version purement combinatoire, une version pipelinée, et une version pipelinée avec agrégation des étages impairs du pipeline. Dans les variantes pipelinées, le flot de conception inclut une gestion explicite des signaux internes, fondée sur le suivi des cycles, afin de déterminer les durées de vie et les alignements temporels des opérandes. Le processus de développement s'appuie sur une implémentation initiale en Python, utilisée comme référence pour valider les transformations successives jusqu'à la génération de modules synthétisables en SystemVerilog.

Les trois variantes architecturales générées à partir de l'algorithme proposé présentent des compromis distincts en termes de latence, de consommation de ressources et de débit. Pour en évaluer concrètement les performances et la robustesse face à la variation des paramètres de taille et de précision, une série d'expériences a été menée sur FPGA à partir de configurations représentatives. Le chapitre suivant est consacré à détaille cette évaluation, en commençant par des cas génériques, puis en examinant une application réaliste inspirée d'un réseau de neurones quantifié.

CHAPITRE 4 EXPÉRIENCES ET RÉSULTATS

4.1 Introduction

L'évaluation des performances d'un multiplieur repose sur l'analyse de plusieurs paramètres architecturaux, tels que la latence, l'utilisation des ressources logiques, la consommation énergétique et la précision des calculs. Dans cette section, une exploration des différentes architectures de multiplieurs est présentée. On examine d'abord des cas génériques qui serviront de base de comparaison, avant de s'intéresser à un cas pratique d'une couche linéaire issue d'un modèle de la littérature. Une étude est aussi portée sur une variante dans laquelle l'application des poids est approximé par une réduction de précision. Dans ce dernier cas de figure, la représentation du facteur d'échelle est limitée aux neuf bits les plus significatifs, ce qui permet de diminuer davantage l'empreinte matérielle.

4.2 Cas génériques

Dans cette section, on présente deux architectures multiplieurs qui serviront de référence afin de comparaison pour les trois modules présentée au chapitre 3.

4.2.1 Algorithme de multiplication binaire standard

La première implémentation de référence pour la multiplication matrice-vecteur (MatVecMult) est détaillée dans l'Algorithme 4. Il s'agit d'une implémentation directe de la définition mathématique de la multiplication matrice-vecteur, où chaque élément du vecteur de sortie $OutputVector[i]$ résulte de la multiplication des éléments du vecteur d'entrée par ceux de la ligne correspondante de la matrice, suivie de la sommation des produits.

Algorithm 4 Algorithme MatVecMult

Input: Matrice W (size ROWS x COLS),
$InputVector$ (length COLS)
Output: $OutputVector$

```
for i = 0 to ROWS - 1 do
    for j = 0 to COLS - 1 do
        OutputVector[i] += InputVector[j] × W[i][j]
    end for
end for
```

En `SystemVerilog`, cette référence implémente la multiplication d'un vecteur d'entrée par

une matrice entière W de dimensions ROWS×COLS en utilisant l'opérateur de multiplication standard (*) de Verilog et permet à l'outil de synthèse de Vivado d'optimiser l'opération. Un extrait de l'exemple est fourni en annexe (Annexe G).

4.2.2 Algorithme de multiplication par décalage

Le second algorithme de références réalise la multiplication matricielle en utilisant des décalages de bits et des additions. Elle est notamment décrite par Parhami [45]. L'algorithme *shift&add* décale à gauche un facteur $InputVector[j]$ d'un nombre de positions correspondant aux bits de valeur '1' du multiplicateur $W[i][j]$ et additionne les produits partiels. Cette approche est décrite dans l'algorithme 5 et sont implémentation dans l'annexe H.

Algorithm 5 Algorithme Shift&Add

Input: Matrice constante W (Taille ROWS x COLS),
$InputVector$ (longueur COLS)
Output: $OutputVector$

1: **for** $i = 0$ to ROWS - 1 **do**
2: **for** $j = 0$ to COLS - 1 **do**
3: Initialiser $P \leftarrow 0$
4: **for** $k = 0$ to (nombre de bits $W[i][j]$) **do**
5: **if** $W[i][j][k] = 1$ **then**
6: $P \mathrel{+}= InputVector[j] \ll k$
7: **end if**
8: **end for**
9: P$[i][j] \leftarrow P$
10: **end for**
11: $OutputVector[i] = \sum_{j=0}^{COLS-1}$ P$[i][j]$
12: **end for**

4.2.3 Description de la plateforme de test FPGA

Toutes les architectures ont été décrites et synthétisées pour un SoC Zynq-7000, dont la logique programmable est équivalente à celle d'un FPGA Xilinx Artix-7, en utilisant la stratégie de synthèse `area_optimized_high`. La version du synthétiseur Vivado utilisée est la `2023.2.1`. La puce FPGA ciblée est la `xc7z020clg400-1`. L'inférence automatique des blocs DSP a été désactivée afin de garantir que toutes les conceptions soient implémentées exclusivement à l'aide de LUT et d'additionneurs câblés en chaîne de retenue. Cette contrainte assure une comparaison équitable des résultats. Les modules synchrones (variantes pipelinés) ont été synthétisés pour une fréquence cible de 100 MHz.

4.2.4 Configuration expérimentale

La vérification fonctionnelle des modules a été effectuée à l'aide d'une suite de tests appliquée à un ensemble de matrices servant de prédicteurs linéaires pour un réseau de neurones profond (DNN). Un prédicteur linéaire est une fonction affine, définie comme une combinaison linéaire pondérée des variables d'entrée. Sa forme mathématique est la suivante :

$$f(i) = W_0 + W_1 x_{i1} + \cdots + W_p x_{ip} \tag{4.1}$$

où x_{ik} désigne la valeur de la k-ième variable du vecteur d'entrée associée à l'exemple i, et $W_0, W_1, \ldots, W_p$ sont les coefficients du modèle. En supposant que W soit une matrice de poids représentés en virgule fixe, le calcul d'une fonction de pré-activation pour une couche de DNN peut être simulé par la multiplication de cette matrice avec des vecteurs d'entrée aléatoires. Cette opération modélise de manière fidèle le comportement arithmétique de la phase d'inférence dans un environnement quantifié.

La configuration finale est composée des matrices de tailles 5×5, 10×10, 50×50, 100×100, et de coefficients matriciels représentés sous forme d'entiers aléatoires avec des largeurs de bits de 6, 8, 10, 12 et 16 bits. Les largeurs de mot inférieures à 6 bits n'apportent pas d'écarts significatifs en termes de consommation de ressources, tandis que des largeurs supérieures à 16 bits sont rarement justifiées en pratique, étant donné que des recherches existantes ont montré que de nombreux modèles, y compris MobileNet, BERT et ResNet, peuvent être quantifiés avec des poids de 8 bits tout en maintenant une précision à moins de 1% par rapport à leur version en virgule flottante [46] [47] [48]. Ces configurations couvrent ainsi l'essentiel des cas d'usage courants pour des modèles modernes déployés sur des plateformes à ressources contraintes.

4.2.5 Analyse des performances

Les résultats obtenus à partir des rapports de synthèse générés par l'outil Vivado sont présentés ci-dessous. On prête une attention particulière à la surface utilisée, notamment en termes d'éléments FPGA : tables de correspondance, bascules et bloc mémoire, et aux délais sur le chemin critique pour chaque modèle, afin d'en calculer la latence.

a) Performances matérielles :

Le tableau 4.1 présente la consommation des ressources LUT post-synthèse des nouvelles architectures proposées (Combinatoire, Pipeliné et Pipeliné Agrégé), comparées aux implémentations de référence Shift&Add et MatVecMult, en fonction des configurations de matrices

et des différentes largeurs de bits (6, 8, 10, 12, 16 bits).

Tableau 4.1 Consommation de LUT post-synthèse (kLUT)

Taille matrice	Algorithme	Largeur opérandes (bits)				
		6	8	10	12	16
5×5	Shift & Add	1,2	1,9	3,1	4,6	7,6
	MatVecMult	0,5	1,0	1,2	2,3	3,9
	Proposé - Combinatoire	**0,3**	**0,5**	**0,8**	**1,0**	**1,7**
	Proposé - Pipeliné	0,5	0,7	1,0	1,4	3,0
	Proposé - Pipeliné Agrégé	0,4	0,6	0,9	1,3	1,9
10×10	Shift & Add	5,5	8,5	13,2	19,5	31,8
	MatVecMult	2,2	3,7	5,9	9,6	15,5
	Proposé - Combinatoire	**1,5**	**2,3**	**3,0**	**4,2**	**6,9**
	Proposé - Pipeliné	2,1	3,4	5,0	7,2	11,1
	Proposé - Pipeliné Agrégé	1,5	2,4	3,6	4,6	7,6
50×50	Shift & Add	144,4	218,4	338,5	497,5	801,7
	MatVecMult	57,7	100,1	154,8	250,0	402,2
	Proposé - Combinatoire	**32,3**	**50,0**	**71,0**	**93,9**	**150,0**
	Proposé - Pipeliné	44,9	70,9	101,7	135,9	218,5
	Proposé - Pipeliné Agrégé	35,4	56,0	80,4	107,9	171,9
100×100	Shift & Add	591,5	887,0	1367,2	2012,7	3228,5
	MatVecMult	208,8	333,0	483,5	834,1	1414,3
	Proposé - Combinatoire	**122,6**	**186,2**	**262,5**	**350,9**	**565,7**
	Proposé - Pipeliné	171,2	265,3	378,5	507,4	819,5
	Proposé - Pipeliné Agrégé	139,1	215,2	306,7	412,0	666,9

Une analyse approfondie révèle que l'approche combinatoire obtient systématiquement les meilleures performances en termes d'économie de ressources LUT, particulièrement pour les grandes tailles de matrices et les largeurs de bits élevées. Par exemple, pour une matrice de taille 100×100 à 16 bits, l'approche combinatoire ne consomme que 565,7 kLUT, soit environ 5,7× moins que la méthode Shift&Add (3228,5 kLUT) et 2,5× moins que la méthode MatVecMult (1414,3 kLUT). Cette réduction significative s'explique par la capacité de l'algorithme combinatoire à exploiter pleinement la réutilisation des sous-expressions communes, limitant ainsi la duplication des circuits de calcul. De manière générale, la consommation moyenne de LUT pour l'algorithme combinatoire, à travers toutes les configurations étudiées, est d'environ 72% inférieure à celle de l'approche Shift&Add et d'environ 52% inférieure à celle de MatVecMult. Cette efficacité matérielle notable provient de l'élimination directe des multiplieurs explicites et de la gestion optimisée des expressions intermédiaires par la synthèse de Vivado, qui simplifie et réduit significativement la logique combinatoire.

La version pipelinée montre, quant à elle, une consommation légèrement plus élevée en ressources LUT comparativement à la version combinatoire (en moyenne environ 35% plus élevée

sur toutes les tailles), principalement due à l'ajout de registres intermédiaires (FF) pour segmenter les calculs en plusieurs étapes. Cette segmentation permet de réduire drastiquement les chemins critiques, augmentant ainsi la fréquence d'opération maximale du circuit, au prix d'une augmentation de la complexité matérielle. L'architecture pipelinée agrégée offre un compromis judicieux entre les deux implémentations, combinant une segmentation partielle des calculs avec une réutilisation accrue des expressions communes. Cette approche permet de récupérer environ 18% des ressources LUT utilisées par la version purement pipelinée, au prix d'une réduction de la fréquence maximale de 42,4% en moyenne(Tableau 4.3). Par exemple, pour une matrice 50×50 à 16 bits, la version pipelinée agrégée consomme 171,9 kLUT contre 218,5 kLUT pour la version entièrement pipelinée, soit une économie notable de 21%.

Les différences fondamentales entre les architectures asynchrones (combinatoires) et synchrones (pipelinées) résident dans la gestion du temps et des mémoires intermédiaires. Les versions combinatoires réalisent tous les calculs en un seul cycle d'horloge sans stockage intermédiaire explicite, ce qui conduit à une logique dense et à des chemins critiques longs. À l'inverse, les architectures pipelinées divisent le traitement en étapes successives synchronisées par des registres, permettant ainsi de réduire significativement la longueur des chemins critiques. Cette insertion de registres réduit la complexité combinatoire immédiate, mais introduit une augmentation de la consommation en ressources séquentielles.

L'impact de ces éléments de mémoire intermédiaires est notable. Leur utilisation conduit à une augmentation des ressources matérielles mais améliore considérablement les performances temporelles du circuit, rendant les architectures pipelinées adaptées aux applications nécessitant des fréquences élevées et un débit important. Le compromis réalisé ici concerne l'augmentation modérée des ressources LUT en échange d'une fréquence opérationnelle bien supérieure.

Cette segmentation du calcul modifie l'organisation des opérations logiques : certaines d'entre elles, qui pouvaient être agrégées et optimisées dans une même LUT en mode combinatoire, doivent être séparées lorsqu'elles sont pipelinées. De plus, certains résultats utilisés dans le traitement des étages en aval nécessitent la recopie des signaux vers l'étage suivant, ce qui accroît légèrement l'utilisation des ressources LUT et FF. Cette approche permet néanmoins d'atteindre des fréquences d'opération plus élevées. Les implémentations proposées offrent ainsi chacune des compromis spécifiques adaptés à des scénarios d'applications variés, où la sélection dépend directement des priorités entre la fréquence d'opération, la complexité matérielle et la consommation énergétique.

L'implémentation combinatoire de l'algorithme proposé nécessite entre 3,67× et 5,73× moins

de LUT que le multiplicateur Shift&Add (8 bits pour une matrice 10×10 et 12 bits pour une matrice 100×100, respectivement) et entre 1,49× et 2,68× moins de LUT que le multiplicateur MatVecMult (6 bits pour une matrice 10×10 et 16 bits pour une matrice 50×50, respectivement). Cette efficacité s'étend aux matrices plus grandes, l'algorithme optimisé réduisant de manière substantielle le nombre de LUT. Notamment, une matrice de taille 100×100 avec des données sur 16 bits nécessite 566 kLUT, contre 1 414 et 3 228 kLUT pour les approches MatVecMult et Shift&Add, respectivement. Ces résultats mettent en évidence les économies en LUT de l'algorithme proposé pour des largeurs de bits de plus en plus grandes.

b) Comparaison de latence :

Les versions combinatoires (dites « asynchrones », sans horloge dans le module) sont évaluées par le délai du chemin critique post-synthèse. Dans ce cas, Vivado n'effectue pas une analyse temporelle classique basée sur un lancement et une capture d'horloge, mais mesure uniquement le délai physique le plus long traversant la logique et le routage. On parle ainsi de délai de propagation maximal, qui correspond à la traversée d'un signal depuis une entrée jusqu'à une sortie sans contrainte temporelle imposée. Les résultats relatifs aux délais de propagation sur le chemin critique montrent des variations significatives en fonction des algorithmes et des configurations de matrices.

Tableau 4.2 Délai de propagation maximum sur le chemin critique post-synthèse pour les modules combinatoires (ns)

Taille matrice	Algorithm	Largeur opérandes (bits)				
		6	8	10	12	16
5×5	Shift & Add	**13,5**	14,6	**16,0**	**16,5**	**17,6**
	MatVecMult	14,4	**14,3**	16,7	17,0	17,9
	Combinatoire	15,0	15,4	18,0	17,7	23,3
10×10	Shift & Add	**14,7**	**15,4**	**16,7**	**17,3**	**18,2**
	MatVecMult	20,3	20,7	22,8	23,4	24,5
	Proposé - combinatoire	17,9	19,7	19,5	23,5	21,5
50×50	Shift & Add	**18,9**	**19,9**	**21,5**	**21,7**	**23,1**
	MatVecMult	68,5	69,3	71,8	71,9	73,4
	Proposé - combinatoire	21,6	23,2	24,5	25,2	26,2
100×100	Shift & Add	**19,6**	**20,6**	**21,9**	**22,4**	23,5
	MatVecMult	126,8	127,9	130,4	130,7	**22,7**
	Proposé - combinatoire	23,2	24,3	25,4	26,0	28,7

Les délais de propagation mesurés dans le tableau 4.2 révèlent que l'algorithme Shift&Add présente de manière générale des performances de propagation supérieures, avec des délais

plus courts. Par exemple, pour une matrice de taille 5×5, les délais mesurés varient de 13,5 ns à 17,6 ns pour des largeurs de bits de 6 à 16 bits. Cela montre une augmentation progressive des délais en fonction de la largeur de bits, bien que cette augmentation reste modérée.

L'analyse révèle également que les délais de propagation augmentent proportionnellement avec la taille de la matrice et la largeur des bits. Pour des matrices de petite taille comme 5×5, les délais sont relativement faibles et présentent des écarts réduits entre les différentes largeurs de bits. Cependant, à mesure que la taille de la matrice augmente, les délais de propagation deviennent plus sensibles à la largeur des bits. Pour les matrices plus grandes, comme 50×50 et 100×100, l'écart entre les algorithmes se creuse davantage. Par exemple, pour une matrice 100×100 à 16 bits, l'algorithme Shift&Add maintient un délai de 23,5 ns, tandis que MatVecMult atteint des délais beaucoup plus élevés, jusqu'à 130,7 ns.

L'algorithme combinatoire, bien que performant en termes de consommation de ressources, montre des délais de propagation relativement plus élevés. La performance de l'algorithme proposé est similaire à celle de l'approche Shift&Add, avec un léger surcoût en termes de délai. Ce surcoût est principalement dû à la conception de l'algorithme, qui impose la réutilisation de sous-expressions communes, ce qui entraîne un nombre plus élevé de niveaux sur le chemin critique. Cependant, malgré cet ajout, l'impact sur le délai global reste relativement faible. Par exemple, bien que l'algorithme Shift&Add affiche des délais plus courts dans certaines configurations, l'algorithme proposé parvient à maintenir des performances compétitives, même pour des matrices plus grandes et des largeurs de bits plus élevées, où les niveaux supplémentaires sur le chemin critique n'entraînent pas une dégradation significative des performances. Ainsi, l'algorithme proposé trouve un bon compromis entre la réduction de la consommation des ressources et le maintien de délais de propagation faibles.

Pour déterminer la fréquence maximale, le chemin critique ($\text{Délai}_{\text{chemin critique}}$) découle directement du WNS mesuré et de la période d'horloge fixée à 10 ns (tableau 4.3), selon l'équation :

$$\text{Délai}_{\text{chemin critique}} = T_{clk} - \text{WNS}$$

Par d'exemple, pour une matrice 5×5 avec des opérandes de 16 bits, la version proposée pipelinée possède un WNS de 6,227 ns, conduisant à un délai critique de :

$$10 - 6{,}227 = 3{,}773 \text{ ns}$$

et donc une fréquence maximale d'environ :

$$\frac{1}{3{,}773} \times 1000 \approx 265{,}1 \text{ MHz}$$

En comparaison, la version pipelinée agrégée avec le même paramétrage affiche un WNS de 5,453 ns, soit un délai critique légèrement supérieur de 4,547 ns, correspondant à une fréquence maximale inférieure d'environ 219,9 MHz.

Pour valider la fréquence maximale atteignable et s'assurer qu'elle n'est pas due à des approximations de l'outil de synthèse, une nouvelle synthèse pour chaque configuration a été réalisée avec une contrainte resserrée autour de $T_{\text{clk}} \approx 1/f_{\text{max}}$ jusqu'à obtenir un WNS presque nul. Cette vérification a confirmé que la fréquence mesurée est cohérente avec les performances attendues.

L'analyse du tableau révèle un compromis clair entre la fréquence maximale atteignable par chaque architecture pipelinée et la latence globale des calculs.

Tableau 4.3 Pire marge négative (*Worst Negative Slack*, WNS) pour les versions pipelinées (ns)

Taille matrice	Algorithme	Largeur opérandes (bits)				
		6	8	10	12	16
5×5	Proposé - Pipeliné	6,689	6,681	6,568	6,456	6,227
	Proposé - Pipeliné Agrégé	5,365	5,206	5,539	5,648	5,453
10×10	Proposé - Pipeliné	6,685	6,564	6,441	6,311	6,110
	Proposé - Pipeliné Agrégé	5,139	4,955	4,851	4,944	4,673
50×50	Proposé - Pipeliné	6,178	6,052	5,933	5,815	5,573
	Proposé - Pipeliné Agrégé	4,572	4,423	4,328	4,194	3,880
100×100	Proposé - Pipeliné	6,171	6,046	5,921	5,803	5,563
	Proposé - Pipeliné Agrégé	4,471	4,059	4,064	3,823	3,581

Les résultats montrent que la **version pipelinée standard** présente systématiquement des valeurs WNS supérieures à celles de la version pipelinée agrégée. Pour les matrices de petite taille (5×5 et 10×10), les valeurs WNS de la version standard oscillent entre 6,2 et 6,7 ns, tandis que la version agrégée affiche des valeurs comprises entre 4,7 et 5,6 ns.

Cette différence s'explique par la **densité combinatoire accrue** dans la version agrégée. En regroupant plusieurs opérations arithmétiques dans un même étage de pipeline, la version agrégée augmente la complexité du chemin critique local, réduisant ainsi la marge temporelle disponible (WNS plus faible).

La diminution de la fréquence d'horloge maximale lors du passge de la version pipelinée (264,6 MHz en moyenne) à celle agrégée (186,9 MHz en moyenne) illustre clairement l'impact du regroupement combinatoire des opérations au sein de chaque étage du pipeline. Cependant, le nombre d'étages nécessaires pour compléter une opération étant divisé par deux dans la version agrégée, la latence totale maximale s'en trouve réduite de manière notable. Par

exemple, en supposant que la version pipelinée simple requiert huit cycles pour compléter une opération complète, la latence serait alors 30,184 ns. La version agrégée n'exigerait alors que quatre cycles avec un délai critique plus long par cycle, aboutissant à une latence globale de 18,188 ns. La latence globale est donc ici réduite d'environ 39,7 %, illustrant l'avantage majeur de l'approche pipelinée agrégée.

Ce constat s'observe systématiquement pour toutes les tailles de matrices et largeurs d'opérandes. Ainsi, pour une matrice 100×100 avec des opérandes de 16 bits, la version agrégée présente une baisse de fréquence d'environ 30,9 % par rapport à la version pipelinée standard. Malgré cette diminution, la latence globale reste favorable à la version agrégée, avec une réduction proche de moitié du nombre de cycles nécessaires.

Les meilleurs gains observés sur l'ensemble des configurations testées indiquent une réduction de la latence globale comprise entre 26,7 % et 50 %, selon la taille de la matrice et la largeur des opérandes. Le gain minimal est obtenu pour la configuration 10×10 avec opérandes de 6 bits, tandis que le gain maximal est observé pour le cas 5×5 avec 12 bits.

En contrepartie, la fréquence maximale atteignable est réduite de 23 % à 31,7 % dans les configurations testées. Le plus faible recul en fréquence s'observe pour une matrice 5×5 avec des opérandes de 10 bits, tandis que la perte maximale est constatée pour le cas 10×10 avec opérandes de 6 bits.

Par conséquent, la conception pipeline agrégée démontre sa pertinence dans des contextes où la réduction de la latence globale importe davantage que la fréquence d'horloge maximale isolée.

4.3 Cas d'étude

Bien que la réalisation d'un réseau complet avec cette architecture soit possible, la démonstration à l'aide d'une seule couche est suffisante pour illustrer la génération de multiplieurs adaptés à la réalisation de modèles plus complexes. Cette étude de cas examine la couche finale d'un modèle issu des travaux de Jeziorek et al. [2], dont les poids ont été fournis par les auteurs. Le choix de ce modèle particulier est justifié par une collaboration initialement planifiée avec les auteurs de l'article cité. Bien que cette collaboration n'ait pas abouti aux résultats escomptés, les travaux préliminaires ont été exploités et sont présentés dans cette section à des fins d'analyse comparative et de validation des méthodes proposées. Le modèle étudié implémente un système de détection événementielle pour une plateforme SoC FPGA. Il s'agit d'un réseau de neurones convolutifs comprenant notamment des couches linéaires qui traitent les nœuds du graphe de manière pipelinée (Fig 4.1). Cette architecture est particu-

lièrement adaptée au traitement de données événementielles en temps réel, où les contraintes de performance sont significatives.

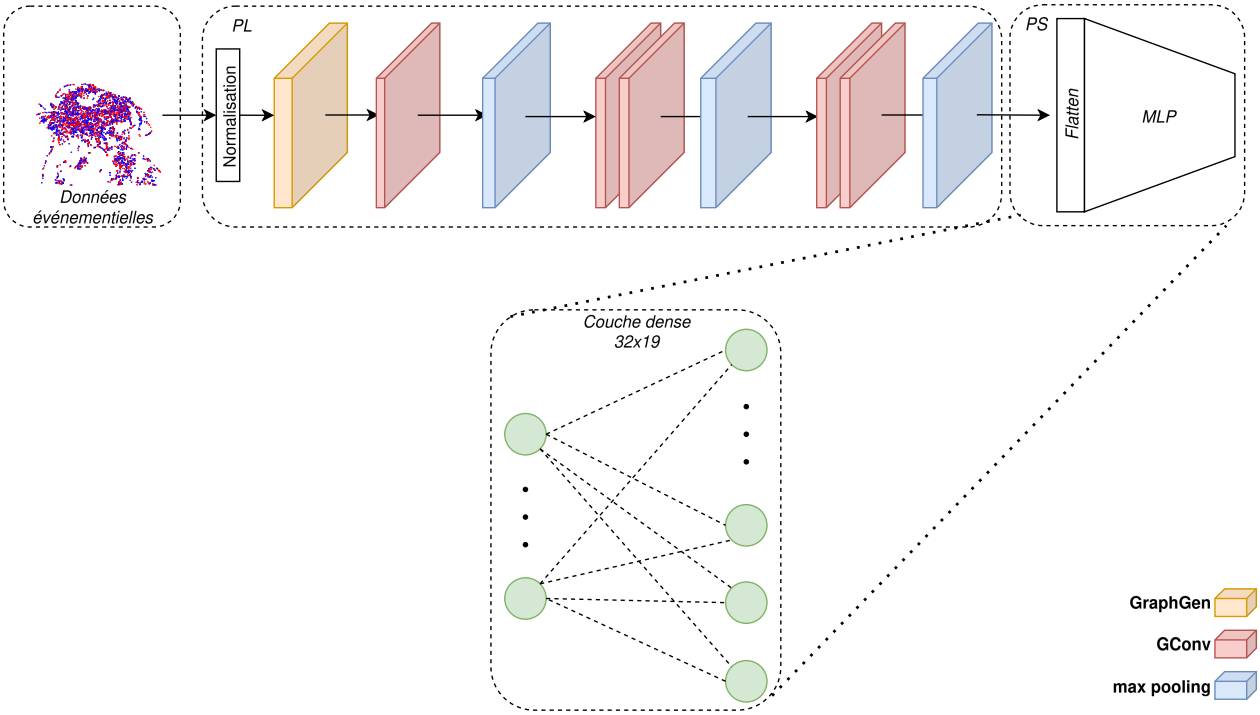


FIGURE 4.1 Architecture du réseau de neurones utilisée dans [2]

Le traitement effectué dans les couches linéaires repose sur une séquence d'opérations arithmétiques combinant multiplication, addition, mise à l'échelle et recentrage. Cette opération peut être formellement résumée par l'équation suivante, qui constitue le cœur fonctionnel du module :

$$\hat{y} = ((W \cdot \alpha + \beta) \cdot S) + Z \tag{4.2}$$

où W désigne la matrice constante de poids entiers signés, α le vecteur d'entrée, β le vecteur de biais, S un facteur de mise à l'échelle constant, et Z le point zéro de quantification.

Au lieu de multiplier par un facteur d'échelle en virgule flottante, on multiplie par un entier, puis on effectue un décalage binaire :

$$S \approx \frac{M}{2^{32}}$$

où S désigne le facteur d'échelle en FP32, et M (pour *Multiplier*) désigne le multiplicateur entier non signé sur 24 bits. L'opération de décalage à droite de 32 bits implémente une division par 2^{32}, transformant le résultat entier mis à l'échelle dans la plage appropriée.

Après avoir distribué le facteur d'échelle dans l'équation 4.2, on peut faire apparaître expli-

citement les termes $W \cdot M$ et $\beta \cdot M$ (équation 4.3) :

$$\Rightarrow \hat{y} = [(W \cdot M) \cdot \alpha + (\beta \cdot M)] \gg 32 + Z \tag{4.3}$$

Le décalage $\gg 32$ est appliqué à l'intérieur du module ; comme il s'agit d'un recâblage, cela n'ajoute aucun pas de cycles supplémentaires. La constante $M = 11\,639\,801$ représente une valeur d'environ 0,002710102 lorsqu'elle est interprétée dans un format à point fixe :

$$0{,}002710102 \approx 11\,639\,801 \times 2^{-32} \tag{4.4}$$

Les nouveaux termes $W \cdot M$ et $\beta \cdot M$ peuvent être pré-calculés en logiciel avant la synthèse du module, ce qui permet de concevoir un multiplieur intégrant directement cette opération, au prix toutefois d'une augmentation de la magnitude des poids. Cette amplification peut entraîner un accroissement de la logique utilisée, et donc une hausse du coût matériel.

Nous évaluons trois variantes d'implémentations sur une partie de l'équation 4.3 avec notre générateur de multiplieurs :

1. **Multiplication directe ($W \cdot \alpha$)** : la matrice de poids W, codée sur 9 bits signés, est utilisée sans transformation. Le multiplieur réalise uniquement le produit entre W et le vecteur α. Cette approche isole la multiplication des activations, ce qui réduit la complexité arithmétique au sein du multiplieur mais nécessite une logique supplémentaire pour intégrer les opérations de mise à l'échelle, de décalage et d'ajustement par Z en aval en dehors du module.
2. **Poids pré-multipliés par M** : la matrice W est multipliée par le facteur d'échelle entier M en logiciel, produisant une matrice constante $W \cdot M$ intégrée directement dans le module. Cette fusion permet de supprimer, au niveau matériel, l'opération de redimensionnement, en déléguant ce coût au prétraitement. L'algorithme réalise alors $(W \cdot M) \cdot \alpha$, suivi du décalage $\gg 32$ et de l'ajout de Z. L'opération principale est ainsi simplifiée dans le flot matériel, mais au prix d'une augmentation significative de la taille des opérandes, car les produits intermédiaires avec $M = 11\,639\,801$ produisent des coefficients codés sur 32 bits.
3. **Poids approximés ($W \cdot M_{\mathbf{approx}}$)** : l'approximation est appliquée uniquement au facteur M. On extrait ses 9 bits de poids fort et on met les bits restants à zéro, ce qui donne un M_{approx} plus simple à manipuler. Ce facteur est ensuite utilisé pour recalculer $W \cdot M_{\text{approx}}$ en logiciel. Le multiplieur exécute alors l'opération $(W \cdot M_{\text{approx}}) \cdot \alpha \gg 32$. Bien que les poids approximés soit toujours représenté sur 32 bits, le nombre effectif

de termes non nuls à additionner diminue, puisque les 15 bits de poids faibles sont systématiquement nuls. Cette méthode réduit la complexité arithmétique interne, tout en maintenant un encodage compatible avec la variante précédente.

Les performances post-synthèse de ces trois stratégies sont résumées dans le tableau 4.4.

Architecture	W **(9bit)**		$W \cdot M$ **(32bit)**		$W \cdot M_{approx}$ **(32bit)**	
	LUT	FF	LUT	FF	LUT	FF
Shift&Add	73 489	–	260 317	–	262 823	–
MatMulVec	26 576	–	108 456	–	44 797	–
Proposé - Combinatoire	12 747	–	75 067	–	31 361	–
Proposé - Pipeliné	20 459	34 653	116 339	201 814	53 093	102 091
Proposé - Pipeliné Agrégé	16 437	18 582	90 707	106 240	42 616	52 604

TABLEAU 4.4 Consommation de ressources post-synthèse pour les modules approximé et couche linéaire (32x19)

D'après les résultats relevés, la multiplication directe (W) affiche la plus faible empreinte matérielle, mais requiert un traitement supplémentaire externe du facteur d'échelle, augmentant ainsi potentiellement la complexité globale du système.

Le tableau 4.4 révèle aussi une tendance claire : dès qu'on fusionne le facteur d'échelle avec les poids, la largeur des opérandes grimpe à 32 bits et la surface logique explose. Le passage de poids codés sur 9 bits à des coefficients sur 32 bits induit une complexité arithmétique nettement plus élevée. Dans la version pipelinée, cette stratégie consomme 116 kLUT et 201 kFF, soit respectivement $\times$5,7 et $\times$5,8 plus que la version directe. Cette tendance est observée à tous les niveaux, même dans le cas combinatoire, où la version $W \cdot M$ utilise près de six fois plus de ressources que la multiplication directe ($W \cdot \alpha$).

Les deux versions pré-multipliées engendrent une consommation accrue des ressources FPGA, particulièrement pour les versions pipelinées : $\times$5,7 en LUT et $\times$5,8 en FF pour la version pipelinée ; $\times$5,5 en LUT et $\times$5,7 en FF pour la version pipelinée agrégée). L'utilisation de $W \cdot M_{\text{approx}}$ permet de retrouver un compromis efficace entre coût logique et intégration fonctionnelle, divisant par plus la moitié le nombre de LUT et de FF. Ces gains se confirment dans la version agrégée, qui atteint une réduction de $\times$2,1 en LUT et $\times$2 en FF. Dans le cas de l'architecture $Shift\&Add$, le simple passage de W à $(W \cdot M)$ fait bondir l'empreinte de 73 kLUT à plus de 260 k LUT, soit un facteur $\times$3,5. L'approximation de M n'apporte en revanche aucun bénéfice particulier, car le synthétiseur est forcé de générer une chaîne d'additions en série tout aussi profonde, et ne peut pas simplifier certains décalages ou additions entre colonnes.

Afin de consolider l'analyse précédente sur les cas génériques, l'ensemble des résultats montre que les multiplieurs proposés réduisent significativement la consommation logique par rapport aux architectures de référence. Pour la multiplication directe (W), les gains sont substantiels : la version combinatoire proposée consomme $\times$5,8 moins de LUT que *Shift&Add* (12 vs 73 kLUT) et $\times$2,1 moins que *MatMulVec* (12 vs 26 kLUT). Les versions pipelinées, bien qu'utilisant davantage de ressources que la version combinatoire, restent compétitives avec respectivement 20 kLUT et 16 kLUT pour les versions pipelinée et pipelinée agrégée. Même avec l'intégration du facteur d'échelle dans le flot de calcul ($W \cdot M$), les algorithmes proposés maintiennent leur avantage : la version combinatoire utilise $\times$3,5 moins de LUT que *Shift&Add* (75 vs 260 kLUT) et $\times$1,4 moins que *MatMulVec* (75 vs 108 kLUT), tandis que les versions pipelinées conservent des gains significatifs de $\times$2,2 à $\times$2,9 face à *Shift&Add*. Ces gains se confirment également avec l'approximation ($W \cdot M_{\text{approx}}$), où l'écart face à *Shift&Add* atteint un facteur $\times$8,4 pour la version combinatoire (31 vs 262 kLUT), démontrant la robustesse de l'approche proposée sur l'ensemble des configurations testées.

En résumé, l'intégration directe de M alourdit considérablement les modules, mais permet de simplifier le flot en aval. À l'opposé, la multiplication directe conserve une logique compacte mais impose une charge de traitement reportée. L'approche approximative constitue un équilibre entre coût logique et intégration fonctionnelle, avec des gains visibles sur l'ensemble des architectures à l'exception module $Shift\&Add$.

CHAPITRE 5 DISCUSSION

L'analyse des résultats a permis d'identifier clairement les avantages spécifiques de chaque variante de l'architecture proposée. Notamment, l'approche combinatoire s'avère particulièrement efficace en termes de surface, atteignant des réductions jusqu'à 3,5× à 5,7× inférieure à l'approche Shift&Add. Cependant, cette approche présente des limites claires en termes de latence, due à la croissance rapide du chemin critique à mesure que la complexité de la matrice augmente, ce qui limite sa fréquence d'opération maximale. Ces résultats suggèrent que l'architecture combinatoire est particulièrement adaptée aux scénarios où l'espace est fortement contraint et où la fréquence d'opération élevée n'est pas une priorité absolue, tels que les dispositifs embarqués à faible consommation, des capteurs autonomes fonctionnant sur batterie ou encore les applications IoT (Internet des Objets) nécessitant une minimisation extrême des ressources matérielles pour prolonger l'autonomie énergétique.

L'architecture pipelinée, en revanche, offre une performance nettement supérieure en termes de fréquence d'opération, en permettant une réduction drastique du chemin critique grâce à l'insertion de registres entre les étapes de calcul intermédiaires. Cependant, cette amélioration de la fréquence d'opération se fait au coût d'une augmentation d'environ 50% à 60% des ressources LUT utilisées, comparativement à l'approche combinatoire. Cette pénalité est particulièrement marquée pour les matrices de grande taille et les largeurs de bits importantes, comme le montre la figure 5.1, ce qui limite l'efficacité spatiale de l'approche purement pipelinée.

Face à cette limitation, l'architecture pipeline agrégée se présente comme un compromis très avantageux, puisqu'elle récupère typiquement entre 5 et 25% des LUT nécessaires par rapport à l'architecture purement pipelinée, tout en maintenant des performances en fréquence très proches. Au-delà de l'équilibre entre ressources et performance, cette approche présente également des compromis intéressants en termes de consommation énergétique et de complexité de conception. En effet, l'architecture agrégée, en réduisant le nombre de registres intermédiaires et les étages logiques nécessaires, diminue potentiellement la consommation énergétique liée au basculement fréquent des registres. De plus, la simplification relative de la gestion des ressources FPGA contribue à réduire la complexité globale de la conception matérielle, facilitant ainsi l'intégration et la mise en œuvre pratique. Ce compromis est particulièrement utile dans les scénarios applicatifs où une fréquence modérée, une consommation énergétique maîtrisée et une complexité de conception raisonnable sont des critères essentiels.

En parallèle, cette structure pipelinée agrégée permet une réduction directe de la latence

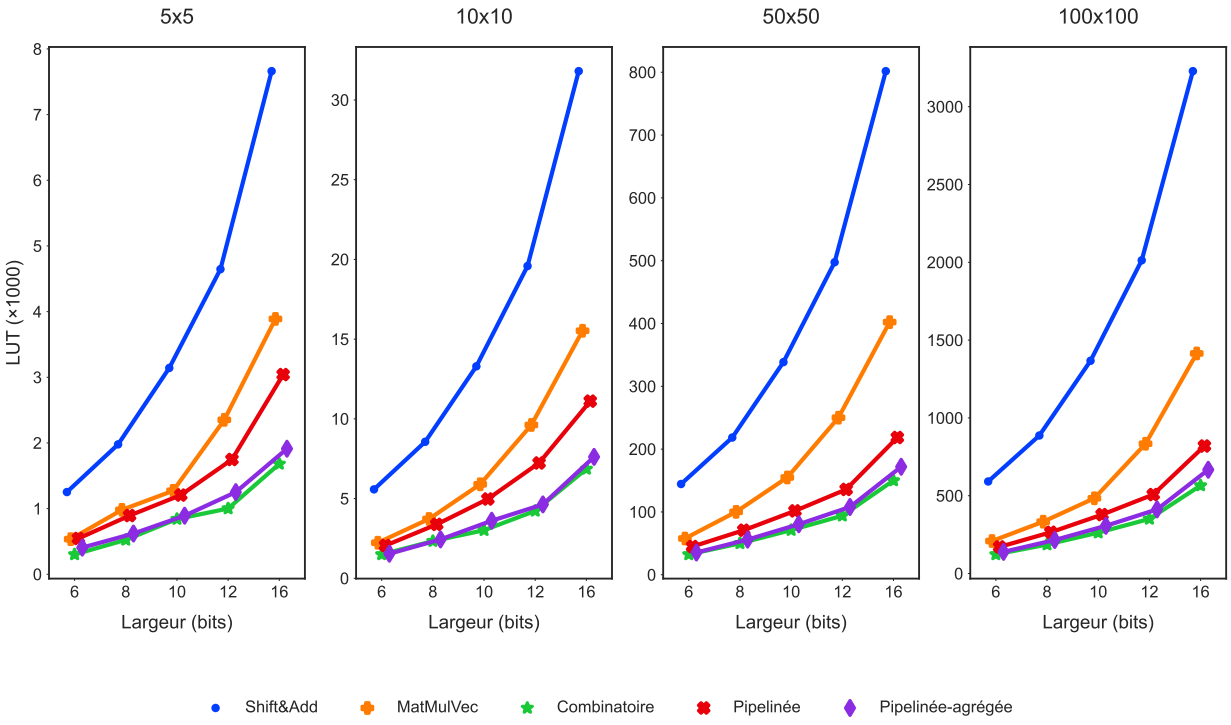


FIGURE 5.1 Consommation de LUT à travers les largeurs de bits pour quatre tailles de matrices (5×5, 10×10, 50×50, 100×100) comparant les implémentations de référence (Shift&Add, MatVecMult), combinatoire, pipelinée et pipelinée-agrégée.

globale, malgré une légère hausse du délai de cycle. Les gains mesurés vont de 26,7 % à 50 % selon la taille de la matrice et la largeur des opérandes. Ce bénéfice provient d'un regroupement des opérations arithmétiques sur moins d'étapes, ce qui permet de réduire le nombre total de cycles nécessaires pour compléter le calcul. Même si la fréquence maximale chute de 23 % à 31,7 %, la diminution du nombre de cycles compense cette perte. Par exemple, une opération nécessitant huit cycles à 4,1 ns dans un pipeline classique peut être exécutée en quatre cycles à 6 ns dans la version agrégée, ramenant la latence globale de 32,8 ns à 24 ns. Cette amélioration rend l'architecture particulièrement adaptée aux systèmes orientés débit, où la minimisation du temps d'exécution global prime sur l'optimisation locale du cycle d'horloge.

Un point particulièrement notable mis en évidence par nos résultats concerne la tendance observée dans les délais de propagation sur le chemin critique (Tableau 4.2). Cette tendance semble perturbée dans le cas de l'approche MatVecMult pour la configuration de matrice 100×100 et largeur de bits 16. Cette rupture de tendance peut s'expliquer par la nature particulière des coefficients constants utilisés lors des tests. En effet, certaines constantes

peuvent engendrer des structures combinatoires nettement plus complexes lors de la synthèse par Vivado, entraînant une augmentation non linéaire des délais de propagation, phénomène également observé dans la littérature, notamment par Aksoy et al. [28] et Kumm et al. [34] , qui mettent en évidence l'augmentation significative de la complexité combinatoire lors de l'implémentation matérielle des multiplications impliquant des constantes complexes ou particulièrement denses en termes de bits non nuls.

Un autre aspect important discuté dans cette étude concerne la comparaison directe avec les solutions optimales issues de la littérature. Bien que notre approche montre clairement son intérêt dans les scénarios de matrices grandes (par exemple 50×50 ou 100×100), une comparaison directe avec des solutions optimales déjà présentes dans la littérature est difficile à établir. En effet, les solutions optimales disponibles reposent souvent sur des recherches exhaustives adaptées uniquement aux petites tailles de matrices, typiquement entre 3×3 et 10×10. Kumm et Zipf [34] on par exemple on collecte un ensemble de 8 matrices provenant de la littérature allant de 3×3 a 11×16.

Notre algorithme en revanche est spécifiquement conçu pour être évolutif, permettant de gérer efficacement des tailles nettement supérieures. Cet aspect évolutif s'avère particulièrement pertinent dans des applications réelles exigeantes en termes de calcul, telles que les réseaux neuronaux profonds, les systèmes embarqués de traitement d'images en temps réel ou encore les applications de cryptographie qui nécessitent fréquemment des multiplications matricielles à grande échelle. Par conséquent, les comparaisons directes avec ces travaux optimaux restent délicates, et notre contribution réside plutôt dans la capacité à traiter efficacement et à grande échelle des problèmes pratiques qui demeurent hors de portée pour ces solutions optimales exactes, ouvrant ainsi de nouvelles perspectives d'applications industrielles et technologiques.

Enfin, l'estimation précise de la consommation énergétique sur FPGA demeure un défi technique. Les outils comme *Vivado* reposent souvent sur des approximations statistiques, utilisant par défaut un taux de commutation arbitraire (typiquement de 12,5 %). Cette valeur, rarement représentative des activités réelles, affecte directement la précision des estimations, puisque la puissance dynamique dépend du taux de commutation selon la relation :

$$P_{\text{dynamic}} = \alpha C V^2 f$$

où α désigne le taux de commutation moyen, C la capacité commutée, V la tension, et f la fréquence. En l'absence de profils d'activité détaillés, les estimations restent souvent inadéquates, limitant les comparaisons fiables entre architectures. Des méthodes alternatives, telles que la simulation détaillée par analyse temporelle des commutations de signaux spécifiques ou l'emploi de modèles énergétiques basés sur des mesures expérimentales effectuées sur

des prototypes physiques, pourraient améliorer significativement la précision des estimations. Ainsi, les évaluations énergétiques présentées ici restent qualitatives, reposant sur l'hypothèse raisonnable que la consommation énergétique est proportionnelle au nombre de LUT utilisées, mais des études futures pourraient explorer ces méthodes avancées afin d'obtenir des estimations énergétiques plus fiables et réalistes.

En synthèse, notre étude met en lumière les avantages distinctifs et les limitations inhérentes de chaque variante architecturale proposée, et souligne clairement les scénarios applicatifs dans lesquels chacune des variantes peut être préférentiellement exploitée. Cette analyse approfondie et détaillée permet d'établir solidement les fondements pour l'application pratique et l'exploitation efficace de l'algorithme proposé dans des contextes réels à forte contrainte de ressources matérielles.

CHAPITRE 6 CONCLUSION

6.1 Synthèse des travaux

Ce mémoire présente un algorithme optimisé de multiplication par matrice constante destiné aux implémentations FPGA. Il offre une plus grande réutilisation des résultats partiels, permettant ainsi de traiter des problèmes de taille supérieure d'au moins un ordre de grandeur que ceux précédemment abordés dans la littérature. Cette approche permet de réaliser d'importantes économies en ressources matérielles, principalement en termes de LUT, comparativement aux approches traditionnelles comme Shift&Add et MatVecMult. L'algorithme exploite la réutilisation itérative des sous-expressions communes pour minimiser le nombre d'additions nécessaires, réduisant ainsi considérablement la complexité combinatoire du circuit final. L'analyse comparative approfondie menée sur des matrices de tailles variées (5×5 à 100×100) a clairement démontré l'intérêt de notre méthode, particulièrement dans les contextes où les contraintes matérielles sont strictes.

Les expériences menées ont porté sur plusieurs tailles de matrices (de 5×5 à 100×100) et des largeurs de bits variées (de 6 à 16 bits). Trois variantes architecturales distinctes ont été analysées en profondeur : une architecture combinatoire, une architecture pipelinée et une architecture pipelinée agrégée. L'architecture combinatoire s'est distinguée par sa faible consommation en ressources, ce qui en fait une solution idéale pour les systèmes embarqués et les dispositifs IoT où l'économie d'énergie est primordiale. À l'opposé, l'architecture pipelinée a démontré des performances élevées en fréquence, la rendant particulièrement adaptée aux applications exigeantes en débit. L'approche pipeline agrégée a permis d'établir un compromis équilibré entre consommation de ressources et performances temporelles, récupérant de 5 à 25% des ressources utilisées par l'architecture purement pipelinée.

Au-delà des gains en surface, nos résultats montrent également une amélioration notable de la latence d'exécution pour l'architecture pipelinée agrégée. En regroupant les opérations arithmétiques sur un nombre réduit d'étapes, cette variante permet une réduction de la latence globale comprise entre 26,7 % et 50 %, selon les dimensions de la matrice et la largeur des opérandes. Bien que cette optimisation s'accompagne d'une baisse modérée de la fréquence maximale (de l'ordre de 23 % à 31,7 %), l'économie en nombre de cycles compense largement ce recul, ce qui renforce la pertinence de cette architecture dans les scénarios où le débit d'exécution prime sur la vitesse d'horloge individuelle.

Ces résultats démontrent clairement que notre solution offre des économies substantielles

de LUT allant jusqu'à un facteur de 5,7× par rapport aux approches classiques telles que Shift&Add, tout en maintenant des performances temporelles compétitives.

La réduction significative des ressources LUT rend l'algorithme proposé viable pour le déploiement de réseaux de neurones dans des applications à ressources limitées telles que les appareils connectés (IoT), les téléphones intelligents et la conduite autonome.

Ce travail de recherche a également abouti à une publication dans une conférence internationale, dont la présentation a été réalisée en août 2024, témoignant ainsi de l'intérêt et de la pertinence scientifique de nos résultats.

6.2 Limitations de la solution proposée

Bien que les résultats obtenus soient prometteurs, plusieurs limitations de cette recherche doivent être soulignées et examinées attentivement pour orienter les travaux futurs.

Tout d'abord, la sensibilité aux coefficients constants des matrices utilisées constitue une limite notable. Cette variabilité des délais de propagation liée aux constantes employées suggère la nécessité de méthodes robustes permettant d'évaluer systématiquement les performances temporelles pour une grand ensemble de coefficients constants. Des études futures pourraient, par exemple, explorer des techniques statistiques avancées ou des méthodes de validation approfondies pour mieux anticiper ces variations.

Ensuite, bien que la réutilisation des sous-expressions diminue efficacement le nombre d'opérations nécessaires, elle introduit souvent une profondeur logique supplémentaire. Cette complexité combinatoire accrue peut constituer un frein dans les applications extrêmement exigeantes en fréquence. Pour pallier cette limitation, des approches hybrides ou des stratégies avancées de pipelinage pourraient être explorées afin de mieux gérer la profondeur logique sans compromettre significativement l'efficacité des ressources.

Par ailleurs, l'approche combinatoire pure ne supporte pas intrinsèquement le pipeline, limitant ainsi son application directe à des scénarios à haut débit. Bien que partiellement résolue par la variante pipelinée agrégée, une analyse plus poussée pourrait conduire à des architectures plus adaptées capables de basculer intelligemment entre différentes stratégies en fonction des contraintes opérationnelles en temps réel.

Enfin, l'estimation précise de la consommation énergétique sur FPGA demeure un enjeu majeur. Les outils de simulation actuels reposent principalement sur des modèles simplifiés ne capturant pas complètement les dynamiques fines de commutation. La mise en place de méthodes avancées telles que la simulation basée sur des mesures physiques réelles ou des modèles énergétiques détaillés pourrait offrir une estimation plus fidèle et permettre une

optimisation énergétique plus poussée.

6.3 Perspectives et pistes de recherche futures

Dans la continuité de ces travaux, plusieurs axes d'exploration et défis restent à aborder.

Au vu des résultats obtenus et des limitations identifiées, les recherches futures devront se concentrer sur une analyse comparative approfondie avec les architectures concurrentes existantes.

Un autre piste de recherche concerne le développement d'architectures adaptatives hybrides capables de choisir dynamiquement l'approche optimale (combinatoire ou pipelinée) selon les besoins spécifiques de performance et les contraintes opérationnelles. De telles architectures pourraient optimiser dynamiquement les compromis entre efficacité matérielle, performance temporelle et consommation énergétique.

L'optimisation préalable des coefficients des matrices pourrait également constituer une voie fructueuse. En étudiant en détail l'impact spécifique des propriétés mathématiques des coefficients sur la performance temporelle, il serait possible d'élaborer des techniques de prétraitement optimisées pour améliorer davantage les résultats obtenus lors de la synthèse matérielle.

Par ailleurs, étendre l'algorithme à une précision numérique supérieure (par exemple 32 bits ou davantage) pourrait répondre aux besoins croissants de certaines applications scientifiques et industrielles nécessitant une précision numérique élevée.

Enfin, une analyse approfondie sur des FPGA de dernière génération comme les Xilinx UltraScale+ ou Versal pourrait être effectuée pour mieux comprendre comment les avancées technologiques récentes pourraient influencer les performances obtenues, notamment en termes d'efficacité énergétique et d'utilisation des ressources matérielles.

En résumé, les résultats présentés mettent en évidence le potentiel d'une réduction significative du nombre de LUT et d'une meilleure efficacité dans le traitement de grands jeux de données et de matrices, suggérant une application prometteuse dans l'implémentation de réseaux de neurones profonds sur FPGA. Bien que les résultats actuels soient encourageants, le travail d'affinage de notre approche et son optimisation pour des applications plus étendues reste à poursuivre.

RÉFÉRENCES

[1] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," dans *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, févr. 2014, p. 10–14, iSSN : 2376-8606. [En ligne]. Disponible : https://ieeexplore.ieee.org/document/6757323

[2] K. Jeziorek, P. Wzorek, K. Blachut, A. Pinna et T. Kryjak, "Embedded Graph Convolutional Networks for Real-Time Event Data Processing on SoC FPGAs," juin 2024. [En ligne]. Disponible : http://arxiv.org/abs/2406.07318

[3] A. Rassau *et al.*, "Extrem-edge : A flexible edge computing architecture," *Sustainable Computing : Informatics and Systems*, vol. 35, 2022.

[4] J. Zhang *et al.*, "Energy-aware resource allocation for iot edge computing," *Future Internet*, vol. 17, n°. 1, 2023.

[5] M. Ali *et al.*, "A survey of smart healthcare monitoring systems using iot," *IEEE Access*, vol. 7, 2019.

[6] M. Tavakkoli *et al.*, "Energy optimization for connected and autonomous vehicles," *Energy*, vol. 295, 2024.

[7] K. Liu et K. Kockelman, "Energy and emissions implications of autonomous vehicles," University of Texas at Austin, Rapport technique, 2019.

[8] R. Elrofai *et al.*, "Cost-effective planning for autonomous vehicle fleets," *Transportation Research Part C*, vol. 134, 2021.

[9] T. Nguyen *et al.*, "Power consumption analysis of microsoft hololens 2," *Energies*, vol. 17, n°. 3, 2024.

[10] D. S. Dsouza *et al.*, "Energy management in multirotor drones : A review," *arXiv preprint arXiv :2501.03102*, 2025.

[11] H. Zhao *et al.*, "Multi-agent reinforcement learning for energy efficiency in uav networks," *Sensors*, vol. 24, n°. 20, 2024.

[12] A. Barros *et al.*, "Optimized matrix multiplication for edge devices : A survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, n°. 1, 2023.

[13] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo et W. Lintner, "United States Data Center Energy Usage Report," Lawrence Berkeley National Laboratory, Berkeley, California, Rapport technique LBNL–2001637, déc. 2024. [En ligne]. Disponible : http://www.osti.gov/servlets/purl/1372902/

[14] International Energy Agency, “Energy and AI,” International Energy Agency, Rapport technique, avr. 2025. [En ligne]. Disponible : https://www.iea.org/reports/energy-and-ai

[15] Fortune Business Insights, “Data Center Market Size, Share & Trends | Growth Report [2032],” Fortune Business Insights, Rapport technique FBI109851, mars 2025. [En ligne]. Disponible : https://www.fortunebusinessinsights.com/data-center-market-109851

[16] Timothy R. Comerford et 2015, “Power Requirements, Energy Costs, and Incentives for Data Centers,” *Biggins Lacy Shapiro & Co.*, nov. 2015. [En ligne]. Disponible : https://blsstrategies.com/insights-press/power-requirements-energy-costs-and-incentives-for-data-centers

[17] V. Strassen, “Gaussian elimination is not optimal,” *Numerische Mathematik*, vol. 13, n°. 4, p. 354–356, août 1969. [En ligne]. Disponible : https://doi.org/10.1007/BF02165411

[18] D. Coppersmith et S. Winograd, “Matrix multiplication via arithmetic progressions,” *Journal of Symbolic Computation*, vol. 9, n°. 3, p. 251–280, mars 1990. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S0747717108800132

[19] A. Kinane, V. Muresan et N. E. O’Connor, “Optimisation of constant matrix multiplication operation hardware using a genetic algorithm,” dans *Applications of Evolutionary Computing*, ser. Lecture Notes in Computer Science, vol. 3907. Springer, 2006, p. 296–307.

[20] A. Hosangadi, F. Fallah et R. Kastner, “Algebraic methods for optimizing constant multiplications in linear systems,” *Journal of Signal Processing Systems*, vol. 49, n°. 1, p. 31–50, 2007.

[21] Y. Voronenko et M. Püschel, “Multiplierless multiple constant multiplication,” *ACM Transactions on Algorithms*, vol. 3, n°. 2, p. 11, mai 2007. [En ligne]. Disponible : https://dl.acm.org/doi/10.1145/1240233.1240234

[22] A. Boutros, A. Arora et V. Betz, “Field-programmable gate array architecture for deep learning : Survey & future directions,” 2024. [En ligne]. Disponible : https://arxiv.org/abs/2404.10076

[23] H. Luo et W. Sun, “Addition is all you need for energy-efficient language models,” 2024. [En ligne]. Disponible : https://arxiv.org/abs/2410.00907

[24] D. S, A. V et A. N. J. Raj, “A review on hardware accelerators for convolutional neural network-based inference engines : Strategies for performance and energy-efficiency enhancement,” *Microprocessors and Microsystems*, vol. 113, p. 105146, mars 2025. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S0141933125000146

[25] A.-A. Zeghaida, D. Daultani, J. P. Langlois et J. P. David, “Scalable Low-Complexity Implementation of Constant Matrix Multiplication Circuits,” dans *2024 IEEE 67th International Midwest Symposium on Circuits and Systems (MWSCAS)*, août 2024, p. 357–361. [En ligne]. Disponible : https://ieeexplore.ieee.org/document/10658880/

[26] D. Bull et D. Horrocks, “Primitive operator digital filters,” *IEE Proceedings G Circuits, Devices and Systems*, 1991.

[27] A. Dempster et M. Macleod, “Use of minimum-adder multiplier blocks in FIR digital filters,” *IEEE Transactions on Circuits and Systems II : Analog and Digital Signal Processing*, 1995.

[28] L. Aksoy, E. O. Güneş et P. Flores, “Search algorithms for the multiple constant multiplications problem : Exact and approximate,” *Microprocessors and Microsystems*, vol. 34, n°. 5, p. 151–162, août 2010. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S0141933109000775

[29] R. Hartley, “Subexpression sharing in filters using canonic signed digit multipliers,” *IEEE Transactions on Circuits and Systems II : Analog and Digital Signal Processing*, 1996.

[30] R. Pasko, P. Schaumont, V. Derudder, S. Vernalde et D. Durackova, “A new algorithm for elimination of common subexpressions,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, n°. 1, p. 58–68, janv. 1999. [En ligne]. Disponible : http://ieeexplore.ieee.org/document/739059/

[31] M. Faust et C.-H. Chang, “Minimal Logic Depth adder tree optimization for Multiple Constant Multiplication,” dans *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, mai 2010, p. 457–460. [En ligne]. Disponible : https://ieeexplore.ieee.org/document/5537658/?arnumber=5537658

[32] M. Kumm, M. Hardieck, J. Willkomm, P. Zipf et U. Meyer-Baese, “Multiple constant multiplication with ternary adders,” *2013 23rd International Conference on Field programmable Logic and Applications*, p. 1–8, sept. 2013. [En ligne]. Disponible : http://ieeexplore.ieee.org/document/6645543/

[33] M. Kumm, O. Gustafsson, M. Garrido et P. Zipf, “Optimal Single Constant Multiplication Using Ternary Adders,” *IEEE Transactions on Circuits and Systems II : Express Briefs*, vol. 65, n°. 7, p. 928–932, juill. 2018. [En ligne]. Disponible : https://ieeexplore.ieee.org/document/7752883/?arnumber=7752883

[34] M. Kumm, M. Hardieck et P. Zipf, “Optimization of Constant Matrix Multiplication with Low Power and High Throughput,” *IEEE Transactions on Computers*, 2017.

[35] M. Kumm, P. Zipf, M. Faust et C.-H. Chang, "Pipelined adder graph optimization for high speed multiple constant multiplication," *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2012.

[36] A. Kinane, V. Muresan et N. O'Connor, "Towards an optimised VLSI design algorithm for the constant matrix multiplication problem," dans *2006 IEEE International Symposium on Circuits and Systems (ISCAS)*, mai 2006, p. 4 pp.–. [En ligne]. Disponible : https://ieeexplore.ieee.org/document/1693782

[37] A. Kinane, V. Muresan et N. O'Connor, "Optimisation of constant matrix multiplication operation hardware using a genetic algorithm," dans *Proceedings of the International Conference on Applications of Evolutionary Computing*. Berlin, Heidelberg : Springer-Verlag, 2006.

[38] L. Aksoy, P. Flores et J. Monteiro, "A novel method for the approximation of multiplier-less constant matrix vector multiplication," dans *Proceedings of the IEEE International Conference on Embedded and Ubiquitous Computing*, 2015.

[39] L. Aksoy, E. O. Gunes et P. Flores, "An Exact Breadth-First Search Algorithm for the Multiple Constant Multiplications Problem," dans *2008 NORCHIP*. Tallin, Estonia : IEEE, nov. 2008, p. 41–46. [En ligne]. Disponible : http://ieeexplore.ieee.org/document/4738280/

[40] A. Lehnert, P. Holzinger, S. Pfenning, R. Müller et M. Reichenbach, "Most Resource Efficient Matrix Vector Multiplication on FPGAs," *IEEE Access*, vol. 11, p. 3881–3898, 2023.

[41] Y. Li, X. Dong et W. Wang, "Additive Powers-of-Two Quantization : An Efficient Non-uniform Discretization for Neural Networks," févr. 2020. [En ligne]. Disponible : http://arxiv.org/abs/1909.13144

[42] Xilinx Inc., *7 Series FPGAs Configurable Logic Block User Guide*, sept. 2016, version 1.8. [En ligne]. Disponible : https://docs.xilinx.com/r/en-US/ug474_7Series_CLB

[43] A. Arora, A. Boutros, D. Rauch, A. Rajen, A. Borda, S. A. Damghani, S. Mehta, S. Kate, P. Patel, K. B. Kent, V. Betz et L. K. John, "Koios : A deep learning benchmark suite for fpga architecture and cad research," *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, p. 355–362, 2021. [En ligne]. Disponible : https://api.semanticscholar.org/CorpusID:235422278

[44] Xilinx Inc., *7 Series DSP48E1 Slice User Guide*, mars 2018, version 1.10. [En ligne]. Disponible : https://docs.xilinx.com/r/en-US/ug479_7Series_DSP48E1

[45] B. Parhami, *Computer Arithmetic : Algorithms and Hardware Designs*, 2^{e} éd. New York : Oxford University Press, 2010.

[46] H. Wu, P. Judd, X. Zhang, M. Isaev et P. Micikevicius, “Integer quantization for deep learning inference : Principles and empirical evaluation,” *ArXiv*, vol. abs/2004.09602, 2020.

[47] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney et K. Keutzer, “A survey of quantization methods for efficient neural network inference,” 2021. [En ligne]. Disponible : https://arxiv.org/abs/2103.13630

[48] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy et D. S. Modha, “Learned Step Size Quantization,” mai 2020. [En ligne]. Disponible : http://arxiv.org/abs/1902.08153

ANNEXE A MODULE COMBINATOIRE SYSTEMVERILOG GÉNÉRÉ

```
module combinatorial_3x3_4_20#(
    parameter ROWS = 3,
    parameter COLS = 3,
    parameter MEM_SIZE = 20,
    parameter input_bit_width = 4,
    parameter output_bit_width = 9
)(
    input wire [input_bit_width-1:0] input_vector [0: COLS-1],
    output wire [output_bit_width-1:0] output_vector [0: ROWS-1]
);

    wire [output_bit_width-1:0] MEM [0:MEM_SIZE];

    assign MEM[0] = input_vector[0] << 3;
    assign MEM[1] = input_vector[0] << 2;
    assign MEM[2] = input_vector[0] << 1;
    assign MEM[3] = input_vector[0] << 0;
    assign MEM[4] = input_vector[1] << 3;
    assign MEM[5] = input_vector[1] << 2;
    assign MEM[6] = input_vector[1] << 1;
    assign MEM[7] = input_vector[1] << 0;
    assign MEM[8] = input_vector[2] << 3;
    assign MEM[9] = input_vector[2] << 2;
    assign MEM[10] = input_vector[2] << 1;
    assign MEM[11] = input_vector[2] << 0;
    assign MEM[12] = MEM[3] + MEM[11];
    assign MEM[13] = MEM[5] + MEM[8];
    assign MEM[14] = MEM[6] + MEM[7];
    assign MEM[15] = MEM[12] + MEM[13];
    assign MEM[16] = MEM[2] + MEM[10];
    assign MEM[17] = MEM[14] + MEM[15];
    assign MEM[18] = MEM[1] + MEM[15];
    assign MEM[19] = MEM[12] + MEM[14];
    assign MEM[20] = MEM[16] + MEM[17];
    assign output_vector[0] = MEM[18];
    assign output_vector[1] = MEM[19];
    assign output_vector[2] = MEM[20];

endmodule
```

ANNEXE B FONCTION CYCLECOUNT() POUR LE CALCUL DES CYCLES

```python
def cycleCount(operationsArray):
    result = []
    max_cycle = 0
    for entry in operationsArray[1:]:
        element = entry.copy()
        op_type = entry[3]
        first_cycle = 0
        last_cycle = 0

        if op_type in ('<<', '-<<'):
            first_cycle = 0
            last_cycle = -1

        elif op_type == '+':
            left_op = entry[0]
            right_op = entry[1]
            first_cycle = max(result[left_op][4], result[right_op][4]) + 1
            last_cycle = 0
            max_cycle = max(max_cycle, first_cycle)
            result[left_op][5] = max(first_cycle - 1, result[left_op][5])
            result[right_op][5] = max(first_cycle - 1, result[right_op
                ↪ ][5])

        elif op_type == '=':
            source_op = entry[1]
            first_cycle = max_cycle
            last_cycle = max_cycle
            result[source_op][5] = max(max_cycle, result[source_op][5])

        else:
            print(f"{op_type} is not a valid operation")
        element.append(first_cycle)
        element.append(last_cycle)
        result.append(element)
# Add header row. It contains the parameters (weights, bitwdith, etc...)
    ↪ required by the funciton that writes the verilog code
    result.insert(0, operationsArray[0] + ['FirstCycle', 'LastCycle'])
    return result
```

ANNEXE C MODULE SYSTEMVERILOG PIPELINÉ

```
module pipelined_3x3_4_11#(
    parameter ROWS = 3,
    parameter COLS = 3,
    parameter MEM_SIZE = 11,
    parameter input_bit_width = 4,
    parameter output_bit_width = 9
)(
    input wire clk,
    input wire reset,
    input wire [input_bit_width-1:0] input_vector [0: COLS-1],
    output wire [output_bit_width-1:0] output_vector [0: ROWS-1]
);

    // Pipeline registers declaration
    reg [output_bit_width-1:0] stage_0_0;
    reg [output_bit_width-1:0] stage_0_1;
    reg [output_bit_width-1:0] stage_0_10;
    reg [output_bit_width-1:0] stage_0_11;
    reg [output_bit_width-1:0] stage_0_2;
    reg [output_bit_width-1:0] stage_0_3;
    reg [output_bit_width-1:0] stage_0_4;
    reg [output_bit_width-1:0] stage_0_5;
    reg [output_bit_width-1:0] stage_0_6;
    reg [output_bit_width-1:0] stage_0_7;
    reg [output_bit_width-1:0] stage_0_8;
    reg [output_bit_width-1:0] stage_0_9;
    reg [output_bit_width-1:0] stage_1_1;
    reg [output_bit_width-1:0] stage_1_12;
    reg [output_bit_width-1:0] stage_1_13;
    reg [output_bit_width-1:0] stage_1_14;
    reg [output_bit_width-1:0] stage_1_16;
    reg [output_bit_width-1:0] stage_2_1;
    reg [output_bit_width-1:0] stage_2_14;
    reg [output_bit_width-1:0] stage_2_15;
    reg [output_bit_width-1:0] stage_2_16;
    reg [output_bit_width-1:0] stage_2_19;
    reg [output_bit_width-1:0] stage_3_16;
    reg [output_bit_width-1:0] stage_3_17;
    reg [output_bit_width-1:0] stage_3_18;
```

```
40      reg [output_bit_width-1:0] stage_3_19;
41      reg [output_bit_width-1:0] stage_4_18;
42      reg [output_bit_width-1:0] stage_4_19;
43      reg [output_bit_width-1:0] stage_4_20;
44
45      wire [output_bit_width-1:0] MEM [0:11];
46
47      // MEM initialization
48          assign MEM[0] = input_vector[0] << 3;
49          assign MEM[1] = input_vector[0] << 2;
50          assign MEM[2] = input_vector[0] << 1;
51          assign MEM[3] = input_vector[0] << 0;
52          assign MEM[4] = input_vector[1] << 3;
53          assign MEM[5] = input_vector[1] << 2;
54          assign MEM[6] = input_vector[1] << 1;
55          assign MEM[7] = input_vector[1] << 0;
56          assign MEM[8] = input_vector[2] << 3;
57          assign MEM[9] = input_vector[2] << 2;
58          assign MEM[10] = input_vector[2] << 1;
59          assign MEM[11] = input_vector[2] << 0;
60
61      always @(posedge clk) begin
62          if (reset) begin
63              // Reset the pipeline registers
64              stage_0_0 <= 0;
65              stage_0_1 <= 0;
66              stage_0_10 <= 0;
67              stage_0_11 <= 0;
68              stage_0_2 <= 0;
69              stage_0_3 <= 0;
70              stage_0_4 <= 0;
71              stage_0_5 <= 0;
72              stage_0_6 <= 0;
73              stage_0_7 <= 0;
74              stage_0_8 <= 0;
75              stage_0_9 <= 0;
76              stage_1_1 <= 0;
77              stage_1_12 <= 0;
78              stage_1_13 <= 0;
79              stage_1_14 <= 0;
80              stage_1_16 <= 0;
81              stage_2_1 <= 0;
82              stage_2_14 <= 0;
```

```
83              stage_2_15 <= 0;
84              stage_2_16 <= 0;
85              stage_2_19 <= 0;
86              stage_3_16 <= 0;
87              stage_3_17 <= 0;
88              stage_3_18 <= 0;
89              stage_3_19 <= 0;
90              stage_4_18 <= 0;
91              stage_4_19 <= 0;
92              stage_4_20 <= 0;
93          end else begin
94              // Stage 0
95              stage_0_0 <= MEM[0];
96              stage_0_1 <= MEM[1];
97              stage_0_2 <= MEM[2];
98              stage_0_3 <= MEM[3];
99              stage_0_4 <= MEM[4];
100             stage_0_5 <= MEM[5];
101             stage_0_6 <= MEM[6];
102             stage_0_7 <= MEM[7];
103             stage_0_8 <= MEM[8];
104             stage_0_9 <= MEM[9];
105             stage_0_10 <= MEM[10];
106             stage_0_11 <= MEM[11];
107
108             // Stage 1
109             stage_1_12 <= stage_0_11 + stage_0_3;
110             stage_1_13 <= stage_0_8 + stage_0_5;
111             stage_1_14 <= stage_0_7 + stage_0_6;
112             stage_1_16 <= stage_0_10 + stage_0_2;
113             // Reassigning old values
114             stage_1_1 <= stage_0_1;
115
116             // Stage 2
117             stage_2_15 <= stage_1_13 + stage_1_12;
118             stage_2_19 <= stage_1_14 + stage_1_12;
119             // Reassigning old values
120             stage_2_1 <= stage_1_1;
121             stage_2_14 <= stage_1_14;
122             stage_2_16 <= stage_1_16;
123
124             // Stage 3
125             stage_3_17 <= stage_2_15 + stage_2_14;
```

```
126             stage_3_18 <= stage_2_15 + stage_2_1;
127             // Reassigning old values
128             stage_3_16 <= stage_2_16;
129             stage_3_19 <= stage_2_19;
130 
131             // Stage 4
132             stage_4_20 <= stage_3_17 + stage_3_16;
133             // Reassigning old values
134             stage_4_18 <= stage_3_18;
135             stage_4_19 <= stage_3_19;
136         end
137     end
138 
139     assign output_vector[0] = stage_4_18;
140     assign output_vector[1] = stage_4_19;
141     assign output_vector[2] = stage_4_20;
142 
143 endmodule
```

ANNEXE D MODULE SYSTEMVERILOG PIPELINÉ AGRÉGÉ

```
module pipelined_aggregated_3x3_4_11#(
    parameter ROWS = 3,
    parameter COLS = 3,
    parameter MEM_SIZE = 11,
    parameter input_bit_width = 4,
    parameter output_bit_width = 9
)(
   input wire clk,
   input wire reset,
   input wire [input_bit_width-1:0] input_vector [0: COLS-1],
   output wire [output_bit_width-1:0] output_vector [0: ROWS-1]
);
    // Pipeline registers declaration
    reg [output_bit_width-1:0] stage_0_0;
    reg [output_bit_width-1:0] stage_0_1;
    reg [output_bit_width-1:0] stage_0_10;
    reg [output_bit_width-1:0] stage_0_11;
    reg [output_bit_width-1:0] stage_0_2;
    reg [output_bit_width-1:0] stage_0_3;
    reg [output_bit_width-1:0] stage_0_4;
    reg [output_bit_width-1:0] stage_0_5;
    reg [output_bit_width-1:0] stage_0_6;
    reg [output_bit_width-1:0] stage_0_7;
    reg [output_bit_width-1:0] stage_0_8;
    reg [output_bit_width-1:0] stage_0_9;
    reg [output_bit_width-1:0] stage_1_1;
    reg [output_bit_width-1:0] stage_1_14;
    reg [output_bit_width-1:0] stage_1_15;
    reg [output_bit_width-1:0] stage_1_16;
    reg [output_bit_width-1:0] stage_1_19;
    reg [output_bit_width-1:0] stage_2_18;
    reg [output_bit_width-1:0] stage_2_19;
    reg [output_bit_width-1:0] stage_2_20;

    wire [output_bit_width-1:0] MEM [0:11];

    // MEM initialization
    assign MEM[0] = input_vector[0] << 3;
    assign MEM[1] = input_vector[0] << 2;
```

```
40      assign MEM[2] = input_vector[0] << 1;
41      assign MEM[3] = input_vector[0] << 0;
42      assign MEM[4] = input_vector[1] << 3;
43      assign MEM[5] = input_vector[1] << 2;
44      assign MEM[6] = input_vector[1] << 1;
45      assign MEM[7] = input_vector[1] << 0;
46      assign MEM[8] = input_vector[2] << 3;
47      assign MEM[9] = input_vector[2] << 2;
48      assign MEM[10] = input_vector[2] << 1;
49      assign MEM[11] = input_vector[2] << 0;
50
51      always @(posedge clk) begin
52          if (reset) begin
53              // Reset the pipeline registers
54              stage_0_0 <= 0;
55              stage_0_1 <= 0;
56              stage_0_10 <= 0;
57              stage_0_11 <= 0;
58              stage_0_2 <= 0;
59              stage_0_3 <= 0;
60              stage_0_4 <= 0;
61              stage_0_5 <= 0;
62              stage_0_6 <= 0;
63              stage_0_7 <= 0;
64              stage_0_8 <= 0;
65              stage_0_9 <= 0;
66              stage_1_1 <= 0;
67              stage_1_14 <= 0;
68              stage_1_15 <= 0;
69              stage_1_16 <= 0;
70              stage_1_19 <= 0;
71              stage_2_18 <= 0;
72              stage_2_19 <= 0;
73              stage_2_20 <= 0;
74
75          end else begin
76              // Stage 0
77              stage_0_0 <= MEM[0];
78              stage_0_1 <= MEM[1];
79              stage_0_2 <= MEM[2];
80              stage_0_3 <= MEM[3];
81              stage_0_4 <= MEM[4];
82              stage_0_5 <= MEM[5];
```

```
83              stage_0_6 <= MEM[6];
84              stage_0_7 <= MEM[7];
85              stage_0_8 <= MEM[8];
86              stage_0_9 <= MEM[9];
87              stage_0_10 <= MEM[10];
88              stage_0_11 <= MEM[11];
89
90              // Stage 1
91              stage_1_15 <= (stage_0_8 + stage_0_5) + (stage_0_11 +
                   ↪ stage_0_3);
92              stage_1_19 <= (stage_0_7 + stage_0_6) + (stage_0_11 +
                   ↪ stage_0_3);
93              // Reassigning old values
94              stage_1_1 <= (stage_0_1);
95              stage_1_14 <= (stage_0_7 + stage_0_6);
96              stage_1_16 <= (stage_0_10 + stage_0_2);
97
98              // Stage 2
99              stage_2_20 <= (stage_1_15 + stage_1_14) + (stage_1_16);
100             // Reassigning old values
101             stage_2_18 <= (stage_1_15 + stage_1_1);
102             stage_2_19 <= (stage_1_19);
103         end
104     end
105     assign output_vector[0] = stage_4_18;
106     assign output_vector[1] = stage_4_19;
107     assign output_vector[2] = stage_4_20;
108
109 endmodule
```

ANNEXE E ORDONNANCEMENT DES CALCULS SUR UN ENSEMBLE DE NŒUDS REPRÉSENTANT LES OPÉRATIONS DU GRAPHE DE DÉPENDANCES POUR L'ARCHITECTURE PIPELINÉ

Chaque ligne représente un nœud. Les nœuds du cycle 0 correspondent aux feuilles du graphe. Les opérations `<=` sur une même opérande (comme à la ligne 12) représentent une réaffectation servant à maintenir en vie le signal, en le transférant du registre de l'étage précédent vers le registre de cycle de l'opérande de gauche.

```
1  cycle 0:
2      1 <= 1
3      2 <= 2
4      3 <= 3
5      5 <= 5
6      6 <= 6
7      7 <= 7
8      8 <= 8
9      10 <= 10
10     11 <= 11
11 cycle 1:
12     1 <= 1
13     12 <= 11 + 3
14     13 <= 8 + 5
15     14 <= 7 + 6
16     16 <= 10 + 2
17 cycle 2:
18     1 <= 1
19     14 <= 14
20     15 <= 13 + 12
21     16 <= 16
22     19 <= 14 + 12
23 cycle 3:
24     16 <= 16
25     17 <= 15 + 14
26     18 <= 15 + 1
27     19 <= 19
28 cycle 4:
29     18 <= 18
30     19 <= 19
31     20 <= 17 + 16
```

ANNEXE F ORDONNANCEMENT DES CALCULS POUR L'ARCHITECTURE PIPELINÉ AGRÉGÉ

```
cycle 0:
   1 <= 1
   2 <= 2
   3 <= 3
   5 <= 5
   6 <= 6
   7 <= 7
   8 <= 8
   10 <= 10
   11 <= 11

cycle 1:
   1 <= 1
   14 <= 7 + 6
   15 <= (8 + 5) + (11 + 3)
   16 <= 10 + 2
   19 <= (7 + 6) + (11 + 3)

cycle 2:
   18 <= 15 + 1
   19 <= 19
   20 <= (15 + 14) + 16
```

ANNEXE G MODULE MATMULVEC

```
module MATMULVEC_3x3_4#(
    parameter ROWS = 3,
    parameter COLS = 3,
    parameter input_bit_width = 4,
    parameter output_bit_width = 9
)(
    input wire [input_bit_width-1:0] input_vector [0: COLS-1],
    output wire [output_bit_width-1:0] output_vector [0: ROWS-1]
);

    // Matrix A
    wire [3:0] matrix [0:ROWS-1][0:COLS-1];
    assign matrix[0][0] = 5;
    assign matrix[0][1] = 4;
    assign matrix[0][2] = 9;
    assign matrix[1][0] = 1;
    assign matrix[1][1] = 3;
    assign matrix[1][2] = 1;
    assign matrix[2][0] = 3;
    assign matrix[2][1] = 7;
    assign matrix[2][2] = 11;

    // Intermediate mults
    wire [8:0] P_0_0 = matrix[0][0] * input_vector[0];
    wire [8:0] P_0_1 = matrix[0][1] * input_vector[1];
    wire [8:0] P_0_2 = matrix[0][2] * input_vector[2];
    wire [8:0] P_1_0 = matrix[1][0] * input_vector[0];
    wire [8:0] P_1_1 = matrix[1][1] * input_vector[1];
    wire [8:0] P_1_2 = matrix[1][2] * input_vector[2];
    wire [8:0] P_2_0 = matrix[2][0] * input_vector[0];
    wire [8:0] P_2_1 = matrix[2][1] * input_vector[1];
    wire [8:0] P_2_2 = matrix[2][2] * input_vector[2];

    assign output_vector[0] = P_0_0 + P_0_1 + P_0_2;
    assign output_vector[1] = P_1_0 + P_1_1 + P_1_2;
    assign output_vector[2] = P_2_0 + P_2_1 + P_2_2;

endmodule
```

ANNEXE H MODULE SHIFT&ADD

```
module shift_add_3x3_4#(
    parameter ROWS = 3,
    parameter COLS = 3,
    parameter input_bit_width = 4,
    parameter output_bit_width = 9
)(
    input wire [input_bit_width-1:0] input_vector [0: COLS-1],
    output wire [output_bit_width-1:0] output_vector [0: ROWS-1]
);

    // Matrix A
    wire [3:0] matrix [0:ROWS-1][0:COLS-1];

    assign matrix[0][0] = 5;
    assign matrix[0][1] = 4;
    assign matrix[0][2] = 9;
    assign matrix[1][0] = 1;
    assign matrix[1][1] = 3;
    assign matrix[1][2] = 1;
    assign matrix[2][0] = 3;
    assign matrix[2][1] = 7;
    assign matrix[2][2] = 11;

    // intermediate variables
    wire [8:0] P_0_0;
    wire [8:0] P_0_1;
    wire [8:0] P_0_2;
    wire [8:0] P_1_0;
    wire [8:0] P_1_1;
    wire [8:0] P_1_2;
    wire [8:0] P_2_0;
    wire [8:0] P_2_1;
    wire [8:0] P_2_2;

    // Shift_add modules
    shift_add#(4,4,9) mult_0_0 (.input(input_vector[0]), .coefficient(
        ↪ matrix[0][0]), .P(P_0_0));
    shift_add#(4,4,9) mult_0_1 (.input(input_vector[1]), .coefficient(
        ↪ matrix[0][1]), .P(P_0_1));
```

```
38      shift_add#(4,4,9) mult_0_2 (.input(input_vector[2]), .coefficient(
            ↪ matrix[0][2]), .P(P_0_2));
39      assign output_vector[0] = P_0_0 + P_0_1 + P_0_2;
40      shift_add#(4,4,9) mult_1_0 (.input(input_vector[0]), .coefficient(
            ↪ matrix[1][0]), .P(P_1_0));
41      shift_add#(4,4,9) mult_1_1 (.input(input_vector[1]), .coefficient(
            ↪ matrix[1][1]), .P(P_1_1));
42      shift_add#(4,4,9) mult_1_2 (.input(input_vector[2]), .coefficient(
            ↪ matrix[1][2]), .P(P_1_2));
43      assign output_vector[1] = P_1_0 + P_1_1 + P_1_2;
44      shift_add#(4,4,9) mult_2_0 (.input(input_vector[0]), .coefficient(
            ↪ matrix[2][0]), .P(P_2_0));
45      shift_add#(4,4,9) mult_2_1 (.input(input_vector[1]), .coefficient(
            ↪ matrix[2][1]), .P(P_2_1));
46      shift_add#(4,4,9) mult_2_2 (.input(input_vector[2]), .coefficient(
            ↪ matrix[2][2]), .P(P_2_2));
47      assign output_vector[2] = P_2_0 + P_2_1 + P_2_2;
48
49  endmodule
```