Étude d'architectures dédiées aux systèmes embarqués intelligents et efficaces en énergie

Guillaume Devic

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE L'UNIVERSITÉ DE MONTPELLIER

En Systèmes Automatiques et Microélectroniques (SyAM)

École doctorale Information, Structure, Systèmes (I2S)

Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM)

Étude d'architectures dédiées aux systèmes embarqués intelligents et efficaces en énergie

Présentée par Guillaume DEVIC

Le 03 décembre 2021

Sous la direction de Abdoulaye GAMATIÉ, directeur de thèse et Gilles SASSATELLI, co-directeur de thèse.

Devant le jury composé de

Abdoulaye GAMATIÉ, Directeur de Recherche, CNRS, Université de Montpellier	Directeur de thèse
Gilles SASSATELLI, Directeur de Recherche, CNRS, Université de Montpellier	Co-directeur de thèse
Cécile BELLEUDY, Maitre de conférences – HDR, Université Côte d'Azur	Rapporteur
Erven ROHOU, Directeur de Recherche, Inria Rennes – Bretagne Atlantique	Rapporteur
Pierre BOULET, Professeur des Universités, Université de Lille	Président du jury

UNIVERSITÉ DE MONTPELLIER

Remerciements

Ces trois années de thèses et plus particulièrement ce manuscrit closent le chapitre de ma vie consacré aux études universitaires. Avant de rentrer dans le vif du sujet, je souhaiterais adresser mes remerciements aux personnes qui ont contribué, de près ou de loin, à l'achèvement de ces années de thèse.

Tout d'abord, je tiens à remercier les deux rapporteurs de ce manuscrit Cécile BELLEUDY et Erven ROHOU. Les remarques et commentaires enrichissants contenus dans leurs rapports respectifs m'ont été bénéfiques. Je tiens également à remercier Pierre BOULET d'avoir consacré du temps à la lecture de ce manuscrit et d'avoir présidé le jury de thèse.

Je remercie mes deux directeurs de thèses Abdoulaye GAMATIÉ et Gilles SASSATELLI d'avoir été présents tout le long de ces trois années de thèses qui se sont révélées être à la fois surprenantes et éprouvantes.

Je tiens à remercier Maxime FRANCE-PILLOIS de son temps et de ses conseils qu'il m'a consacrés. Sincèrement, merci à Jérémie SALLES, Thierry GIL et Laurent DEKNYFF pour leurs disponibilités et de l'aide qu'ils ont pu me procurer.

Je remercie spécialement l'équipe ADAC qui m'a accueillie où il y règne une ambiance et une entraide rarement égalées. Et plus largement, je remercie l'ensemble des personnes qui travaillent au LIRMM. La vie de doctorant est jonchée de rencontres qui sont à la fois des sources de motivations et d'inspirations. Je souhaite remercier Bastien, Frédéric, Guillaume, Julie, Marcos, Mathilde et Pierre.

Je tiens à chaleureusement remercier Francesco DI GREGORIO et Maxime MIRKA avec qui j'ai eu le plaisir de partager ces trois années de thèse au sein du même bureau.
Merci pour tout.

Mes derniers remerciements sont évidemment destinés à ma famille, je pense en particulier à mes parents et ma soeur. Leurs soutiens sans faille ont largement contribué à l'achèvement de mon cursus universitaire et plus particulièrement de cette thèse.

En bref, merci à tous.

Résumé

L'informatique en périphérie ou *edge computing* est un paradigme de calcul distribué récent permettant d'adresser la problématique des données massives, notamment dans le contexte des objets connectés. Ces derniers prennent une place toujours plus prépondérante dans nos vies. Les exemples vont de la montre connectée à la maison intelligente, en passant par les voitures connectées. Pour des raisons de réactivité due à la surcharge du réseau et d'efficacité énergétique, les traitements des données ainsi générées par ces objets sont passés progressivement d'infrastructures centralisées dans le cloud à des systèmes distribués intégrant des serveurs puissants et des systèmes embarqués utilisables au plus près des sources de données. Aujourd'hui, le traitement de ces dernières intègre de plus en plus d'algorithmes d'intelligence artificielle (typiquement, pour l'analyse de données et la prise de décision) dans le *edge computing*. Pour rendre cela viable sur les supports embarqués, il est important d'étudier de nouvelles architectures suffisamment performantes et peu gourmandes en énergie.

Cette thèse aborde la problématique du calcul embarqué dédié au *edge computing*. En particulier, elle se focalise sur la conception d'architectures à faible consommation permettant de traiter des algorithmes d'apprentissage machine. Dans un premier temps, elle explore une piste basée sur une architecture multicœur hétérogène afin de voir dans quelle mesure cela permet de répondre à une large demande algorithmique. Cette architecture innovante repose sur la technologie de processeur proposée par la société française Cortus S.A. Ensuite, la thèse se concentre sur l'accélération des réseaux profonds en proposant une nouvelle unité MAC (*multiply-accumulate*) à la fois flexible et efficace en énergie. Les gains fournis par cette unité MAC sont évalués à travers une modélisation de haut niveau dans des architectures d'accélérateurs de réseau de neurones convolutif. Plus généralement, le travail présenté dans cette thèse offre des enseignements intéressants quant au choix entre des architectures multi-cœurs généralistes et des architectures dédiées de type accélérateur d'intelligence artificielle, pour des nœuds de calcul efficaces en énergie pour le *edge computing*.

Abstract

Edge computing is a recent paradigm of distributed computing that addresses the problem of massive data, particularly in the context of connected objects. These objects are becoming increasingly important in our lives. Examples include connected watches, smart homes, and connected cars. For reasons of reactivity due to network overload and energy efficiency, the processing of data generated by these objects has progressively moved from centralized infrastructures in the cloud to distributed systems integrating powerful servers and embedded systems that can be used as close as possible to the data sources. Today, the data processing is gradually integrating more and more artificial intelligence algorithms (typically for data analysis and decision making) in edge computing. To make this viable on embedded systems, it is important to study new architectures with sufficient performance and low power consumption.

This thesis addresses the issue of embedded computing dedicated to edge computing. In particular, it focuses on the design of low-power architectures for processing machine learning algorithms. First, it explores a solution based on a heterogeneous multicore architecture in order to see to what extent it can meet a large algorithmic demand. This innovative architecture is based on the processor technology proposed by the French company Cortus S.A. Then, the thesis focuses on the acceleration of deep networks by proposing a new MAC (multiply-accumulate) unit that is both flexible and energy efficient. The gains provided by this MAC unit are evaluated through high-level modeling in convolutional neural network accelerators architectures. More generally, the work presented in this thesis offers interesting insights into the choice between general-purpose multicore architectures and dedicated artificial intelligence accelerators architectures for energy-efficient edge computing nodes.

Table des matières

Liste des figures

Liste des tableaux

Liste des abréviations

BC Basse Consommation

BCF Basse Consommation Flottant

CNN *Convolutional Neural Network* - Réseau Neuronal Convolutif

DDP Différence De Potentiel

DSP *Digital Signal Processing*

FF *Flip-Flop*

FPGA *Field Programmable Gate Array*

FPU *Floating Point Unit*

GPU *Graphics Processing Unit*

HP Haute Performance

IA Intelligence Artificielle

IP *Intellectual Property*

ISA *Instruction Set Architecture*

LUT *LookUp Table*

MAC Multiplication-ACcumulation

ML *Machine Learning*

PE *Processing Elements*

SIMD *Single Instruction Multiple Data*

XBAR *Crossbars*

Chapitre 1

Introduction

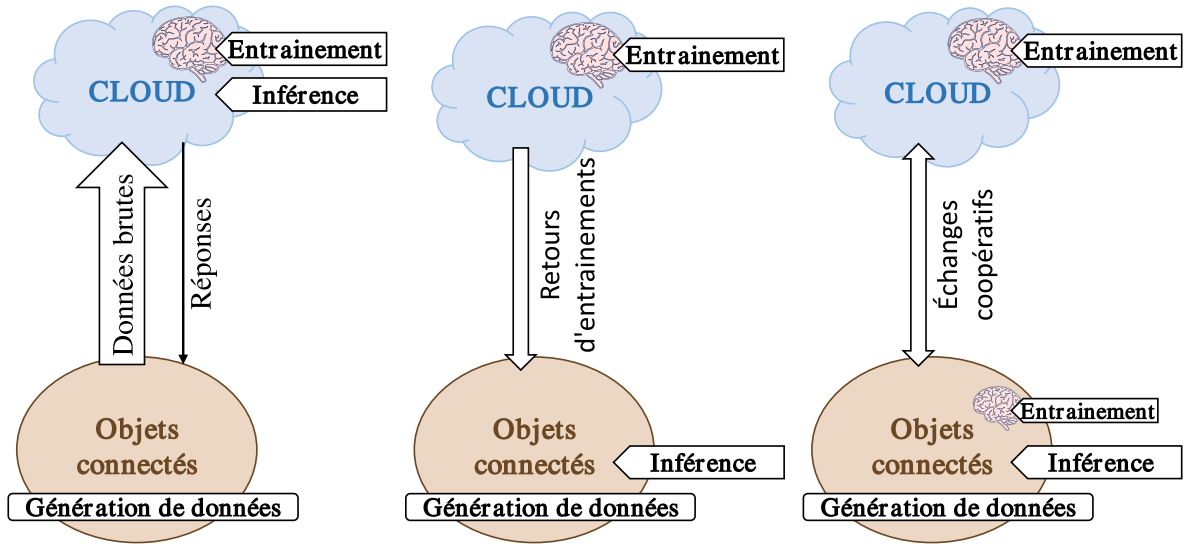
Table des matières du chapitre

1.1 L'intelligence artificielle (IA) embarquée

Notre société est de plus en plus tournée vers l'utilisation des objets connectés. Ces dernières années les entreprises et les start-up ne tarissent pas d'imagination pour proposer de nouveaux types d'applications liées aux objets connectés. L'internet des objets représente de nos jours environ 50 milliards d'objets et devrait atteindre les 250 milliards d'ici 2030 d'après O. Bonnaud [8]. Aujourd'hui, le traitement des données issu des objets connectés est largement centralisé dans l'informatique dans le nuage (adaptation du terme anglais *Cloud*). En conservant ce paradigme centralisé et avec l'évolution exponentielle du nombre d'objets connectés, de nombreuses problématiques inhérentes au fonctionnement de l'internet des objets deviendront prédominantes.

Comme l'illustre la Figure 1.1a, les objets connectés (essentiellement des capteurs) génèrent des données qui sont en intégralité transmises vers le *cloud*. Une fois le traitement des données

est achevé par ce dernier, le résultat est renvoyé vers les objets connectés. L'envoi massif de données vers le *cloud* entraîne des saturations de la bande passante du réseau.



(a) Paradigme centralisé sur le *cloud*.

(b) Avènement de l'*edge computing*.

(c) Vers le *smart edge computing*.

Figure 1.1 – Évolution vers des systèmes embarqués intelligents.

L'une des solutions est l'émergence du paradigme de l'informatique en périphérie (adaptation du terme anglais *edge computing* - voir Figure 1.1b) qui a pour objectif de rapprocher les capacités de calcul au plus près des sources des données. L'informatique en périphérie présente un certain nombre d'avantages, parmi lesquels [9–11]:

- la diminution de la bande passante de communication et donc des coûts énergétiques grâce à la réduction des transferts de données vers les serveurs en nuage centralisés ;
- la capacité de traitement des données en temps réel facilitée par la proximité physique entre les données et les dispositifs de calcul ;
- la confidentialité des données sensibles grâce à un traitement *in situ* sur les dispositifs en périphérie, évitant ainsi leur déchargement vers un traitement à distance ;
- et la fiabilité et l'évolutivité grâce à la nature distribuée des nœuds de périphérie, où la défaillance de certains nœuds peut être facilement contournée avec un impact minimal sur le système global et où l'ajout de nouveaux dispositifs au système pour plus de puissance de calcul est possible de manière modulaire.

Cependant, l'une des principales difficultés dans l'utilisation de l'informatique en périphérie est la mise en œuvre des outils logiciels d'analyse des données. De nombreuses applications nécessitant une analyse de données exploitent en particulier l'intelligence artificielle (IA). Ces applications concernent l'analyse vidéo en temps réel, l'extraction de caractéristiques à partir d'images, l'identification de caractères manuscrits, et la surveillance liée à la santé des personnes jusqu'à la maison intelligente. Le fonctionnement de nombreux algorithmes d'IA est basé sur deux phases de traitement distinctes : l'entraînement et l'inférence. L'entraînement fonctionne sur un large ensemble de données pour en tirer des enseignements et l'inférence réutilise ces enseignements pour analyser les prochaines données reçues. Les systèmes embarqués devront être adaptés pour exécuter des algorithmes d'IA souvent utiles au niveau des noeuds terminaux de l'informatique en périphérie. Comme le montre la Figure 1.1c, actuellement des objets connectés commencent à avoir les ressources matérielles suffisantes pour effectuer l'inférence des données. Mais, l'entraînement reste réservé au cloud. Pour pleinement profiter de la décentralisation du traitement des données, les systèmes embarqués doivent parvenir à exécuter une partie ou la totalité de l'entraînement des algorithmes d'IA.

1.2 Problématique de l'efficacité énergétique

La conception des systèmes embarqués vise des implantations consommant peu d'énergie, avec des performances de calcul adaptées aux contraintes des applications et de l'environnement dans lequel les systèmes sont déployés. Elle est devenue davantage problématique en visant l'exécution d'algorithmes de *Machine Learning* (ML). Les efforts de recherche consacrés à cet effet sont considérables au cours des dernières années. Il s'agit de proposer des architectures innovantes et des optimisations garantissant la meilleure efficacité énergétique. Des exemples emblématiques sont les accélérateurs matériels pour exécuter des réseaux neuronaux convolutifs. Cependant, la conception de tels accélérateurs dédiés pose la question de l'évolutivité des systèmes dans le temps, ainsi que de leur applicabilité au-delà du contexte du *machine learning*.

La figure 1.2 donne un aperçu général de divers supports de calcul modernes [1]. Les auteurs distinguent les accélérateurs faible consommation des architectures de processeurs retrouvés dans les serveurs de calcul plus puissants. Les premiers sont adaptés pour l'inférence en IA alors que les seconds permettent en plus l'apprentissage des réseaux de neurones, car ayant plus de mémoire et de ressources de calcul.

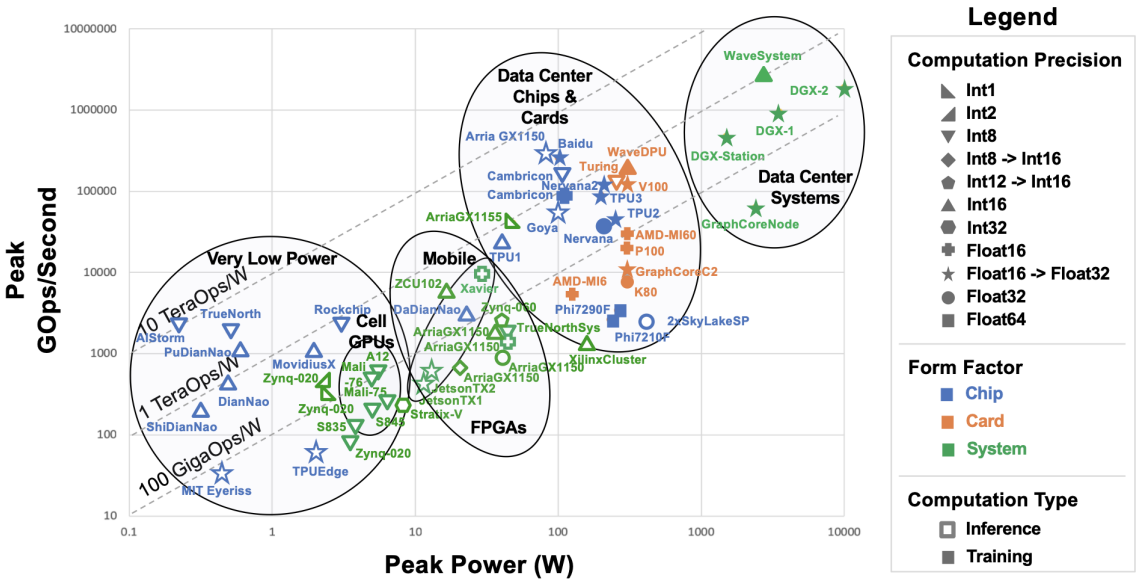


Figure 1.2 – Paysage des accélérateurs et processeurs dédiés au ML [1].

Dans cette thèse, les systèmes visés se placent surtout sur la partie gauche de l'axe horizontal dans la figure 1.2, et plus précisément en dessous des 2 W. Cette tranche de consommation est seulement occupée par des accélérateurs.

1.3 Objectifs de la thèse

Notre objectif est d'étudier des choix architecturaux pertinents pour l'exécution de divers types d'algorithmes, y compris ceux de *machine learning*, tout en garantissant un équilibre entre les capacités de calcul et une faible consommation d'énergie. Il s'inscrit dans le contexte de l'informatique en périphérie où divers traitements sont appliqués aux nombreuses données collectées auprès des capteurs. Ces traitements comprennent par exemple la compression de données, l'analyse de données, la prise de décisions. Ces algorithmes, en particulier ceux du *machine learning*, ont la spécificité de traiter de grandes masses de données nécessitant parfois une charge de calcul importante. Cela pose la problématique de la mise en adéquation des besoins en performance de calcul et en basse consommation pour réaliser ces algorithmes.

Pour des raisons d'applicabilité de notre proposition à une diversité d'algorithmes, nous prenons le parti dans cette thèse de considérer des paradigmes architecturaux classiques, contrairement à certaines approches récentes et prometteuses de la littérature [12, 13]. Nous privilégions des technologies de processeurs avec une microarchitecture courante, en explorant

comment des modifications de cette dernière pourraient contribuer à atteindre notre but. De plus, nous exploitons l'hétérogénéité des éléments de calcul avec l'idée de couvrir différents besoins algorithmiques à l'aide de processeurs soigneusement choisis. Des approches nouvelles telles que le calcul en mémoire [12] ou le calcul neuromorphique [13] prônent des philosophies différentes en promouvant des paradigmes de calcul et des technologies de mémoires émergentes.

1.4 Contributions de la thèse

Après avoir évalué deux cartes embarquées typiques (GAPuino et Coral Dev Board) pour le traitement d'algorithmes d'apprentissage machine, nous comparons les performances des cartes au regard de leurs différences architecturales. Pour cela, nous exécutons un réseau de neurones convolutionnel de type CNN sur le processeur multicœur de GAPuino constitué de 8 coeurs RISC-V d'une part, et sur l'accélérateur de CNN intégré à la Coral Dev Board. Les enseignements tirés de cette étude préliminaire nous ont conduits à approfondir notre travail sur deux types d'architectures en guise de contributions principales lors de cette thèse :

— **La conception et l'analyse d'une architecture à faible consommation d'énergie adaptée pour l'informatique périphérique.** Nous concevons une architecture multicœur hétérogène basée sur la technologie de processeurs fournie par la société Cortus S.A. Nous évaluons son efficacité énergétique lors de l'exécution de différents types algorithmes. L'objectif est d'appréhender la pertinence de ce type d'architecture, y compris pour des algorithmes de *machine learning*.

— **La proposition d'une unité Multiplication-ACCumulation flexible.** Nous présentons la conception d'une unité Multiplication-ACCumulation (MAC) dédiée aux microprocesseurs afin d'augmenter leurs capacités de parallélisation d'opération MAC très présentes dans les réseaux de neurones artificiels. Cette contribution vise à améliorer les performances des microprocesseurs en leur offrant la capacité de s'adapter à la précision des données traitées, en termes de largeur de bits.

— **L'analyse de l'impact de l'unité MAC proposée dans des architectures d'accélérateurs pour le ML.** Nous définissons une modélisation haut niveau intégrant l'unité MAC définie dans des architectures d'accélérateur de ML connus. L'idée est d'explorer

les conséquences de l'utilisation de notre MAC flexible dans ces architectures, en étudiant de possibles améliorations de l'efficacité d'énergie.

1.5 Plan du manuscrit

Après cette brève introduction aux travaux décrits dans ce manuscrit, le chapitre 2 présente les notions relatives à l'IA et plus particulièrement au ML. Il introduit la problématique de l'exécution du ML dans les systèmes embarqués.

Le chapitre 3 est dédié à un état de l'art des solutions matérielles pour l'implantation efficace de techniques ML dans les systèmes embarqués. Le chapitre est conclu par une évaluation rapide de deux cartes embarquées pour le ML.

Le chapitre 4 présente une analyse d'une architecture multicœur hétérogène que nous avons étudiée dans la thèse. Celle-ci est destinée à supporter l'exécution embarquée de divers types d'algorithmes, y compris de ML.

Le chapitre 5 décrit nos travaux relatifs à la conception d'une unité MAC, destinée aux accélérateurs dédiés à l'exécution de réseaux de neurones. Dans un premier temps, il est présenté les détails de l'architecture de l'unité MAC et dans un second temps une rapide évaluation.

Le chapitre 6 aborde une évaluation de l'impact de l'utilisation de l'unité MAC proposée dans trois architectures d'accélérateurs de ML connus.

Et pour terminer, le chapitre 7 présente la conclusion générale concernant nos travaux, ainsi qu'une discussion des perspectives.

Chapitre 2

Concepts de base sur le *machine learning* (ML) embarqué

Table des matières du chapitre

Ce chapitre présente un aperçu général du contexte dans lequel s'inscrit cette thèse. Dans un premier temps, la section 2.1 est une introduction aux techniques de ML. Face aux contraintes imposées par ces techniques, la section 2.2 présentera des procédés connus dans l'état de l'art pour optimiser leurs utilisations dans les systèmes embarqués. La section 2.3 expose les composants architecturaux clés à prendre en compte pour une exécution efficace des réseaux de neurones sur des systèmes embarqués.

2.1 Introduction aux techniques de ML

2.1.1 Généralités

L'apprentissage automatique est l'une des branches de l'IA, le terme anglais correspondant est le ML. L'utilisation du terme anglais est courante, c'est ce terme qui sera utilisé dans ce manuscrit. Les algorithmes de ML utilisent des données en entrée pour prédire de nouvelles valeurs de sortie. La popularité du ML est forte aujourd'hui, car il a permis un progrès significatif dans la résolution problème complexe considéré comme inatteignable avec les autres approches d'IA [14]. En effet, la résolution de ces problèmes passait par une approche consistant à créer un programme spécifique et personnalisé pour résoudre individuellement chaque problème d'un domaine. Loin de cette approche laborieuse, l'un des avantages du ML est la possibilité de résoudre chaque nouveau problème après un processus préalable d'apprentissage.

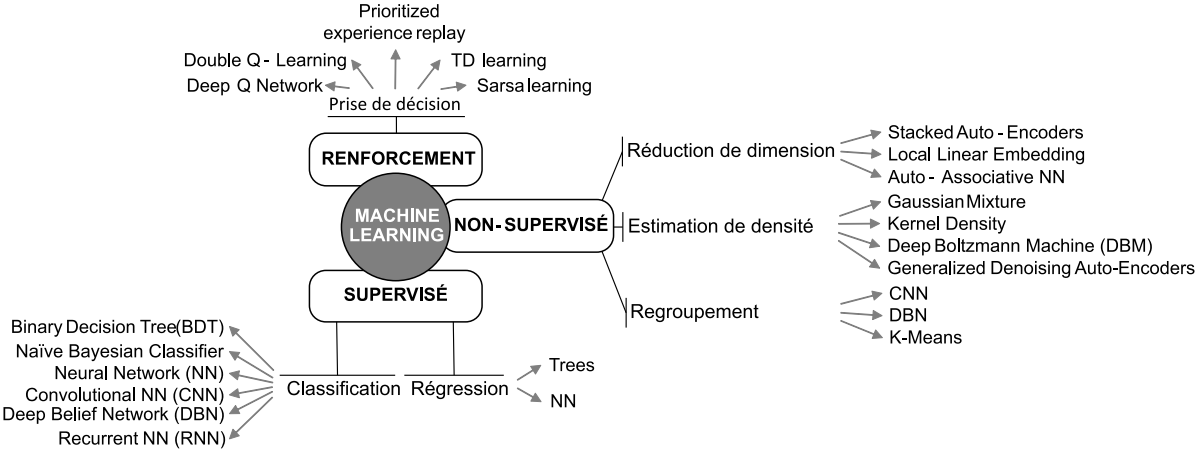


Figure 2.1 – Différents types de techniques de ML [2].

Globalement, l'exécution d'algorithmes de ML passe par deux phases : l'entraînement et l'inférence. La première phase, celle d'entraînement, permet à un algorithme d'analyser les informations issues d'un large jeu de donnée permettant de compléter son modèle. L'entraînement est un processus itératif qui consiste à minimiser l'erreur en fonction des paramètres du modèle de ML. Une fois que l'entraînement est jugé achevé, la seconde phase, celle d'inférence, est opérationnelle pour extraire des informations recherchées dans de nouvelles données.

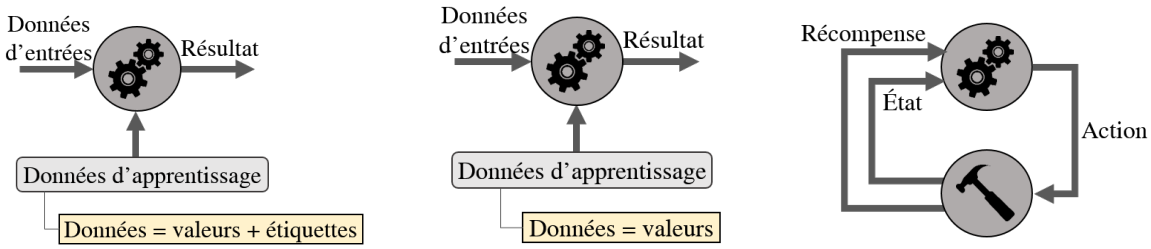
Le ML est un domaine vaste dont trois familles se distinguent [15] : non-supervisé, supervisé et renforcement. La Figure 2.1 présente les trois familles de ML en question ainsi que le type

de problème à résoudre. De plus, il est également présenté sur la Figure 2.1 quelques noms d'algorithmes significatifs de chaque type de problème.

L'apprentissage **supervisé** analyse un ensemble de données étiqueté pour réaliser des tâches de classification ou de régression. Un exemple assez classique est l'utilisation d'une base de données contenant de nombreuses images de chien et de chat. Durant l'entraînement, les images de la base de données sont montrées une à une à l'algorithme, en précisant ce que représentent les images (chien ou chat). Une fois l'apprentissage achevé, l'algorithme sera capable d'identifier ce que représente la prochaine image qui lui sera montrée. Dans le cas de l'exemple, l'algorithme reconnaîtra si l'image représente un chien ou un chat. Le résultat de sortie de l'algorithme se présente généralement sous la forme d'un vecteur contenant un score pour chaque étiquette possible que l'image peut recevoir. Le score traduit la probabilité que chaque catégorie soit la bonne.

Comme l'apprentissage supervisé, l'apprentissage **non supervisé** nécessite un large ensemble de données à la différence près qu'il ne nécessite pas d'étiquettes. Ce type d'apprentissage est utilisé pour l'analyse de données brutes dans le but d'en extraire des informations.

Contrairement aux deux précédentes catégories qui sont orientées sur l'analyse de donnée, l'apprentissage par **renforcement** est plutôt orienté sur les problèmes de prise de décision. La prise de décision est guidée par une récompense positive ou négative déterminée par un retour avec l'environnement. La Figure 2.2 représente brièvement ces trois types d'apprentissages.



(a) Supervisé. (b) Non-supervisé. (c) Par renforcement.

Figure 2.2 – Illustration des types d'apprentissages.

Focus sur les réseaux de neurones

Au sein du ML, il existe de nombreux algorithmes pour la réalisation de tâche de classification et de régression. Cependant, c'est bien l'utilisation de réseau de neurones artificiels qui est le plus populaire et le plus représentatif du ML aujourd'hui. L'une des raisons expliquant

l'attrait pour les réseaux de neurones est leurs capacités à être réentrainées pour répondre à de nouveaux problèmes.

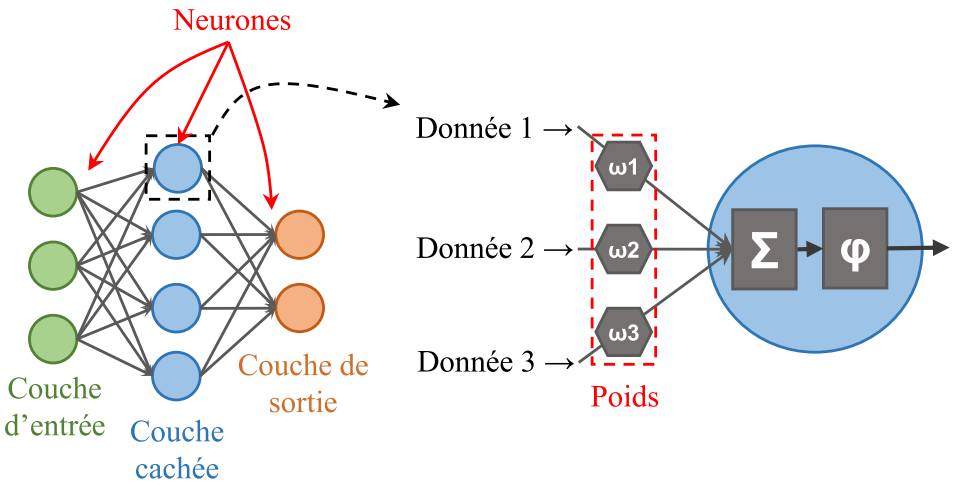
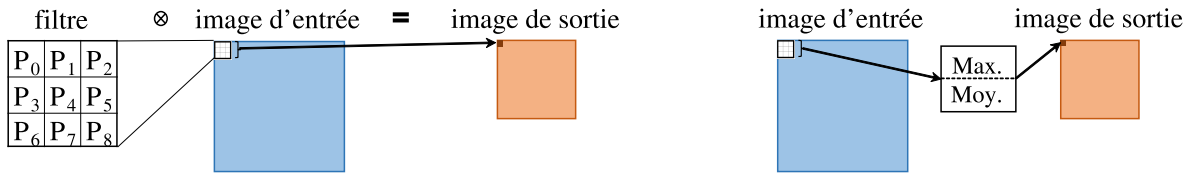


Figure 2.3 – Illustration d'un réseau de neurones (a) et d'un neurone (b).

Un réseau de neurones artificiel s'inspire du fonctionnement du cerveau humain avec l'utilisation de neurones organisés en couches interconnectées entre elles par des synapses. Le principe de fonctionnement d'un neurone est la récupération en entrée des données issues des neurones de la couche précédente. Toutes les données sont additionnées, mais préalablement multipliées par un poids correspondant à la connexion entre les deux neurones. L'obtention du résultat en sortie du neurone est réalisée par une fonction dite d'activation. La Figure 2.3 montre un schéma simplifié d'un réseau de neurones ainsi que le schéma d'un neurone. Lors de la définition d'un réseau de neurones, l'utilisateur peut définir le nombre de couches du réseau, le nombre de neurones de chaque couche et la fonction d'activation. La valeur des poids est directement issue du processus d'apprentissage.

Il existe plusieurs topologies de réseaux de neurones du simple perceptron composée de deux neurones sur la couche d'entrée et d'un neurone sur la couche de sortie à des réseaux de neurones plus complexes. Au-delà de deux couches cachées, le terme associé n'est plus simplement réseau de neurones, mais devient réseau de neurones profond [16–18]. Par exemple les Réseaux de Neurones Récurrent ou encore les *Long Short-Term Memory* (LSTM) qui sont plutôt dédiés à la résolution de problèmes associés à une temporalité. Mais, l'un des réseaux de neurones profonds privilégiés pour la réalisation d'analyse d'image est le Réseau

Neuronal Convolutif, plus connu avec l'abréviation CNN (issu de l'anglais *Convolutional Neural Network*).



(a) Couche de convolution.

(b) Couche de *pooling*.

Figure 2.4 – Illustration du fonctionnement de couche de convolution et de *pooling*.

La particularité des CNN est la manipulation de données de multiples dimensions, le plus commun étant de deux à trois dimensions. Le traitement est effectué à partir de 3 principaux types de couches. La couche principale d'un CNN est la couche de convolutions. Cette couche est composée de filtres permettant l'extraction des caractéristiques de leur entrée. Ces filtres contiennent des poids dont la valeur est ajustée durant le processus d'entraînement. L'image résultant de cette couche de convolution est nommée carte de caractéristiques (feature map en anglais). La couche de *Pooling* a pour objectif de réduire la taille de la sortie issue d'une couche de convolution. Encore basé sur l'utilisation d'un filtre, mais qui ne réalise pas d'opération de convolution et ne contient aucun poids. Le filtre permet d'extraire au choix soit la valeur moyenne soit la valeur maximum des données. Les couches de convolution et de *Pooling* sont illustrées sur la Figure 2.4. Enfin, les CNN sont généralement composés de couches de réseau de neurones *Fully-Connected* (FC) au niveau de leurs dernières couches préalablement précédées d'une couche d'aplanissement permettant la mise au format 1D des données issu des couches de convolutions 2D ou 3D.

Tableau 2.1 – Résumé des paramètres de CNN populaires.

	MobileNet 2017 [19]	AlexNet 2012 [20]	GoogleNet 2014 [21]	ResNet-50 2015 [22]	VGGNet-16 2014 [23]
Input size	224×224	227×227	224×224	224×224	224×224
Nb couche de conv.	22	5	57	53	13
Nb Poids	3,17 M	2,3 M	6 M	23,5 M	14,7 M
Nb op. MAC	564 M	666 M	1,43 G	3,86 G	15,3 G
Nb couche de FC	1	3	1	1	3
Nb Poids	1 M	58,6 M	1 M	2 M	124 M
Nb op. MAC	1 M	58,6 M	1 M	2 M	124 M
Total Poids	4,2 M	61 M	7 M	25,5 M	138 M
Total op. MAC	564 M	724 M	1,43 G	3,9 G	15,5 G

De nombreux réseaux de neurones sont considérés comme des références aujourd'hui. Ils se différencient par des choix différents faits par les concepteurs des paramètres constituant les CNN. Ces paramètres choisis ont un impact significatif sur la complexité des CNN. Pour illustrer leurs niveaux de complexité, le tableau 2.1 montre le nombre de poids et d'opérations MAC de cinq CNN.

L'exécution de ces CNN peut demander 564 millions à 15,5 milliards d'opérations MAC. Ces opérations sont largement comprises dans les couches de convolution. Pour les réseaux présents dans le tableau 2.1, le nombre de couches de convolution est de 5 à 57. Les couches FC présentes dans ces CNN sont au nombre de 1 à 3. Ces couches permettent au réseau de classifier l'image d'entrée. Malgré leurs nombres réduits, les couches FC peuvent être amenées à représenter la majeure partie des poids du système. Cependant, leurs impacts sur l'exécution d'un CNN restent marginaux par rapport aux couches de convolution. La raison est un nombre d'opérations MAC dans la couche FC inférieur de 10 à 1000 fois au nombre d'opérations MAC dans la couche CNN.

Le tableau 2.1 n'est pas exhaustif, il existe d'autres CNN bien connus de la communauté. En complément, on peut citer les réseaux CNN : LeNet [24], OverFeat [25], MobileNet V2 [26], SqueezeNet [27] et ShuffleNet [28].

Exemple: CNN MNIST

Pour rentrer un peu plus dans les détails d'un CNN existant, nous présentons un CNN résolvant la base de données MNIST [29, 30]. Ce modèle est composé de deux couches successives de convolutions à deux dimensions (Conv2D), comme illustré dans la Figure 2.5. La première couche de convolution reçoit une image de dimensions 28x28 issue de la base de données MNIST, contenant des images de chiffres manuscrits en noir et blanc. Cette couche est constituée de 32 filtres de convolution de dimensions 5x5. La deuxième couche de convolution est constituée d'un nombre de filtres doublé par rapport à la première couche de convolution. Soixante mille images servent à l'entraînement du réseau et dix mille au test. En plus des couches de convolutions, le réseau comporte une couche de *Max-Pooling* permettant une réduction de la carte de caractéristiques (*feature map* en anglais), une couche *Flatten* permettant d'aplanir les données 2D en un vecteur 1D requis par les deux dernières couches *Denses*.

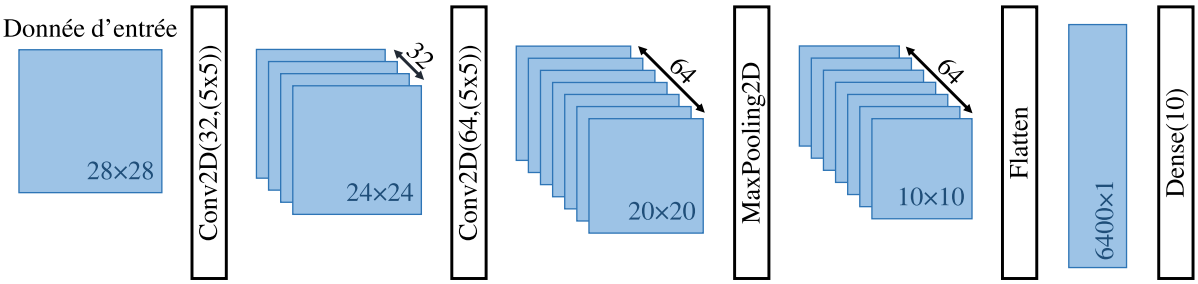


Figure 2.5 – Représentation du CNN utilisée.

2.1.2 Quelques méthodes (pertinentes dans l'embarqué)

L'informatique en périphérie, c'est un récent paradigme visant à changer la nature des interactions entre le domaine de l'internet des objets et l'informatique dans le nuage. Aujourd'hui, de nombreuses applications et services fonctionnent exclusivement via l'informatique dans le nuage, l'informatique en périphérie est l'une des solutions pour permettre une décentralisation apportant son lot de bénéfices (*Edge Centric Computing*). Pour permettre que les données soient traitées au niveau périphérique du réseau, les dispositifs de calcul doivent pouvoir exécuter des techniques de ML. Le terme *Smart Edge Computing* qualifie les systèmes étant au niveau de la périphérie du réseau et ayant les ressources matérielles adaptées pour l'exécution de technique IA notamment de ML.

Dans un cadre orienté système embarqué, la majeure partie des algorithmes de ML mis en œuvre sur les périphériques concerne l'inférence. L'entraînement est donc réalisé soit dans le *cloud* soit sur une machine dédiée. Une fois l'entraînement terminé, l'ensemble des données résultant de l'entraînement sont transmises aux systèmes embarqués. Parmi les raisons de l'absence d'apprentissage, on peut citer l'énergie et le coût du matériel excessif. Les algorithmes de ML ont tendance à demander d'importantes ressources de calcul et de mémoires au vu des énormes quantités de données brutes à traiter et de la complexité des algorithmes. Cela nécessite une grande puissance de calcul et une capacité de stockage des données qui ne sont pas disponibles sur les dispositifs périphériques.

Pourtant, l'intérêt d'utiliser des techniques de ML dans les systèmes embarqués est grand notamment pour les problèmes de classification ou de regroupement. De nombreuses de ces techniques peuvent être utilisées dans un cadre de système embarqué tel que K-means et Support Vector Machines (SVMs) par exemple. Et grâce aux capacités d'apprentissage et de prédiction des réseaux de neurones, leurs utilisations dans des applications liées à l'analyse d'image et de vidéo sont souhaitables.

2.2 Optimisation de l'implantation embarquée du ML

Un support matériel pour exécuter des techniques de ML n'est pas suffisant pour permettre leurs utilisations dans des dispositifs embarqués. Ces techniques doivent réduire leurs complexités pour être utilisables sur des dispositifs embarqués. Différentes techniques existent dont leur objectif est de réduire la complexité de calcul, le volume de données et les ressources matérielles utilisées pendant l'exécution des DNN afin d'améliorer l'efficacité énergétique correspondante.

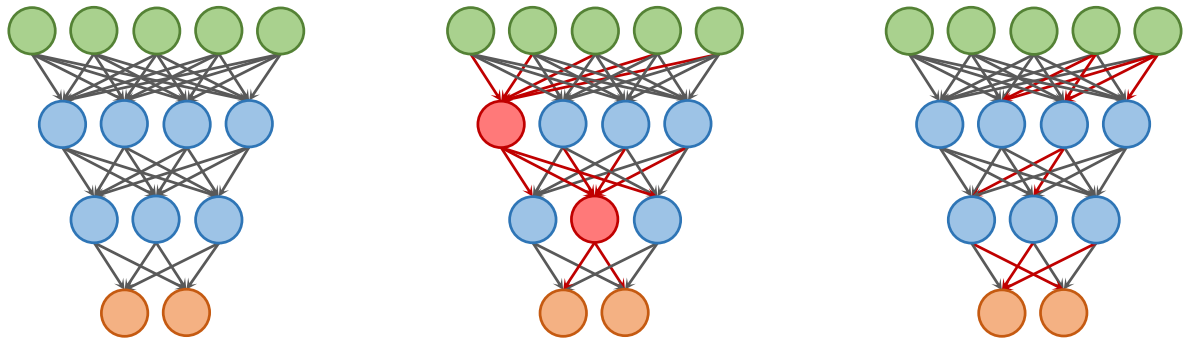
Au vu du défi qu'implique l'exécution de techniques de ML sur des systèmes aux ressources matérielles limitées et contraints en énergie, plusieurs études ont abordé le coût d'exécution de ces techniques, en particulier celles basées sur les réseaux de neurones. Paleo [31] propose une méthodologie qui permet d'évaluer l'exécution sur un GPU d'un réseau de neurones en se concentrant sur ses couches. Il permet donc de localiser les couches les plus lourdes du point de vue du calcul. Dans le même esprit, Neural Power [32] va encore plus loin en proposant l'évaluation de la puissance et de l'énergie consommées par les couches d'un réseau de neurones. Identifier les meilleurs paramètres réseau de neurones pour un problème d'apprentissage est généralement une tâche difficile. Pour la simplifier, il est possible d'utiliser les techniques de recherche hyperparamétrique pour une exploration systématique des meilleurs paramètres. En général, ces techniques se concentrent sur le nombre de couches, le nombre de neurones dans chaque couche, le taux d'apprentissage, etc. Si la précision des réseaux de neurones est souvent la principale préoccupation, les techniques ne prennent pas systématiquement en compte le coût des ressources matérielles requises, c'est-à-dire la mémoire et l'utilisation du processeur, qui ont un impact sur la puissance dissipée. Cela peut être problématique lorsque les modèles ML sélectionnés ne peuvent pas être pris en charge par la plate-forme d'exécution cible. Stamoulis *et al.* ont proposé HyperPower [33] pour prendre en compte les contraintes telles que la puissance et la mémoire dans la conception des réseaux de neurones. Leur solution est basée sur les méthodes d'optimisation bayésienne et de recherche aléatoire, en exploitant les modèles de prédiction de l'utilisation de la puissance et de la mémoire.

2.2.1 Élagage

Les réseaux de neurones sont composés de nombreux poids ayant un impact variable sur le résultat de sortie du réseau. La valeur des poids est fixée durant la phase d'entraînement du réseau de neurones. Il est courant que certains de ces poids soient à zéro ou bien qu'ils aient des valeurs les rendant peu significatifs. Les techniques d'élagage sont développées pour permettre une élimination des poids impactant faiblement la précision du réseau de neurones. Ces techniques sont particulièrement populaires, car elles permettent de réduire à la fois le nombre d'opérations à réaliser et les données à manipuler. Deux types d'élagage peuvent être considérés: l'élagage de neurones et l'élagage de poids. La Figure 2.6a illustre ces deux types d'élagages avec les neurones et connexions élaguées en rouge.

Comme l'illustre la Figure 2.6b, l'élagage de neurone modifie le plus la structure du réseau de neurones. En plus d'un neurone supprimé, il y aura autant de poids que de neurones de la couche précédente et suivante qui sont supprimés. En Figure 2.6, l'élagage d'une connexion a un impact sur la précision moindre que l'élagage d'un neurone. Mais, l'élagage de plusieurs connexions précautionneusement sélectionnées permet un élagage plus fin.

La réduction de la complexité des réseaux de neurones est un enjeu important, de nombreux travaux de recherche sur les techniques d'élagage sont réalisés. Sunil Vadera et Salem Ameen ont réalisé une enquête [34] regroupant jusqu'à 150 études récentes liées à l'élagage. On peut citer comme technique populaire l'élagage basé sur la valeur des poids [35] consistant à supprimer les poids à zéro ou bien trop proche de zéro. Et aussi l'élagage basé sur le partage de poids [36, 37] et sur une approche d'élagage aléatoire [38].



(a) Réseau de neurones complet.

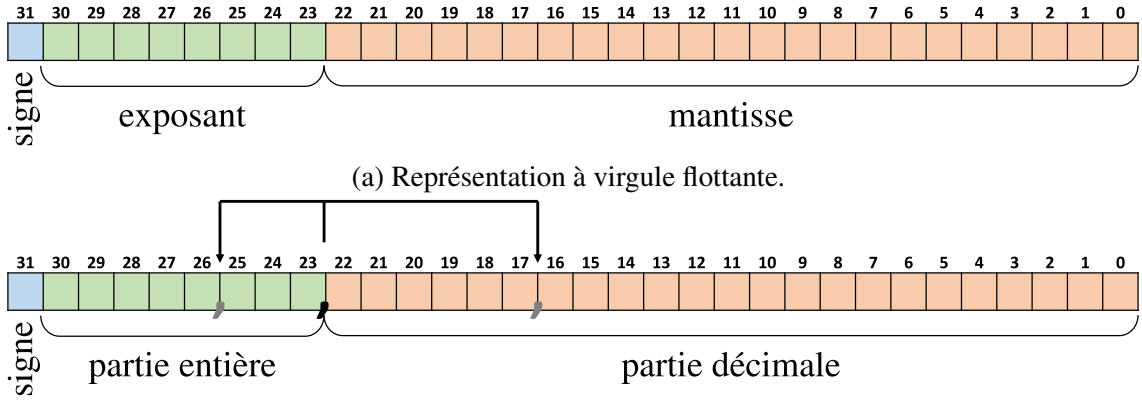
(b) Élagage de deux neurones.

(c) Élagage de plusieurs poids.

Figure 2.6 – Illustration d'élagage d'un réseau de neurones.

2.2.2 Quantification et précision mixte

La quantification est une technique d'optimisation destinée à réduire la complexité de la représentation binaire des données manipulées par un réseau de neurones. Cette technique est particulièrement considérée lors de l'utilisation de systèmes embarqués. Nous pouvons citer deux orientations possibles de la quantification : une première basée sur une transformation d'une représentation à virgule flottante à virgule fixe et une deuxième basée sur une conversion d'une représentation à virgule flottante à entier. L'avantage de ces deux approches est de retirer la nécessité d'avoir de la logique matérielle apte à traiter des données à virgule flottante répondant à la norme IEEE standard 754 [39]. D'un point de vue des systèmes embarqués, cette économie matérielle n'est pas négligeable. De plus, les processeurs embarqués ciblant une utilisation en basse consommation omettent, généralement, le support des nombres à virgule flottante [40–42].



(a) Représentation à virgule flottante.

(b) Représentation à virgule fixe.

Figure 2.7 – Illustration de deux méthodes de représentation de nombre décimal.

La Figure 2.7a est la représentation typique d'une donnée à virgule flottante 32 bits de la norme IEEE standard 754. Les 32 bits de cette représentation sont divisés en trois parties distinctes permettant le support de données allant de 1.17×10^{-38} à 3.4×10^{38}. La Figure 2.7b est une représentation bien plus naturelle d'un nombre décimal avec des bits dédiés à la partie entière et des bits dédiés à la partie décimale et un bit de signe optionnel. L'utilisateur de cette représentation est libre de définir sur les 32 bits ou placer conceptuellement la virgule ainsi en fonction de la nature des données utilisées la partie entière ou décimale peut être privilégiée. L'enjeu principal de l'utilisation de cette représentation est le placement de la virgule. Dans un contexte d'application de cette représentation à des réseaux de neurones, elle permet de limiter les pertes de précision qu'offre la virgule flottante [43].

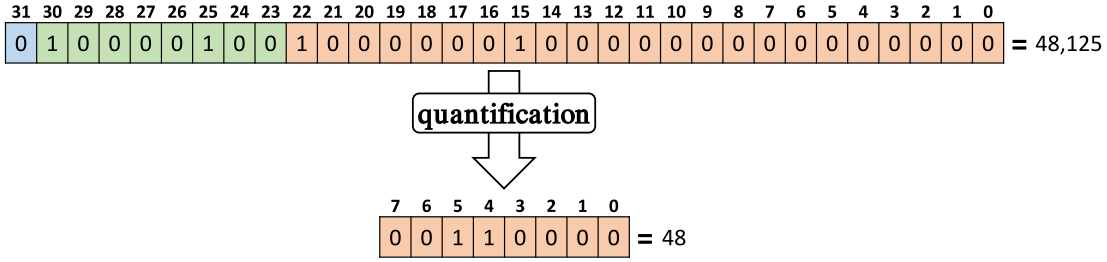


Figure 2.8 – Illustration de la quantification.

Une autre approche de la quantification fréquemment utilisée pour l'emploi de techniques de ML sur des systèmes embarqués est illustrée par la Figure 2.8. Cette approche consiste à convertir une donnée suivant la représentation à virgule flottante du IEEE standard 754 à une donnée représentée seulement par un entier. Ce changement de représentation est réalisé par des fonctions mathématiques et de nombreux travaux de recherche visent à identifier quelle fonction mathématique est la plus adaptée. Cette approche de quantification est principalement utilisée pour réduire le nombre de bits utilisés pour représenter les données. L'exemple de la Figure 2.8 est la réduction du nombre de bits de 32 bits à 8 bits. Ainsi, les concepteurs de systèmes embarqués dédiés à du ML peuvent définir des systèmes supportant des données sur moins de bits et consommant moins d'énergie. L'enjeu avec cette approche est de sélectionner la méthode mathématique la plus adaptée et le nombre de bits évitant une trop grande perte de précision du réseau de neurones [44].

Tableau 2.2 – Comparaison entre la quantification et la précision mixte [7].

	Opération	Précision (%)	Énergie (mJ)
Quantification	8 bits × 8 bits	70.82	31.03
Précision mixte	6 bits × 4 bits	67.51	16.57

En principe, la quantification est utilisée sur toutes les données du réseau de neurones considéré. Mais, il est possible de personnaliser le nombre de bits en fonction du type de données et cette approche se nomme la précision mixte. Par exemple, dans le cadre d'un CNN, les données d'entrée du réseau peuvent être sur 16 bits tandis que les poids du réseau sont sur 8 bits. L'approche de la précision mixte permet d'affiner la quantification [45]. Les données présentes dans le Tableau 2.2 issu de la publication [7] montrent qu'un meilleur compromis peut être atteint. Dans le cas du Tableau 2.2, la sélection d'un nombre de bits personnalisé aux types de données a permis une division par deux de la consommation d'énergie au prix d'une réduction de la précision de 4%.

2.3 Composants architecturaux clés pour du ML embarqué efficace

La conception de systèmes embarqués est soumise à de nombreuses contraintes : énergie accessible, puissance de calcul, encombrement, etc. Malheureusement, ces contraintes sont contradictoires avec les besoins des réseaux de neurones. Pour que les réseaux de neurones soient exploitables, ils ont besoin d'une importante quantité de mémoire disponible et d'une forte capacité de parallélisation des opérations. La complexité des réseaux de neurones et plus particulièrement les CNN est parfaitement personnalisable au problème à résoudre. Bien sûr, plus leurs complexités sont élevées, plus les tâches réalisables peuvent être complexes, par exemple les applications d'analyse vidéo. Les paramètres définissant les réseaux de neurones et CNN ayant un impact significatif sur l'exécution sont ceux impactant le nombre de poids et le nombre d'opérations à réaliser. On peut citer le nombre de couches et de neurones par couche pour les réseaux de neurones et pour les CNN le nombre de couches de convolution, le nombre de filtres et la taille de ces filtres.

Un concepteur d'une architecture dédiée à l'exécution de réseau de neurones doit attacher une attention particulière à la conception de la mémoire et à la conception des unités arithmétiques. Dans cette section, nous allons voir les choix existants dans l'état de l'art pour ces deux aspects.

2.3.1 La mémoire

Globalement, les CNN sont composés de quatre types de données clé essentielles : les données d'entrée d'une couche, les poids, les résultats intermédiaires des opérations et les données en sortie d'une couche. Les CNN peuvent être vues comme de multiples multiplications de matrices d'un point de vue algorithmique, ce sont de multiples boucles imbriquées.

Les performances d'exécution de réseau de neurones et CNN par une architecture sont directement liées à sa mémoire. Nous aborderons l'aspect de la hiérarchie mémoire. Cet aspect a l'objectif de réduire les mouvements des données entre les mémoires et de limiter au maximum le recours au mémoire DRAM coûteux d'un point de vue énergétique et latence.

Nous pouvions distinguer deux types de mémoires dans les architectures accélérateurs de CNN la mémoire hors puce et la mémoire sur puce. La mémoire hors puce est généralement une mémoire DRAM. L'organisation de la mémoire sur puce est l'un des points discriminants des architectures d'accélérateurs. L'enjeu est de définir une organisation mémoire sur puce

capable de limiter l'utilisation de la mémoire hors puce DRAM. En considérant une mémoire sur puce de type SRAM, l'accès à la mémoire DRAM entraîne un coût énergétique 128x supérieur [46, 47].

2.3.2 L'unité multiplication-accumulation (MAC) pour le calcul

Mentionnés précédemment, les CNN peuvent être vus comme une succession de multiplications de matrices. L'opération centrale des multiplications de matrice est le MAC. Basiquement pour réaliser cette opération, un multiplieur suivi d'un additionneur est nécessaire. Comme le montre le Tableau 2.1, les CNN peuvent être composés de plusieurs millions à plusieurs milliards d'opérations MAC pour une seule inférence. L'unité MAC est donc un composant clé dans un système désirant exécuter des CNN. C'est la raison pour laquelle, les systèmes pouvant paralléliser les opérations MAC tels que les GPU ou les accélérateurs de CNN sont privilégiés. Les concepteurs d'architectures d'accélérateurs de CNN cherchent à définir des architectures capables de paralléliser au mieux leur exécution en maximisant le nombre d'opérations MAC réalisé. Pour cela, des unités MAC personnalisées à l'accélérateur de CNN peuvent être conçues. Dans un contexte de systèmes embarqués, les architectures d'accélérateur de CNN sont en majorités conçues pour supporter des CNN quantifiés. D'un point de vue des concepteurs d'architecture, les données représentées sur moins de 32 bits par la quantification permettent de réduire la complexité des unités arithmétiques telles que les MAC. Leur permettant de densifier le nombre d'unités MAC sur la même surface de puce et de limiter la consommation d'énergie.

2.4 Résumé

Dans ce chapitre, nous avons présenté un aperçu des éléments de base concernant les techniques de ML, notamment les réseaux de neurones de type CNN. L'utilisation de ces derniers dans des systèmes embarqués nécessite souvent l'usage d'optimisations logicielles et matérielles. Plusieurs techniques d'optimisation consistent à réduire le nombre d'opérations arithmétiques et à réduire l'empreinte mémoire. La principale conséquence de ces optimisations agressives est une réduction de la précision des techniques de ML et plus spécifiquement celle des réseaux de neurones. Cependant, ces derniers ont montré leurs robustesses face à des changements limités de leurs structures.

L'optimisation fréquemment utilisée dans un contexte des systèmes embarqués est la quantification. Elle permet une réduction du nombre de bits utilisé pour représenter les données.

Ce chapitre a également mis l'accent sur deux composants architecturaux ayant un impact significatif sur l'exécution de réseaux de neurones. Le premier composant essentiel est l'unité arithmétique MAC, car les réseaux en sont majoritairement composés. Le deuxième composant essentiel est la mémoire permettant de stocker l'ensemble des paramètres des réseaux.

Le prochain chapitre approfondit la discussion en proposant un état de l'art sur l'implantation des techniques de *machine learning* dans les systèmes embarqués.

Chapitre 3

État de l'art sur l'implantation embarquée des techniques de ML

Table des matières du chapitre

L'intérêt suscité par l'exécution de technique de ML par des systèmes embarqués a entrainé la conception de nombreuses solutions matérielles que nous présentons dans la section 3.1. Dans la section 3.2, nous présentons les approches existantes permettant la conception de système de calcul. En clôture de ce chapitre, nous proposons une évaluation de deux cartes embarquée destinée au ML dans la section 3.3.

3.1 Architectures embarquées destinées au ML

Face aux performances limitées des systèmes embarqués lors de l'exécution de techniques de ML, des architectures nouvelles sont développées dédiées ou non. Nous abordons trois types d'approches considérées par les concepteurs de systèmes embarqués pour répondre au défi de l'utilisation efficace de techniques de ML: des approches CPU multicœurs, des approches reconfigurables et des approches orientées accélérateurs.

3.1.1 Approches CPU multicœurs

L'approche basée sur l'utilisation de CPU permet au dispositif d'être une solution la plus flexible des approches matérielles considérées pour exécuter des techniques de ML. Les approches polyvalentes basées sur des architectures multicœurs, illustrées en Figure 3.1, permettent d'exécuter tout type de charge de travail. Contrairement aux accélérateurs qui sont restreints seulement à certains types de DNN.

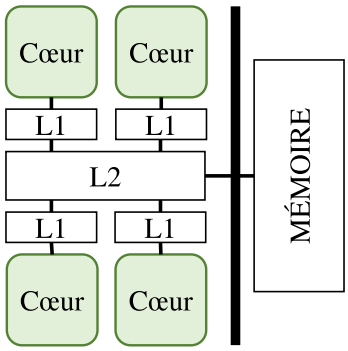
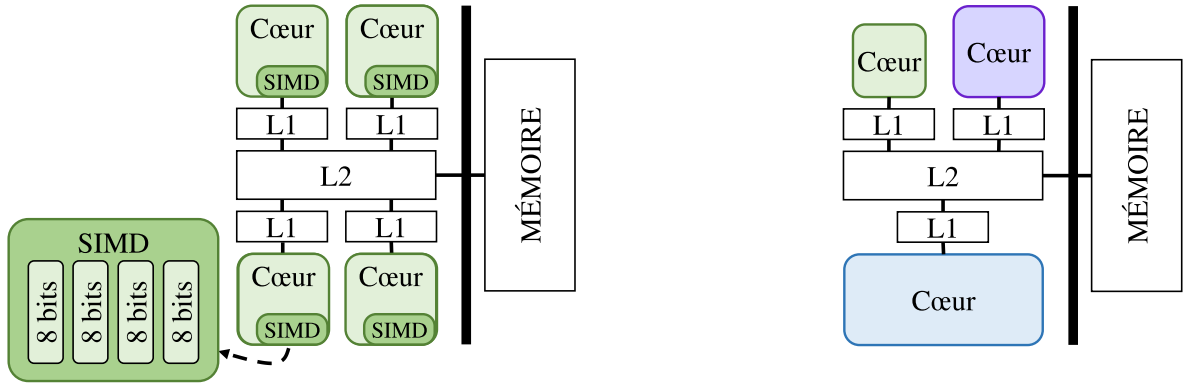


Figure 3.1 – Architecture CPU multicœur typique.

La technologie ARM est souvent adoptée grâce à son efficacité énergétique inhérente à leur objectif d'utilisations dans les systèmes embarqués. Les systèmes peuvent être conçus avec un CPU multicœur donnant accès au parallélisme et d'un GPU pour massivement réaliser des opérations MAC. Par exemple la carte Jetson TX2[48] associant des cœurs ARM Cortex-A57 à un GPU à 256 cœurs Pascal. Ils offrent des performances de calcul plus élevées au prix d'une consommation d'énergie plus importante en raison de la présence des GPU. On peut également citer le dispositif émergeant GAP8 [49], basé sur l'architecture ouverte RISC-V et s'appuie sur l'utilisation d'un accélérateur. Cependant, l'approche multicœur ne suffit pas à être compétitive face pour l'exécution d'algorithmes de ML complexe tel que le CNN. Une solution est d'utiliser

une approche *manycore* comme l'a récemment proposée Esperanto Technologies [50]. Une architecture pouvant atteindre les 1000 cœurs devrait permettre des performances suffisantes, mais cette solution n'est pas forcément idéale pour nombreux dispositifs embarqués du point de vue de la consommation et de la complexité de l'architecture.

L'une des solides solutions est l'utilisation de fonctionnalité *Single Instruction Multiple Data* (SIMD) couramment implantée dans les CPU modernes permettant la parallélisation d'opération arithmétique telle que l'opération MAC, illustré en Figure 3.2a. On peut citer deux exemples typiques, le processeur ARM M-Cortex offre un support SIMD pour les données de 16 et 8 bits [51]. Et le prometteur processeur RISC-V RI5CY [5] supporte également les opérations SIMD de 16 à 2 bits via une extension ISA [6]. Illustré en Figure 3.2b, une autre possibilité est de concevoir une architecture multicoeur hétérogène. Le fondement de cette architecture s'appuie sur l'association de différents types de coeurs permettant d'atteindre une meilleure efficacité énergétique [52, 53].



(a) Architecture CPU multicœur intégrant des opérateurs arithmétiques SIMD.

(b) Architecture CPU multicœur hétérogène.

Figure 3.2 – Schéma simplifié d'architectures.

3.1.2 Approches reconfigurables

L'un des moyens pour obtenir la meilleure efficacité énergétique possible lors de l'exécution d'un algorithme est de concevoir un accélérateur spécifique remplissant exclusivement une fonction donnée. Cependant, cet accélérateur sera performant seulement pour l'algorithme pour lequel il a été conçu. Empêchant toute évolutivité du système ayant cet accélérateur.

Pour contourner ce problème, l'une des solutions est d'utiliser des circuits reconfigurables sur lesquels sera implanté un accélérateur désiré au lieu de figer dans le silicium l'accélérateur.

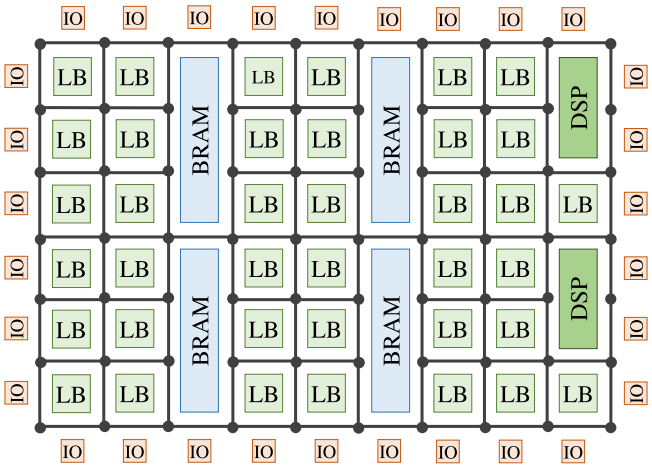


Figure 3.3 – Architecture de puce FPGA.

De plus, l'utilisation reconfigurable permet une réduction de l'effort de conception du circuit. L'un des supports adaptés pour la reconfigurabilité est la puce *Field Programmable Gate Array* (FPGA) [54], illustré en Figure 3.3. Une puce FPGA générique est composée d'un ensemble de blocs logiques configurables (LB), de blocs mémoires (BRAM), de blocs DSP pour des calculs spécifiques et de blocs d'entrée/sortie (IO). Les connexions entre les blocs sont permises par une hiérarchie d'interconnexion configurable. La configuration de la puce FPGA s'effectue classiquement à partir d'un langage de description matérielle permettant de configurer le comportement des blocs composant la puce ainsi que les interconnexions. Néanmoins, des approches de conception à haut niveau peuvent contribuer à faciliter la programmation orientée FPGA [55].

Dans un cadre de ML, les puces FPGA sont utilisées pour aussi bien implanter des architectures d'accélérateurs telles que celle conçue sous un format de matrice d'éléments de calcul [56, 57]. Afin de faciliter la création et l'implantation d'architecture d'accélérateurs, Wen *et al.* [58] proposent un cadre de conception d'architecture composée de plusieurs éléments de calcul personnalisés pour des CNN. Ou bien, il est possible d'implanter directement un algorithme souhaité. Les paramètres et la complexité définissant un CNN et les nombreuses contraintes entourant l'utilisation de puce FPGA rendent l'implantation de CNN laborieuse. Des travaux de recherche tels que ceux de Venieris *et al.* [59] donnent accès à des cadres de conception permettant l'implantation efficace sur puce FPGA de CNN.

3.1.3 Approches orientées accélérateurs

L'une des approches pour permettre une exécution efficace énergiquement des techniques de ML est la conception d'accélérateurs dédiés. C'est particulièrement l'exécution de DNN qui est ciblé par les concepteurs d'accélérateurs. Les DNN ont la particularité d'utiliser beaucoup de données durant leurs utilisations et de demander beaucoup de ressources de calcul. La conception des accélérateurs prend en compte ces particularités en définissant différentes topologies de la hiérarchie mémoire et des ressources de calcul. Cependant, au vu de la quantité de données et de calcul utilisés lors de l'exécution de DNN, actuellement, les accélérateurs supportent seulement l'exécution du processus d'inférence. Le processus d'entraînement est encore réservé aux puissantes machines.

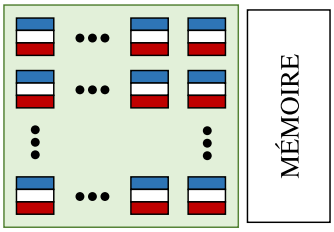


Figure 3.4 – Architecture d'accélérateur de réseau de neurones convolutif.

De nombreuses solutions existent pour concevoir des accélérateurs de DNN, mais l'une des solutions couramment utilisées est de spatialement paralléliser les unités de calculs. Comme montré en Figure 3.4, de nombreux accélérateurs peuvent être représentés sous la forme de matrice d'unités de calcul et d'un bloc mémoire. Ces unités de calcul comprennent généralement une unité arithmétique, une unité de contrôle et des registres de données. L'enjeu pour les concepteurs d'accélérateurs est de convenablement définir la topologie des unités de calcul parallèle et de la hiérarchie mémoire. En effet, l'étude menée par Moolchandani *et al.* [60] sur les accélérateurs de CNN montre la difficulté rencontrée par les concepteurs pour optimiser le parallélisme des calculs à réaliser et réduire autant que faire se peut l'empreinte mémoire. Le Tableau 3.1 présente un échantillon d'accélérateur de CNN. Les accélérateurs EIE [46] et DNPU [61] supportent également les RNN et l'accélérateur QUEST [62] supporte en plus l'exécution de réseau de neurones.

Les concepteurs des accélérateurs ont pris le parti de considérer seulement l'exécution de DNN à valeurs entières et dont la représentation binaire est inférieure à 32 bits. Les

Tableau 3.1 – Comparaison d'accélérateur de technique de DNN.

Nom	Algorithme	Technologie (nm)	Frequence (MHz)	Debit (GOPS)	Puissance (mW)	Eff. énerg. (TOPS/W)	Précision
Eyeriss [63]	CNN	65	200	33,6 - 84	235 - 332	0,15 - 0,35	16b
Eyeriss V2 [64]	CNN	65	200	153.6	-	0,96	8
Thinker [65]	CNN	65	200	364,4	4 - 290 - 447	1,06 - 1,27 - 5,09	8b/16b
B.Moons and M.Verhelst [66]	CNN	40	204	102	76	2,6	1b-16b
Envision [67]	CNN	28	200	76 - 408	7,5 - 290	0,26 - 10	4b/8b/16b
UNPU [68]	CNN	65	200	345,6 - 7371	3,2 - 297	3,08 - 50	1b-16b
DNPU [61]	CNN – RNN	65	200	300 - 1200	34,6 - 279	2,1 - 8,1	4b/8b/16b
Origami [69]	CNN	65	500	74 - 196	510 - 093	0,44-0,80	12b
BRein memory[70]	DNN	65	400	1380	50-600	2,3 - 6,0	1b - 3b
Jaehyeong Sim et al. [71]	CNN	65	125	64	45	1,42	16b
NeuFlow [72]	CNN	45	400	320	600	0,49	-
QUEST [62]	CNN – FC – RNN	40	330	1825	2083	0,88	4
Zhe Yuan et al. [73]	CNN	65	100	9,42E-04 - 13,17	7,3 - 99	0,13 - 13,3	1b/2b/4b/8b
ShiDianNao [74]	CNN	65	1000	194	320	0,61	16b
DianNao [75]	CNN	65	980	452	485	0,93	16b
CirCNN [76]	CNN	45	100	158-400-631	0,79-63,1-440	0,91 - 10 - 100	4b-16b
EIE [46]	CNN – RNN	45	800	102	590	10,49	4b
Cambricon-X [77]	CNN	65	1000	544	954	570,20	16b

avantages sont doubles, la réduction du nombre de bits représentant les données permet à la fois de réduire l'espace mémoire occupé par une donnée et de simplifier l'architecture des unités arithmétique chargée des calculs. De même, l'utilisation de nombre entier permet un allègement de l'architecture des unités arithmétiques. La réduction du nombre de bits représentant une donnée est équivalente à une réduction de précision, c'est une technique connue sous l'appellation de quantification. Nous avons introduit la quantification plus en détail dans la section 2.2. Simplement, les DNNs présentent la caractéristique intéressante de conserver des capacités de prédiction satisfaisante malgré une réduction de la précision des données. Généralement, les accélérateurs supportent des données représentées sur 16 bits ou 8 bits, mais des exceptions émergent avec le support de précision intermédiaire allant de 1 bits à 16 bits.

Du point de vue du débit, les accélérateurs UNPU [68], DNPU [61] et BRein memory[70] se démarquent nettement avec des débits avoisinant les 1300 GOPS. Mais c'est l'accélérateur QUEST [62] qui a le débit le plus élevé du Tableau 3.1 avec un débit annoncé de 1825 GOPS. Ces valeurs de débit sont obtenues pour des précisions basses, dans le Tableau 3.1 les débits affichés correspondent aux précisions en gras. En moyenne, les débits des autres accélérateurs sont aux alentours de 246 GOPS.

La valeur de puissance seule ne permet pas facilement de comparer les accélérateurs entre eux, notamment parce que le nœud technologique n'est pas le même pour tous. Par contre, combiner les débits à la consommation donne l'efficacité énergétique de chaque accélérateur facilitant l'identification du meilleur compromis entre le débit et la consommation. Comme mentionné précédemment, l'accélérateur QUEST [62] a le débit le plus élevé des accélérateurs du Tableau 3.1, mais son efficacité énergétique fait partie des plus faibles. Contrairement, les accélérateurs UNPU [68] et DNPU [61] présentent une efficacité énergétique au-dessus de la moyenne. Cependant, ces valeurs maximum de performance sont à relativiser, car elles sont généralement obtenues pour les valeurs de précision les plus faibles des accélérateurs.

3.1.4 Implantation efficace d'unités MAC

Dans une étude récente de V.Camus *et al.* [85], quatre types d'architectures MAC de précision évolutives dédiées aux accélérateurs d'apprentissage profond sont présentés : Bits en série, Multi-Bits en série, Sous-mots Parallèle et Diviser et Conquérir. Le Tableau 3.2 regroupe des accélérateurs ayant opté pour des architectures d'unité MAC personnalisée pouvant s'adapter à des données représentées sur des nombres de bits différents. Les accélérateurs

Tableau 3.2 – Architecture d'unité MAC.

Accélérateur	Type d'unité MAC	Techniques d'adaptation	Largeur de bit du multiplieur	Nombre de multiplieur	Largeur de bit l'unité MAC	Nb max de MAC / cycle	MULT	Type de données
Envision [67]	Sous-mots Parallèle	Spatiale	8x8b	1	2b à 8b	4	Non	Poids et Caractéristique
QUEST [62]	Bits en série	Temporelle	1x1b	16	1b à 16b	16	Non	Poids
Desoli *et al.* [78]	Sous-mots Parallèle	Spatiale	8x8b	1	2b à 8b	4	Non	Poids et Caractéristique
BitFusion [79]	Diviser et Conquérir	Les deux	2x2b	16	2b à 16b	16	Non	Poids et Caractéristique
UNPU [68]	Bits en série	Temporelle	1x1b	16	1b à 16b	16	Non	Poids
DNPU [61]	Diviser et Conquérir	Spatiale	2x8b	4	2b à 8b	4	Non	Poids
LOOM [80]	Bits en série	Temporelle	1x1b	1	2b à 8b	1	Non	Poids et Caractéristique
Yin *et al.* [81]	Diviser et Conquérir	Spatiale	8x16b	2	8b à 16b	2	Non	Poids
Yue *et al.* [82]	Bits en série	Temporelle	1x12b	1	1b à 128b	1	Non	Poids
Wang *et al.* [83]	Bits en série	Spatiale	1x1b	8	1b à 8b	1	Non	Poids et Caractéristique
Colangelo *et al.* [84]	Bits en série	Temporelle	1x1b	8	1b à 8b	1	Oui	Poids

UNPU [68], DNPU [61] parviennent à supporter des données de différents niveaux de précision. Leur permettant d'avoir à précision faible des débits élevés. Cela repose essentiellement sur une architecture reconfigurable réalisant les opérations arithmétiques MAC. Parmi toutes ces architectures, Diviser et Conquérir se distingue par sa plus grande flexibilité. L'accélérateur matériel de référence mettant en œuvre une telle architecture MAC est BitFusion [79]. Il utilise plusieurs multiplicateurs 2 bits × 2 bits pour traiter des multiplications de largeur de bit variable jusqu'à 8 bits × en 1 cycle d'horloge. Il peut prendre en charge les multiplications sur 16 bits au prix d'un plus grand nombre de cycles d'horloge. Les produits partiels de ces multiplieurs 2 bits × 2 bits sont accumulés avant d'atteindre la sortie.

3.2 Approches d'exploration de l'espace de conception

Différentes approches existent pour permettre une exploration d'architecture [86]. Le choix entre ces différentes approches est guidé par le compromis entre le temps d'exploration et la précision des données résultantes. En d'autres termes, l'utilisateur d'une de ces approches doit définir quel niveau d'abstraction est le plus acceptable.

3.2.1 Intérêt du raisonnement multi-niveaux d'abstraction

L'élaboration d'un système nécessite une étape d'exploration permettant de définir quels sont les choix de conception les plus adaptés. En pratique, l'étape d'exploration passe par différentes phases de conception, chacune de ces phases correspond à un niveau d'abstraction. La procédure classique est de commencer par un niveau d'abstraction élevé décrivant un comportement global du système en cours de développement. Puis de descendre au niveau d'abstraction inférieure pour affiner l'exploration.

Ces niveaux d'abstraction ont leurs particularités et leurs langages de description. Certains langages peuvent être communs à plusieurs niveaux d'abstraction. La Figure 3.5 illustre différents niveaux d'abstraction qui sont détaillés dans la section 3.2.2 suivante. L'intérêt d'utiliser plusieurs niveaux d'abstraction durant l'étape d'exploration est un gain de temps. En effet, les niveaux d'abstraction élevés permettent de décrire un système plus rapidement que les niveaux d'abstraction bas. L'espace d'exploration est plus restreint pour les niveaux d'abstraction élevés, permettant de converger plus rapidement vers un système acceptable. Le niveau d'abstraction peut être ensuite réduit progressivement pour affiner les caractéristiques du système.

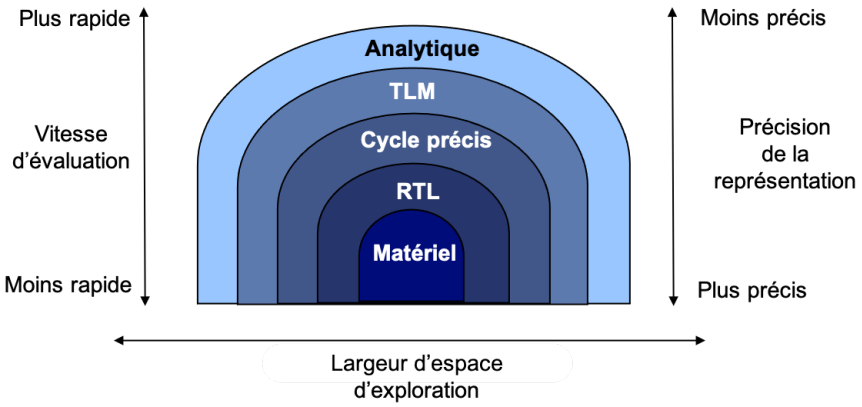


Figure 3.5 – Niveaux d'abstraction pour l'exploration d'architectures (figure inspirée de [3]).

3.2.2 Principaux niveaux d'abstractions

Niveau matériel : ASIC/FPGA

L'une des approches pour évaluer une architecture est de directement l'implanter aux niveaux matériels. C'est l'approche la plus précise, car elle permet d'éviter les éventuels biais qui peuvent exister dans les approches avec un plus haut niveau d'abstraction. Deux solutions sont possibles : une implantation sur ASIC ou FPGA.

L'implantation ASIC est la solution la plus précise, car elle correspond fidèlement à l'architecture définie par le concepteur. Cette solution implique la fabrication du circuit. Par conséquent, cette solution est plutôt dédiée aux phases finales de la conception d'une architecture. Elle se relève donc très limitée pour effectuer de l'exploration.

La solution basée sur l'utilisation de carte FPGA est plus flexible pour effectuer de l'exploration d'architecture. Le concepteur peut analyser le résultat de l'implantation puis effectuer des modifications ou corrections. Cependant, par construction, les FPGA ont des limitations. Même si les FPGA offrent une très bonne alternative aux simulateurs en termes de rapidité, les systèmes implantés sur FPGA pourront difficilement atteindre le GHz. De plus, les FPGA consommeront une énergie supérieure par rapport à un système implanté sur ASIC. Enfin, l'utilisation de FPGA peut être conditionnée par la disponibilité de chaînes d'ingénierie systématiques facilitant la tâche de développement [87–92].

Niveau registre : RTL

Une autre approche avec un faible niveau d'abstractions est de simuler des fichiers RTL utilisant des langages de description matériels comme le VHDL ou le Verilog. À partir de cette

description, le simulateur utilisera des modèles des composants de l'architecture décrite. La précision de ces modèles est largement dépendante des librairies technologiques utilisées et de leurs précisions. De plus, l'obtention de librairie de technologique émergente avec un haut niveau de précision nécessite un partenariat avec un industriel.

La précision de ce type de solution reste satisfaisante. Cependant, du point de vue de l'exploration de l'architecture, le temps de modification de l'architecture par le concepteur sera conséquent [93].

Niveau cycle précis

Une alternative aux approches d'exploration orientée matérielle est l'utilisation des simulateurs logiciels. Ces derniers sont basés sur des langages de programmation de tels que C ou C++. Les simulateurs dits "cycle précis" [94] tirent leurs noms de la précision au niveau cycle du fonctionnement des microarchitectures considérées. L'utilisation de ce type de simulateur donne accès à une bonne flexibilité, cependant à ce niveau de précision, le temps de simulation peut s'avérer lent. Des variantes au "cycle précis" existent connues sous les noms "quasi-cycle précis" ou "cycle approximatif" [95–99].

Niveau transactionnel

Le niveau transaction-level modeling (TLM) [100–104] offre une approche de modélisation et de simulation des systèmes plus rapide qu'au niveau cycle précis. L'approche du TLM est basée sur la modélisation séparant les transactions (communications) des calculs au sein d'un système. Les délais de transmission des transactions ne sont pas pris en compte. Le langage de programmation généralement utilisé est le SystemC.

Niveau analytique

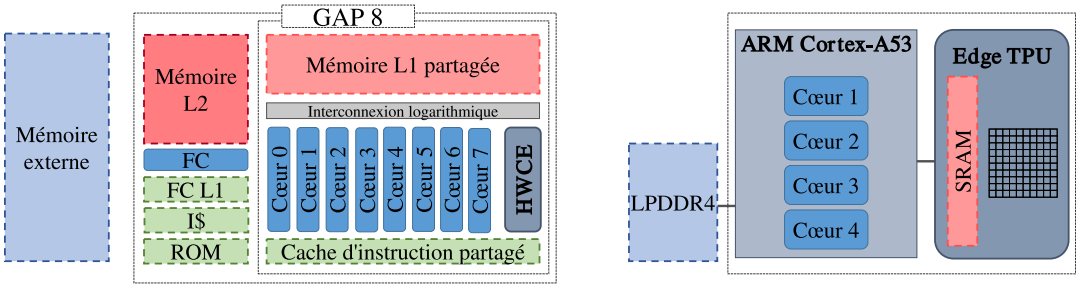
Ce niveau d'abstraction est le plus élevé parmi ceux évoqués jusqu'ici. Le fonctionnement de la modélisation analytique [105–107] est plutôt basé sur des modèles mathématiques décrivant le fonctionnement du système. Le mot d'ordre caractérisant ce type de solution est la rapidité. Elle permet à l'utilisateur une relative simplicité d'utilisation et une vitesse de simulation élevée. Cette rapidité est au prix d'une précision moindre. Dans un cadre lié à la conception d'accélérateur de CNN, Timeloop [108] est un outil permettant d'explorer et d'évaluer des architectures d'accélérateur. L'évaluation se base essentiellement sur l'estimation

de l'efficacité énergétique de l'architecture définie. Cette efficacité énergétique est déterminée grâce à une analyse de l'activité des différents éléments composant l'architecture.

3.3 Évaluation de deux cartes embarquées pour le ML : GAPuino et Coral

Nous évaluons les performances des deux cartes en considérant un benchmark typique de réseaux CNN [109]. Nous nous basons sur la variation de certains paramètres de CNN, à savoir le nombre de couches et de filtres de convolutions. L'idée est de comprendre l'impact de l'organisation mémoire choisie dans chaque cas, tout en explorant de possibles pistes logicielles d'amélioration des performances. En particulier, nous considérons la technique du *Max-Pooling*. Ces deux architectures étant implantées sous forme d'ASIC, nous ne pouvons envisager leur modification matérielle, contrairement à des supports sur FPGA [110] ou définis à l'aide de modèles cycle-précis et flexibles [111].

Approche Multicœur vs Acclerateur



(a) GAPuino. (b) Coral Dev Board.

Figure 3.6 – Architectures des deux cartes électroniques évaluées.

Les systèmes étudiés dans cet article adoptent des organisations différentes concernant la mémoire (cf. Figure 3.6). La carte GAPuino [112] embarque un processeur dédié, le GAP8, composé d'un cluster principal de huit cœurs RISC-V où chaque cœur partage la mémoire *scratchpad* L1 de 64Ko (cf. Figure 3.6a). La mémoire *scratchpad* L2 de 512Ko est à l'extérieur de *cluster* contrôlé par un neuvième cœur RISC-V, nommé *Fabric controller* (FC). Ce dernier assure les fonctions classiques d'un MCU [49] tel que les fonctions de contrôle, communications et sécurité. L'accélérateur est contenu dans le cluster avec un accès direct à la mémoire *scratchpad* L1.

La carte Coral Dev Board, contrairement à la GAPuino, a la capacité d'exécuter Linux [113] grâce au processeur de quatre cœurs A53 de ARM. Sur cette carte, l'accélérateur est un ASIC dit Edge TPU [114] développé par Google (cf. Figure 3.6b). C'est une version allégée du TPU [115] dédiée principalement aux tâches d'inférence. Il contient une mémoire SRAM de 8 Mo pour stocker des réseaux de neurones. Cette mémoire scratchpad est chargée via le processeur ARM. Une mémoire LPDDR4 est accessible aussi pour étendre les capacités mémoires de l'accélérateur.

Cadre expérimental

Nos expérimentations consistent à exécuter sur chaque carte des variantes du modèle de réseau de neurones décrit dans la Figure 2.5, soit en répliquant soit en supprimant les couches de convolution uniquement. Ce sont ces couches qui ont l'impact le plus important sur le temps d'exécution des réseaux. De plus, nous faisons varier leur nombre de filtres internes.

Pour expliquer la notation utilisée dans les figures de cette section, un modèle à N couches de convolution respectivement de $C1$, $C2$, ..., CN filtres est noté "NxConv C1-C2-...-CN". Par exemple, un réseau ayant 2 couches de convolution respectivement de 32 et 64 filtres sera noté "2xConv 32-64". Les autres couches du réseau n'étant pas modifiées, elles ne figurent pas dans la notation. Un appareil d'acquisition de National Instrument [116] est utilisé pour évaluer les différentes exécutions de réseaux sur les cartes.

Effets du *Max-Pooling* sur le temps d'inférence.

L'objectif de la fonction de *pooling* est de réduire la taille de la carte des caractéristiques issue d'une couche convolutive. Le principe est de définir un filtre qui condense un groupe d'éléments de la carte.

Le *Max-Pooling* retient la plus grande valeur de ces éléments. Il réduit ainsi la taille des données traitées par le réseau, sans réduire la précision de ce dernier, tout en favorisant une réduction du sur-apprentissage. Ici, nous ajoutons une seule couche supplémentaire de MaxPooling2D au début des variantes de réseaux évalués. Dans les Figures 3.7a et 3.7b, le nombre de paramètres diminue de 15% à 69% avec une couche de *Max-Pooling*, réduisant le temps d'exécution de 5% à 88% pour la GAPuino et de 0.5% à 50% pour la Coral Dev Board. Cette amélioration du temps d'inférence est double : moins de paramètres signifient moins de calculs à effectuer, et moins de paramètres signifient moins d'éléments à stocker et à solliciter en mémoire. Dans les Figures 3.7c et 3.7d, nous observons que les niveaux de

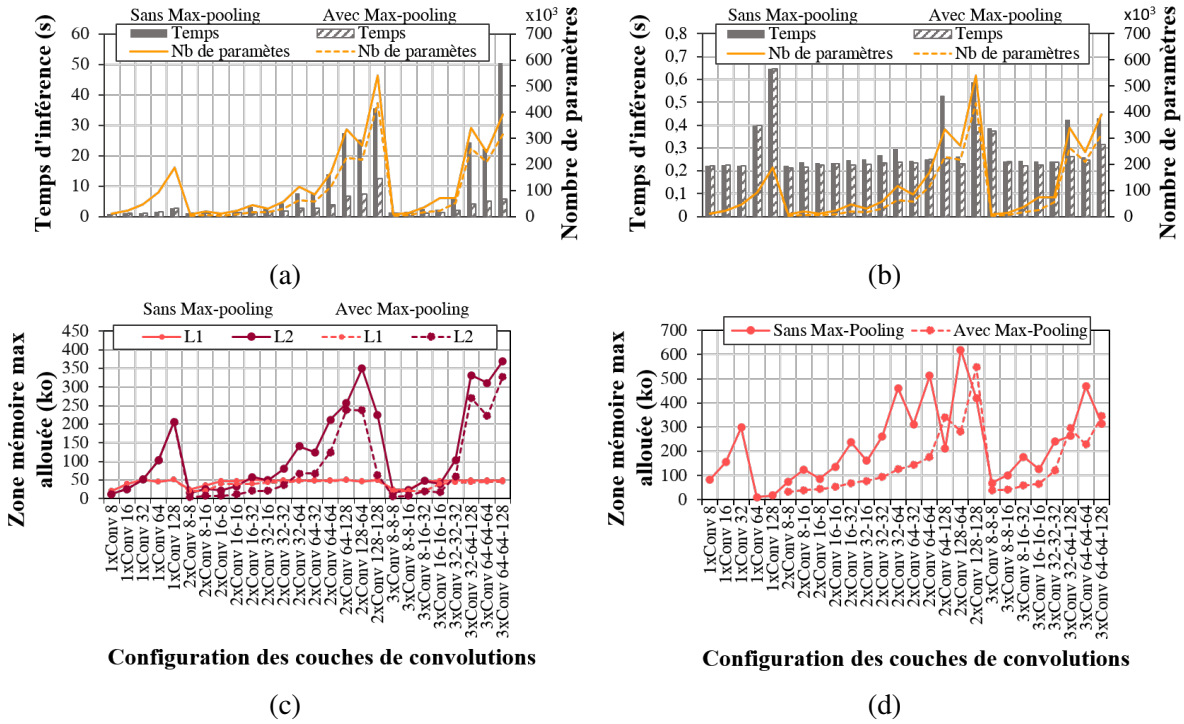


Figure 3.7 – Les figures 3.7a et 3.7b représentent l'évolution du temps d'inférence et du nombre de paramètres de chaque CNN avec ou sans *max-pooling* respectivement de la GAPuino et de la Coral Dev Board. Les figures 3.7c et 3.7d représentent respectivement la zone mémoire allouée max sur les niveaux mémoire L1 et L2 de la GAPuino et la zone mémoire allouée max sur la mémoire *scratchpad* de la Coral Dev Board. Petite précision pour les figures 3.7a et 3.7b, leurs axes x est précisé respectivement sur les figures 3.7c et 3.7d.

mémoire les plus éloignés du CPU sont globalement moins sollicités. Cependant, seulement la configuration "2xConv 128-128" de la GAPuino poursuit l'utilisation de la mémoire externe avec une diminution des accès de 33%. Et quatre configurations de la Coral Dev Board montrent une augmentation jusqu'à 45% de la mémoire *scratchpad* allouée sur la puce EdgeTPU, avec l'abandon de l'utilisation de la mémoire LPDDR4.

Évaluation de la puissance consommée par les cartes

La Figure 3.8 présente la puissance dynamique et l'énergie consommée pour les variantes de réseaux sans Max-Pooling. Ici, la puissance dynamique est définie par la différence entre les puissances consommées "à vide" et pendant l'inférence. Son évaluation repose sur le même principe que [117]. La GAPuino n'exécutant pas de système d'exploitation, sa consommation à vide est proche de la consommation statique de la carte. Cela n'est pas le cas de la Coral Dev Board qui comporte un système d'exploitation. La puissance varie davantage dans la

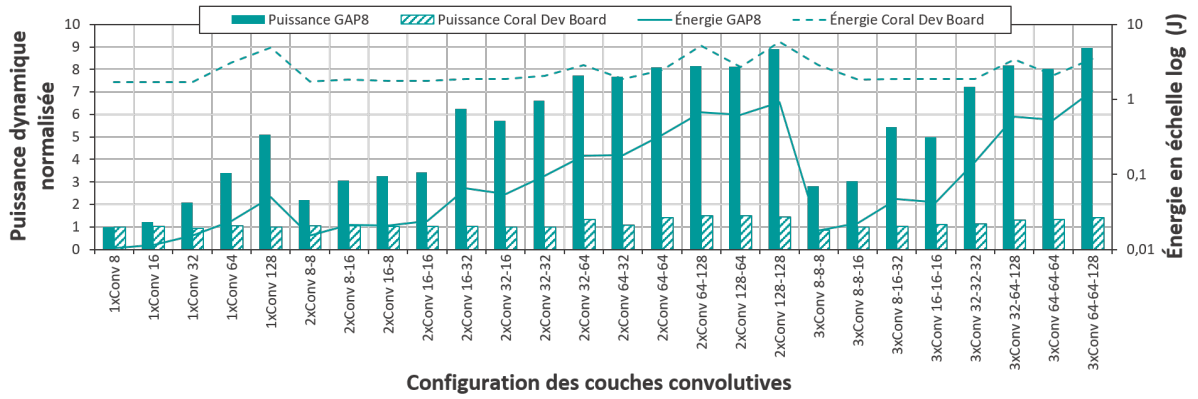


Figure 3.8 – Puissance dynamique et énergie consommée durant l'inférence sans *max-pooling*.

GAPuino en comparaison à la Coral Dev Board. En moyenne, elle est supérieure de 400% et 18% pour les 2 cartes respectives par rapport à la variante de réseau la plus économe, i.e. 1xConv8. L'énergie suit des tendances similaires.

Par ailleurs, à travers les variations des puissances nous observons que les cartes n'utilisent qu'une partie de leurs ressources matérielles pour des petits réseaux. Cela est surtout flagrant pour la GAPuino, dont la hiérarchie mémoire est sollicitée davantage dans l'exécution de réseaux de taille importante. La Coral Dev Board, quant à elle montre une puissance dynamique variant peu. Cela est dû au dimensionnement de la mémoire scratchpad intégrée à la puce EdgeTPU, permettant de traiter *in-situ* la plupart des réseaux évalués.

Quels enseignements tirer ?

L'évaluation préliminaire ci-dessus de deux approches d'exécution adoptées par les cartes GAPuino et Coral Dev Board montre des caractéristiques intéressantes.

L'approche CPU multicœur adopté dans la GAPuino permet avant tout d'exécuter tout type d'algorithmes sans s'enfermer dans l'exécution exclusive de technique de ML et l'approche FPGA peut être reconfigurée au gré du besoin matériel ou de l'évolution des techniques d'IA. La flexibilité offerte par ce type d'architecture en terme d'adaptation de l'utilisation des ressources matérielles en fonction de la complexité des algorithmes permet d'observer une certaine proportionnalité de la puissance dynamique dissipée avec le nombre de couches d'un réseau convolutionnel. Néanmoins, cette architecture CPU multicœur "généraliste" se révèle moins efficace énergétiquement que les accélérateurs pour traiter des CNN. Les secondes architectures étant plus dédiées intègrent des mécanismes adaptés à une exécution des CNN plus performante et moins consommatrice d'énergie. C'est le cas du système implanté dans

la Coral Dev Board. En revanche, cet avantage est acquis au prix d'une flexibilité moindre. Par exemple, en cas d'évolution majeure des techniques de ML, ce type d'accélérateur ultra spécialisé pourrait se révéler inutilisable.

Plus généralement, cette étude préliminaire suggère l'intérêt de creuser la question du choix le plus profitable globalement concernant ces différents types d'architectures, candidates au traitement embarqué efficace en énergie d'algorithmes d'apprentissage machine. Les chapitres suivants de la thèse seront dédiés à ces questions.

3.4 Synthèse

Ce chapitre a présenté un état de l'art sur l'implantation sur l'implantation de techniques de ML dans les systèmes embarqués. Pour cela, ces derniers doivent avoir une hiérarchie mémoire efficace et performante afin d'éviter que celle-ci soit un goulot d'étranglement. Ces systèmes doivent intégrer de hautes capacités de parallélisation d'opérations MAC, souvent présentes dans les réseaux de neurones.

Plus généralement, des optimisations logicielles et matérielles sont nécessaires pour une implantation embarquée efficace. Du point de vue du logiciel, des techniques d'optimisation existantes ont été abordées dans la section 2.2. Du point de vue matériel, la gestion de la mémoire et le nombre d'unités arithmétiques sont les caractéristiques les plus considérées par les concepteurs. En termes d'architectures, diverses approches sont envisageables : les architectures à base de CPU multicœur; les architectures implantées sur des puces reconfigurables comme les FPGA et des architectures d'accélérateurs de réseaux de neurones. Chacune de ces approches matérielles présente des avantages et des inconvénients. On peut tout de même noter que les approches CPU et FPGA sont les plus flexibles et pérennes dans le temps. Les accélérateurs quant à eux visent une exécution de réseaux la plus efficace possible, tout en réduisant la consommation d'énergie et en conservant un temps d'exécution acceptable.

La conception de ces approches matérielles demande un niveau d'expertise et de temps de conception élevé. Pour faciliter le travail des concepteurs matériels, l'utilisation d'outils et de techniques d'exploration matérielle de niveau d'abstraction différent est à considérer.

Les trois prochains chapitres présentent le cœur des travaux réalisés dans le cadre de cette thèse. Ainsi, le chapitre 4 est dédié à l'évaluation d'une architecture multicœur hétérogène. Un des objectifs est d'évaluer les capacités de celle-ci à exécuter efficacement des algorithmes de ML.

Chapitre 4

Approche CPU multicœurs hétérogènes

Table des matières du chapitre

4.1 Introduction

Ce chapitre est consacré aux travaux réalisés sur une architecture multicœur hétérogène destinée à être utilisée pour exécuter des algorithmes de ML. L'évaluation de l'exécution de différents algorithmes sur l'architecture est réalisée grâce à une implantation sur une carte *Field Programmable Gate Array* [118, 119].

Dans un premier temps, nous verrons une description de l'architecture multicœur hétérogène mentionnée. Son implantation sur FPGA sera détaillée ainsi que le dispositif de mesure de la consommation. Dans un second temps, l'architecture sera évaluée à partir d'une série d'algorithmes présentant de différentes caractéristiques. Et dans un troisième temps, une évaluation basée sur des algorithmes de ML sera abordée.

4.2 Motivations pour des architectures multicœurs hétérogènes

4.2.1 Compromis entre nature de calculs et microarchitectures des cœurs

De nombreux paramètres microarchitecturaux influencent les performances d'un cœur de processeur. Tel que la hiérarchie mémoire, les capacités de prédiction de branchement, d'instruction, le niveau de pipeline et les performances des ALU, etc.

Tous les programmes ne se comportent pas de la même manière à l'exécution et présentent des caractéristiques qui leur sont propres. Ces caractéristiques que nous nommerons nature peuvent être :

- *Compute-intensive* : Le programme est composé de nombreuses opérations arithmétiques. L'exécution sera le plus efficace avec des architectures ayant des capacités de parallélisation des opérations.
- *Many branching instructions* : Certaines microarchitectures de cœur embarquent un prédicteur de branchement permettant une amélioration des performances. Cependant, des programmes ne permettent pas une utilisation efficace du prédicteur entraînant de fréquente fausse prédiction. Ces erreurs de prédictions vont créer des pénalités lors de l'exécution.
- *Synchronization-Intensive* : Dans un contexte d'architecture multicœur, l'utilisation de ressource commune ou de données commune entraîne des ralentissements d'accès.

- *High instruction parallelism* : Les programmes ayant de nombreuses instructions seront mieux exécutés par des processeurs avec une grande profondeur de pipeline.

- *Memory-intensive* : Certains programmes font appel à de grands volumes de données. En considérant que la quantité de données est supérieure à la capacité des mémoires caches de nombreux caches *miss* se produiront.

L'impact des caractéristiques de la microarchitecture sur l'exécution d'un algorithme dépend directement de la nature de celui-ci.

Une architecture multicœur hétérogène permet une meilleure adaptabilité à différentes natures d'algorithmes. Lors des expérimentations qui seront détaillées par la suite, nous veillerons à sélectionner des algorithmes qui permettent d'étudier le compromis entre les configurations d'architecture possibles en fonction de la nature de la charge de travail.

4.2.2 La technologie de cœurs Cortus

Le caractère multicœur hétérogène de l'architecture s'appuie sur l'utilisation de trois types de cœurs différents : un cœur haute performance, un cœur basse consommation sans unité de calcul flottant (FPU) et un cœur basse consommation avec FPU. L'approche multicœur hétérogène permet d'obtenir un meilleur compromis entre performance et consommation d'énergie.

Les cœurs embarqués dans l'architecture sont fournis par la société Cortus [120], l'un des leaders dans les services de conception de semi-conducteurs, les systèmes embarqués et les solutions IoT.

Les trois cœurs mentionnés présentent les caractéristiques suivantes :

- APSX2 : Un cœur haute performance applicatif à exécution dans le désordre contenant une unité de gestion de la mémoire et une unité à virgule flottante. Ce cœur est suffisamment performant pour supporter l'exécution de Linux. L'orientation applicative de ce cœur lui donne accès à des caractéristiques telles que la gestion précise des exceptions, la prédiction de branchement et les fils d'exécution multiples. Par rapport aux autres cœurs de Cortus utilisé ici, il offre une plus grande bande passante mémoire grâce à son bus mémoire de 64 bits.

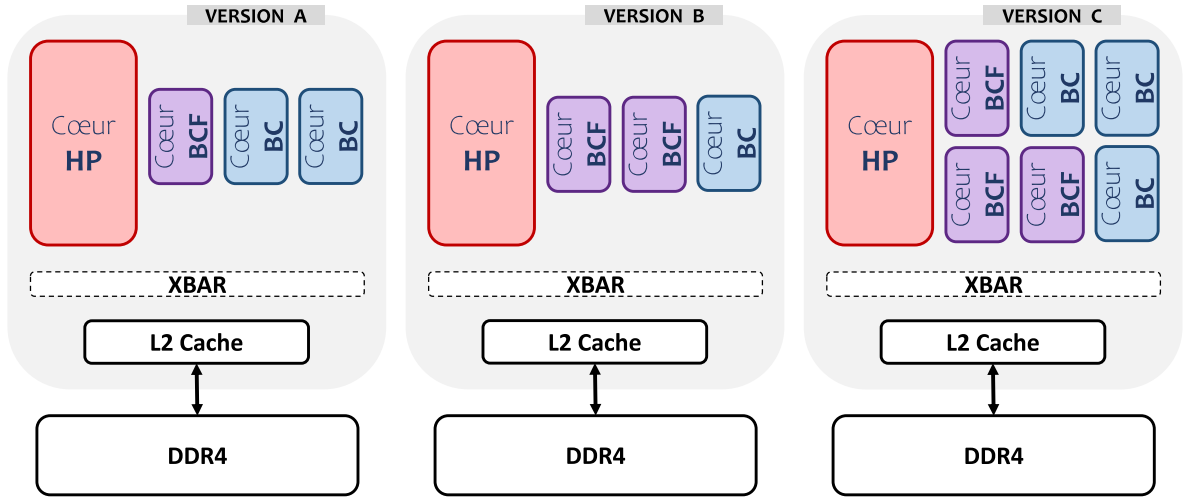
- FPS26 : Un cœur 32 bits extensible, doté d'un FPU. Comme la plupart des cœurs Cortus, il s'appuie sur une architecture Harvard avec un espace d'adressage de 24 Go. Il

convient à la création de systèmes embarqués complexes avec caches, coprocesseurs et cœurs multiples, par exemple dans les applications audio, de vision, de contrôle avancé et de communication. L'arithmétique à virgule flottante profite à un certain nombre d'algorithmes dans ces domaines. Le FPU permet à ce cœur d'exécuter efficacement des codes contenant des variables flottantes tout en restant basse consommation.

- APS25 : Ce cœur est sensiblement similaire aux cœurs FPS26. La différence principale est l'absence de FPU. Cela permet de réduire sa complexité en termes de surface ainsi que sa consommation d'énergie. Et dans un contexte de système embarqué, de nombreux algorithmes ne font pas appel au calcul flottant. L'exécution de calcul en virgule flottante sur ce cœur est réalisée par un mécanisme d'émulation logicielle.

4.3 Architectures considérées

4.3.1 Schémas de principe



(a) Architecture quadricœurs version A.

(b) Architecture quadricœurs version B.

(c) Architecture heptacœurs.

Figure 4.1 – Différents modèles de l'architecture multicœur hétérogène.

La Figure 4.2 présente trois versions de l'architecture multicœur hétérogène considérée. Sur les Figures 4.1a et 4.1b est représenté deux versions de travail préliminaire permettant de valider la conception. Ces deux premières versions que nous nommerons version A et version B ont été utilisées dans une première phase d'expérimentation et ont la particularité

d'être composées de quatre cœurs. Chacune des deux versions a un cœur hautes performances ASPX2 que l'on notera le cœur HP (Haute Performance). La composition des cœurs basse consommation diffère pour les deux versions. Dans la version A, il y a deux cœurs APS25 et un cœur FPS26 et dans la version B, il y a un cœur APS25 et deux cœurs FPS26. Les cœurs APS25 et FPS26 seront respectivement noté cœur BC (Basse Consommation) et BCF (Basse Consommation Flottant).

Le prototype final représenté sur la Figure 4.1c est composé de 7 cœurs. Les six cœurs de basse consommation BC et BCF également répartie sont destinés à l'exécution des charges de travail parallèle, tandis que le cœur de haute performance HP est destiné à l'exécution de charge de travail séquentiel. La communication entre les cœurs et la mémoire partagée est assurée par une interconnexion *crossbars* (XBAR). Chaque cœur a accès à un niveau de cache L1 privé puis à un niveau de cache L2 partagé. La mémoire principale est scindée en zone réservée à chaque cœur et en une zone mémoire partagée entre chaque cœur. La zone partagée permet l'utilisation mutuelle de données.

4.3.2 Programmation orientée multitâche

L'exploitation d'une architecture multicœur passe par l'utilisation d'un modèle de programmation adapté. D'un point de vue syntaxique, le modèle de programmation considéré est proche de la programmation POSIX Threads [121]. Il s'appuie sur la création de tâches permettant l'exécution de fonction préalablement déclarée et définie. Les paramètres des tâches comprennent les arguments d'entrée pris par les fonctions réalisées par chaque tâche, ainsi que les informations de dépendance entre les tâches. Ce modèle de programmation est adapté à l'architecture basée sur la technologie Cortus. Malgré cela, il est possible d'envisager l'utilisation de code écrit en POSIX Threads en raison de leur similarité syntaxique.

Organisation de la mémoire

Chaque cœur a ses zones mémoires propres, utilisées pour le stockage du programme et des données. Une zone mémoire supplémentaire commune entre tous les cœurs permet l'échange de données. Il est ainsi possible de compiler le même programme pour des cœurs implémentant des jeux d'instructions différents. Les fonctions à réaliser par les cœurs sont compilées différemment, ce qui donne des codes-machine, des tailles et des placements mémoire différents, mais la fonctionnalité reste inchangée. Pour l'allocation dynamique de mémoire, les fonctions `smalloc()` et `sfree()` sont disponibles. Un mécanisme de verrouillage de base pour l'accès

exclusif est implémenté dans ces fonctions. Dans les données partagées, trois vecteurs d'états sont fournis : `cpu_ready[4]` pour indiquer si un cœur est prêt, `cpu_valid[4]` pour indiquer à un cœur si les données sont valides et si l'exécution peut commencer, `cpu_assigned_task[4]` pour stocker l'adresse de la tâche à exécuter. De plus, une variable `cpu_lock` est disponible pour implémenter un accès exclusif aux ressources partagées.

Ordonnancement des tâches

Une planification coopérative des tâches est adoptée, c'est-à-dire qu'une tâche se termine avant de passer à une autre tâche sur un cœur donné (ce qui signifie qu'il n'y a pas de changement de contexte). Bien que cette approche soit moins flexible pour les charges de travail en temps-réel, elle est simple et plus efficace pour les charges de travail à forte intensité de calcul. L'exécution multitâche est facilitée ici en donnant aux tâches une liste de dépendances que l'ordonnanceur doit respecter. L'ordonnancement est également dynamique, c'est-à-dire que les tâches peuvent être exécutées par n'importe quel cœur disponible dans n'importe quel ordre lorsqu'il est autorisé. La déclaration des tâches est statique, donc fixée au moment de la compilation. Une bibliothèque de fonctions et de structures de données de niveau utilisateur est fournie pour l'ordonnancement des tâches. Ici, l'ordonnanceur est exécuté sur le cœur HP, qui joue le rôle de cœur " maître " qui attribue les tâches aux cœurs " esclaves " (c'est-à-dire le cœur BCF et les cœurs BC) et à lui-même.

4.3.3 Implantation sur FPGA

Pour permettre l'évaluation de l'architecture multicœur hétérogène, l'utilisation de cartes FPGA est l'un des choix les plus adaptés. En effet, ils donnent la possibilité aux concepteurs d'appliquer des corrections et des modifications rapidement et ceux malgré un coût de développement de départ qui peut s'avérer être conséquent. L'implantation d'une architecture sur une puce FPGA permet l'obtention d'une consommation énergétique plus réaliste qu'une valeur obtenue par simulation.

Dans notre cas les expérimentations qui ont été menées dans ce chapitre sont réalisées au moyen d'une implantation sur deux cartes FPGA : Genesys2 de Digilent [122] et VCU108 de Xilinx [123] (respectivement, Figure 4.3 et 4.4). Les cartes FPGA sont conçues pour correspondre à l'usage de différents utilisateurs. Elles embarquent donc de nombreuses fonctionnalités dont nous n'avons pas eu usage. Par contre, nous sommes plus intéressés par les puces FPGA qu'embarque chaque carte. Le Tableau 4.1 résume les différences de taille des puces FPGA

Kintex-7 et Virtex UltraScale embarqué respectivement sur les cartes Genesys2 et VCU108. L'information principale à retenir est la taille de la puce FPGA Virtex UltraScale qui est 2.5 fois plus grande. Permettant ainsi une implantation d'architecture plus complexe. Autre caractéristique qui sera exploitée, c'est la présence de la mémoire LPDDR4 sur la carte FPGA Virtex UltraScale.

Tableau 4.1 – Résumé des caractéristiques des puces FPGA

	Genesys2 Kintex-7 [124]	VCU108 Virtex UltraScale [125]
LUT	203800	537600
FF	407600	1075200
BRAM	445	1728
DSP	840	768

Les versions quadricœurs A et B de l'architecture respectivement présentée en figures 4.1a et 4.1b, sont implantés sur la carte FPGA Genesys 2. Faute de place suffisante sur la Genesys 2 ,la version C de l'architecture présentée sur la figure 4.1c est implantée sur la carte FPGA VCU108. Le tableau 4.2 résume l'occupation des ressources des puces FPGA.

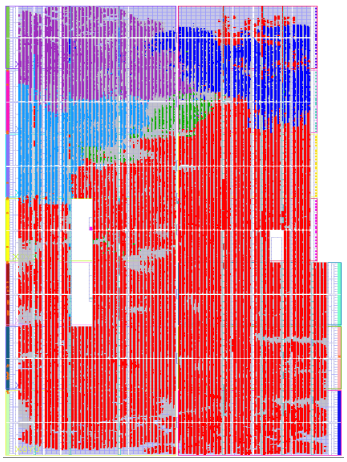
Tableau 4.2 – Résumé des caractéristiques d'implantation.

	Métriques FPGA			
	LUT	FF	BRAM	DSP
Cœur HP	127191	45531	154	17
Cœur BCF	9025	3641	72	0
Cœur BC	4285	1497	72	0
Intercon. (Quadricœur)	2156	197	0	0
Intercon. (Heptacœur)	2512	275	0	0
Quadricœur v.A (Fig. 4.1a)	166765	70796	443	17
Quadricœur v.B (Fig. 4.1b)	171290	72942	443	17
Heptacœur v.C (Fig. 4.1c)	218462	110764	372	20

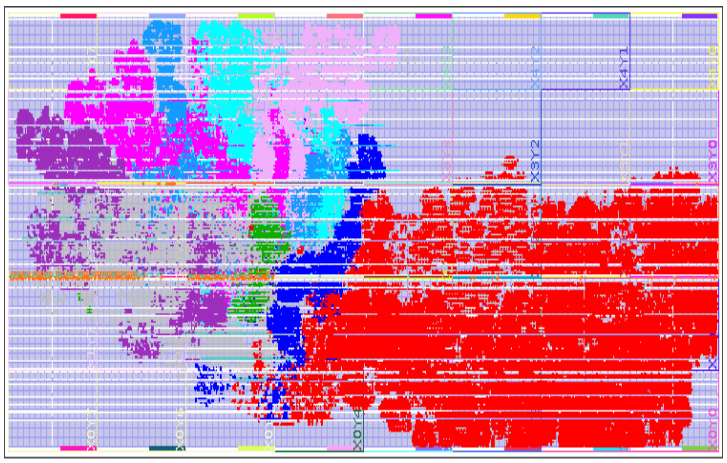
L'instanciation du cœur HP nécessite plus de ressource de la puce FPGA. En moyenne 14 et 30 fois plus de ressources sont utilisées respectivement par rapport au cœur BCF et BC. La présence d'un FPU dans le cœur BCF double sa surface par rapport au BC. Les versions quadricœurs sont parfaitement supportées par la carte FPGA Genesys 2 en occupant 81% et 84

% des *LookUp Table* (LUT) disponibles. La version heptacœur demande une puce FPGA ayant plus de ressources. Son implantation sur la carte VCU108 utilise 40% de ces LUT.

Les figures 4.2a et 4.2b sont des représentations graphiques de l'occupation des puces FPGA occupées respectivement par les architectures 4.1b et 4.1c. Cette représentation graphique est directement issue du logiciel Vivado de Xilinx . La figure 4.2a illustre bien le taux de remplissage supérieur à 80% de la puce FPGA Kintex-7 pour la version A de l'architecture quadricœur. La version heptacœur est implantée sur la carte FPGA VCU108 visible sur la Figure 4.2b. L'espace restant disponible peut donner l'opportunité d'explorer d'autres choix architecturaux sur la carte FPGA VCU108. Sur les deux figures, la partie en rouge représente la surface occupée par le cœur HP. Les variantes de bleu représentent les cœurs BC et les variantes de rose représentent les cœurs BCF. Sur la Figure A, on distingue la présence de deux cœurs BC et d'un cœur BCF et sur la Figure B les trois cœurs BC et BCF sont visibles. La surface occupée par les six cœurs orientés basse consommation (BC et BCF) est inférieure à celle du cœur HP. Enfin, la partie en vert montre les ressources de la puce FPGA utilisées par les interconnexions entre les cœurs.



(a) Architecture version A sur la carte FPGA Genesys 2.



(b) Architecture version C sur la carte FPGA VCU108.

Figure 4.2 – Illustration de la synthèse accessible via Vivado [4] de l'architecture quadricœur version A sur la carte FPGA Genesys 2 (4.2a) et heptacœur version C sur la carte FPGA VCU108 (4.2b).

4.4 Évaluation des architectures proposées

4.4.1 Cadre expérimental considéré

Dispositif de mesure de la consommation de cartes FPGA

L'acquisition de la consommation de la carte Genesys2, illustrée sur la figure 4.3, est inspirée de l'approche JetsonLeap [126]. Cette approche consiste à acquérir des échantillons de Différence De Potentiel (DDP) aux bornes d'une résistance de *shunt* via un appareil d'acquisition de tension de National Instrument [116]. Un programme dédié déterminera la valeur du courant traversant la résistance de *shunt* et déduira l'énergie qui est consommée. Le placement de cette résistance de *shunt* est possible sur la carte FPGA Genesys 2, car les différents rails d'alimentation de cette carte peuvent être alimentés directement sans passer par l'alimentation centralisée (12V) de la carte.

Les cartes FPGA offrent de nombreuses possibilités d'utilisation pour l'utilisateur. Sur les cartes FPGA, on peut trouver un microcontrôleur, divers boutons-poussoirs, interrupteurs, LED, des afficheurs 7 segments, écran, des capteurs environnementaux et d'autres. La présence de tels composants électroniques sur la carte entraîne des surcoûts en consommation d'énergétique.

Ainsi en mesurant la valeur d'énergie directement sur le rail d'alimentation de la puce FPGA, la valeur d'énergie obtenue est celle uniquement consommée par la puce FPGA et non pour l'intégralité de la carte FPGA. Et pour accroître la précision des mesures, l'acquisition des échantillons de DDP est déclenchée par un événement logiciel. Ainsi, seule la partie du code exécuté ayant un intérêt sera mesurée.

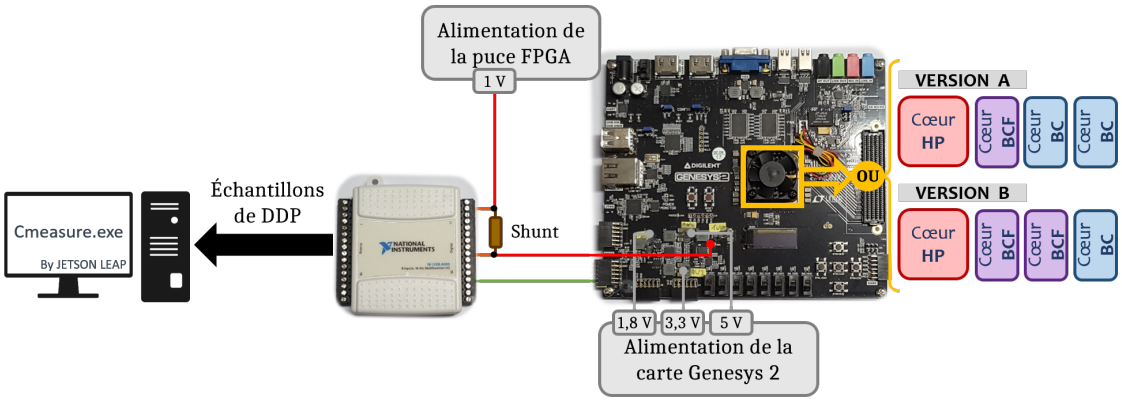


Figure 4.3 – Dispositif de mesure de la consommation de la carte Genesys 2.

Contrairement à la carte FPGA Genesys2, la carte FPGA VCU108 embarque des capteurs de courant et de tension sur chacun de ses rails d'alimentation principaux. Pour permettre l'exploitation de ces capteurs, il faut utiliser une API développée par Xilinx appelée SysMon. Qui une fois activée sur la carte FPGA permet le stockage des données issu des capteurs dans des registres dédiés. Comme illustré sur la figure 4.4, ces données sont ensuite accessibles via le terminal de commande du logiciel Vivado lorsque la carte FPGA est connectée à un ordinateur par USB.

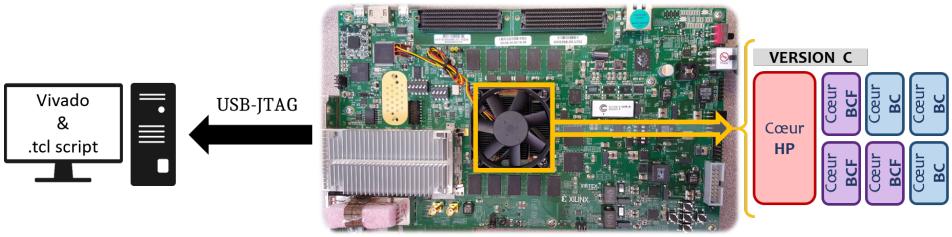


Figure 4.4 – Dispositif de mesure de la consommation de la carte VCU108.

Programmes d'évaluation de l'architecture quadricœur

L'évaluation des versions A et B de l'architecture hétérogène présentée précédemment est réalisée à l'aide d'algorithmes représentatifs de différentes charges de travail typique des systèmes embarqués. Au vu de la conception multicœur hétérogène de l'architecture, les programmes ont été réencodés comme des programmes multitâches parallèles. La plupart d'entre eux consistent en un ensemble de tâches identiques, c'est-à-dire que chaque tâche réalise la même fonction. Le tableau 4.3 résume les douze programmes utilisés.

Les *benchmarks* **I-Factorial** et **F-Factorial** implantent des algorithmes qui calculent la factorielle de nombres entiers et flottants respectivement. Ce sont les seuls programmes séquentiels considérés dans nos expériences. Par conséquent, ils ne seront exécutés que sur des configurations à un seul cœur. Cela permet de comparer les trois types de cœurs faisant partie du portefeuille d'IP de Cortus.

FFT [127] est un programme où chaque tâche exécute la même transformée de Fourier rapide.

I-Matmul et **F-Matmul** codent une multiplication de matrice, respectivement sur des matrices entières et des matrices à valeurs flottantes. Contrairement aux autres programmes parallèles, ces deux programmes sont encodés de telle manière que chaque tâche calcule une colonne différente de la matrice résultante.

Tableau 4.3 – Programmes d'évaluation sélectionnés.

Programmes	Charge de travail parallèle	Nombres Flottants	Nature
I-Factorial	Non	Non	Compute-intensive
F-Factorial	Non	Oui	Compute-intensive
FFT	Oui	Oui	Compute-intensive
I-Matmul	Oui	Non	Compute-intensive
F-Matmul	Oui	Oui	Compute-intensive
Mpeg	Oui	Oui	Compute-intensive
RandNumCmp	Oui	Non	Branch instructions
HashSync	Oui	Non	Sync.-intensive
InstPar	Oui	Non	Instr. parallelism
Bitonic	Oui	Non	Memory-bound
KNN	Oui	Oui	Memory-bound
Stencil	Oui	Oui	Memory-bound

Le programme **Mpeg** est une charge de travail parallèle typique de traitement vidéo et audio qui nécessite une grande quantité de calculs.

Le programme **RandNumCmp** encode un algorithme constitué d'une boucle qui itère cinq instructions de conditions *si* successives. Toutes les conditions booléennes de ses instructions dépendent d'une valeur entière aléatoire. L'objectif de ce programme est de rendre la prédiction des branches difficile pour le processeur. Par conséquent, il en résultera un nombre élevé de mauvaises prédictions de branche.

Le programme **HashSync** implante un algorithme qui déclenche des accès fréquents à une table de hachage partagée et synchronisée. Chaque tâche réalise le calcul d'une clé correspondante à une ligne de la table de hachage et y insère certains éléments. En spécifiant un nombre élevé de tâches, ce programme permet de reproduire le comportement des charges de travail intensives en synchronisation.

InstPar est un programme simple qui contient une séquence d'opérations indépendantes pouvant être exécutées en parallèle. Les processeurs avec des pipelines d'instructions plus profonds exécutent efficacement un tel programme.

Les trois derniers programmes, **Bitonic** [128], **K-Nearest Neighbours** (KNN) [128] et **Stencil** [129], ont en commun d'être liés à la mémoire. Ils peuvent traiter de grandes structures de données de type tableau qui entraînent de nombreuses pertes de cache. Bitonic est un algorithme qui trie les éléments d'un tableau par ordre croissant. KNN met en œuvre un algorithme de classification couramment utilisé dans l'apprentissage automatique. Il s'appuie sur le calcul des distances entre les points d'un espace bi-dimensionnel. Enfin, le repère Stencil, souvent utilisé dans le traitement de l'image, consiste en des algorithmes de moyenne de

cellules matricielles. Étant donné une cellule, il calcule la moyenne des valeurs de la cellule actuelle et de ses quatre cellules adjacentes.

4.4.2 Résultats

Les résultats présentés ici ont été obtenus avec l'exécution des programmes présentés précédemment et résumés dans le tableau 4.3. C'est les versions A et B des prototypes quadricœurs de l'architecture, illustrés sur les figures 4.1a et 4.1b qui ont été utilisés. Dans un premier temps, les résultats exposés seront sur une évaluation de l'efficacité énergétique de l'ensemble des configurations permises par les architectures quadricœurs. Puis, dans un second temps, les enseignements tirés de l'évaluation seront utilisés pour observer le potentiel gain d'énergies que peut offrir une architecture hétérogène.

Efficacité énergétique de l'architecture quadricœur face à différents programmes.

Les Figures 4.5 et 4.6 représentent les gains d'énergies obtenues lors de l'exécution des programmes respectivement de nature *compute-intensive* et les natures restantes. Par souci de simplicité dans la notation des configurations au sein des légendes des Figures 4.5 et 4.6, les notations suivantes sont adoptées pour coder les différentes configurations d'architecture : un seul cœur : 1X deux cœurs : 1X 1Y, et 2X où X et Y désignent soit le cœur HP (abrégé en H), le cœur BCF (abrégé en F) et le cœur BC (abrégé en I). Par exemple, la configuration 1H 1F 2I désigne la configuration quadricœur complète décrite à la figure 4.1a. Pour améliorer la lisibilité, les données représentées sur les figures sont normalisées sur l'exécution monocœur HP. Le choix de la normalisation s'est porté sur HP au vu de ses capacités hautes performances.

Sur la figure 4.5 représentant le gain d'énergie lors de l'exécution des programmes de nature *compute-intensive*, globalement les résultats obtenus montrent qu'une exécution monocœur avec le cœur HP est le plus économe en énergie qu'avec les deux autres cœurs. Pour le programme I-Factorial, nous observons que le cœur HP est 8% plus efficace que le cœur BCF. Au vu de l'absence de calcul à virgule flottante dans ce programme et de la similitude des architectures des cœurs BC et BCF, on pourrait s'attendre à la même efficacité énergétique pour ces deux cœurs, mais BCF est légèrement meilleur, de 4 %. Avec la version en virgule flottante du programme, c'est-à-dire le F-Factorial, l'avantage du FPU dans le cœur BCF devient clairement visible. L'efficacité énergétique par rapport aux cœurs BC est meilleure d'environ 91% comme le montre la Figure 4.5. Plus important encore, pour les programmes effectuant des opérations intensives en virgule flottante telle que F-Factorial, FFT et F-Matmul,

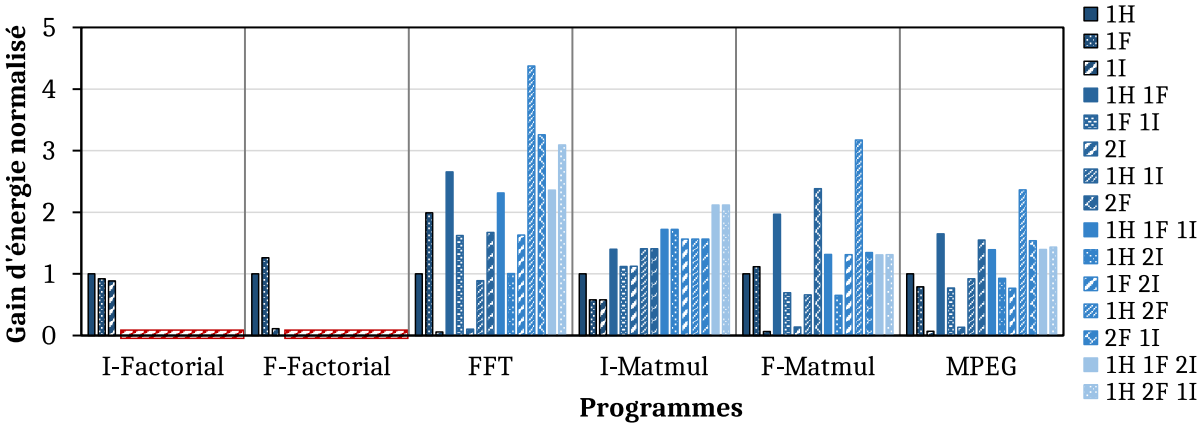


Figure 4.5 – Comparaison de la consommation d'énergie normalisée pour les programmes *compute-intensive* évalués. Petite précision de lecture du graphique, les programmes I-Factorial et F-Factorial sont exécutés seulement en configuration monocœur. L'encadré rouge rappelle donc l'absence de données pour les autres configurations.

le cœur BCF se montre plus efficace que le cœur HP de 26%, 96% et 11,5% respectivement, en raison de l'optimisation agressive du FPU dans le cœur BCF.

Les configurations multicœurs montrent que seuls deux cœurs BCF combinés à un cœur HP offrent la meilleure efficacité d'énergétique pour les programmes *compute-intensive* en virgule flottante FFT et F-Matmul. En l'absence de calcul en virgule flottante, comme dans le programme I-Matmul, la configuration quadricœur complète présentée dans la Fig. 4.1a est la meilleure.

Enfin, en présence de calculs en virgule flottante, les deux architectures quadricœurs complètent représentées sur la Fig. 4.1a et la Fig. 4.1b présentent une consommation d'énergie similaire pour les programmes parallèles F-Matmul et Mpeg. Cela s'explique par l'importante surcharge de temps d'exécution induite par les cœurs BC dans les deux configurations d'architecture, qui masque toute amélioration apportée par le cœur HP et les cœurs BCF. Pour le programme FFT, l'architecture quadricœur 4.1b incluant donc deux cœurs BCF est légèrement meilleure que l'autre configuration quadricœur. De manière plus générale, nous observons que le meilleur gain d'énergie pour les programmes parallèles *compute-intensive* en virgule flottante est obtenu avec la configuration incluant le cœur HP et deux cœurs BCF. Lorsque l'on ajoute des cœurs BC, qui ne comportent pas de FPU dans leur microarchitecture, les performances globales se dégradent malgré un parallélisme d'exécution plus important dû à un plus grand nombre de cœurs.

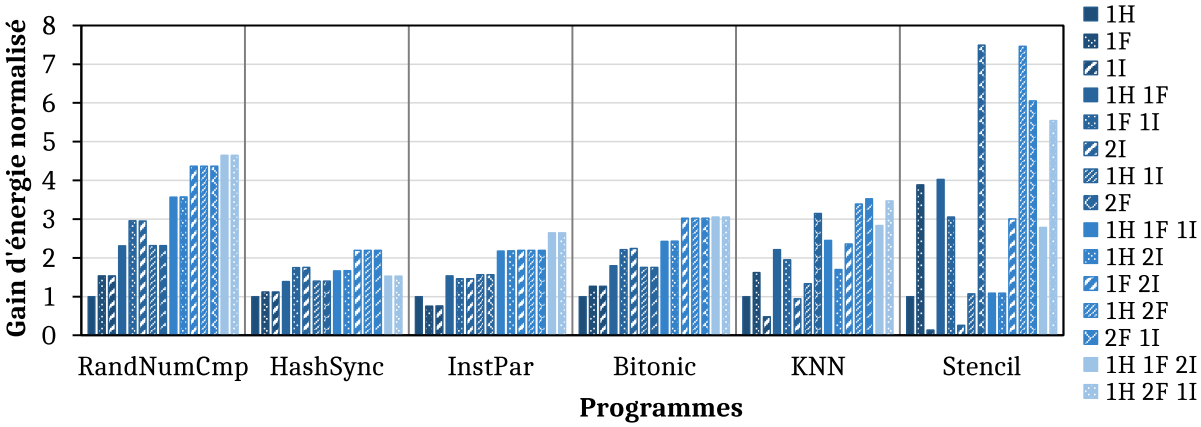


Figure 4.6 – Comparaison de la consommation d'énergie normalisée pour les programmes évalués (*many branching instructions*, *synchronization-intensive*, *high instruction parallelism* et *memory-intensive*).

Contrairement aux observations précédentes faites pour les programmes *compute-intensive*, sur la Figure 4.6 le programme RandNumCmp contenant un nombre élevé d'instructions de branchement présente une consommation d'énergie similaire pour les cœurs BCF et BC dans les exécutions monocœur. Le cœur HP est le moins efficace avec une plus grande consommation d'énergie, les cœurs à faible consommation ont une meilleure efficacité énergétique de presque 50%. Cela s'explique par l'impact néfaste des erreurs fréquentes de prédiction de branchement sur la microarchitecture du cœur HP. En effet, le processeur doit souvent revenir sur tous les résultats intermédiaires lorsque la prédiction s'avère erronée. : cela implique de vider le pipeline d'instructions du cœur, qui doit en outre sauvegarder et restaurer des structures. Dans les configurations multicœurs, même si une utilisation des quatre cœurs est plus efficace sur le plan énergétique, son gain est marginal par rapport à une configuration ne comportant que trois cœurs à faible consommation. Cet écart énergétique minimal est induit par la pénalité de mauvaise prédiction de branche sur le cœur HP avec quadricœur.

Pour le *benchmark* InstPar, qui se caractérise par un parallélisme d'instructions élevé, le cœur HP est plus économe en énergie que les cœurs BC de 33% (voir la Figure 4.6). Ceci est favorisé par la microarchitecture avancée du cœur HP, par exemple, exécution dans le désordre, étages de pipeline plus profonds. L'exécution parallèle de ce programme améliore l'énergie de 50 % par rapport à une configuration monocœur. La plupart des configurations avec un nombre de cœurs équivalent ont une consommation d'énergie comparable, tandis que celles incluant le cœur HP s'exécutent 25 % plus rapidement que les configurations utilisant uniquement des cœurs de faible puissance.

Alors que les observations ci-dessus étaient attendues pour le programme InstPar, un résultat différent est obtenu pour le programme HashSync, qui est de nature en *synchronization-intensive*. Dans une configuration monocœur, l'exécution de ce programme montre que les cœurs à faible puissance sont légèrement plus économes en énergie que le cœur HP, comme l'illustre la Figure 4.6. Dans les configurations avec un nombre équivalent de cœurs, celles incluant le cœur HP sont moins efficaces, de 25 % par rapport aux cœurs à faible puissance uniquement. Ceci s'explique par la surcharge induite par la gestion de la microarchitecture du cœur HP en présence de synchronisations de tâches. Plus précisément, cette surcharge provient des coûteux changements de contexte qui se produisent dans la microarchitecture complexe de ce cœur spécifique (contrairement aux cœurs de faible puissance). L'algorithme HashSync implique de fréquentes suspensions et reprises de tâches. Au final, il y a un surcoût important de temps d'exécution dû au vidage des étages du pipeline.

L'algorithme de tri Bitonic montre que l'exécution sur deux cœurs permet d'obtenir une amélioration de l'efficacité de l'ordre de 40% avec le cœur HP et 70% avec le cœur BC par rapport à un seul cœur (voir la Figure 4.6). Les configurations multicœurs avec le même nombre de cœurs ont un temps d'exécution comparable comme le montre la Figure 4.6, tandis que celles qui n'utilisent que des cœurs à faible puissance sont plus efficaces sur le plan énergétique. La seule exception concerne la configuration à 3 cœurs associant un cœur BCF et deux cœurs BC et la configuration quadricœur, qui ont une consommation énergétique équivalente. Cette dernière configuration présente un gain de 24% en temps d'exécution par rapport à la première.

Dans le cas du programme KNN, un écart énergétique de 70% entre un cœur BC et un cœur BCF est observé, en raison de la présence d'opérations en virgule flottante dans ce programme (voir Fig. 4.6). Le cœur HP est également 40 % moins efficaces en énergie que le cœur BCF. En fait, le modèle quadricœur illustré à la Figure 4.1b, qui contient deux cœurs BCF, est le meilleur choix pour le repère KNN.

Pour le programme Stencil, qui contient également des calculs en virgule flottante, le cœur BC est, bien entendu, le moins efficace sur le plan énergétique. De même, pour le programme KNN, le cœur BCF est le meilleur, de près de 280 % que le cœur HP. Cette énorme différence réduit les avantages de la parallélisation du programme. En général, les configurations comprenant un ou deux cœurs BCF sont les plus efficaces sur le plan énergétique. La configuration avec deux cœurs BCF est la meilleure. Le cœur BC est extrêmement pénalisant, tandis que le cœur HP ne permet qu'une amélioration limitée, d'environ 3%, comme le montre

la Figure 4.6 en comparant la configuration de deux cœurs BCF et la configuration de 1 cœur HP et de deux cœurs BCF.

Les gains d'énergies que nous avons obtenus et présentés dans les Figures 4.5 et 4.6 traduisent à la fois une variation du temps d'exécution des programmes et de consommation de puissance de l'architecture. Ces variations sont influencées par les types de cœurs utilisés dans une configuration. Nous décrivons plus en détail ces variations du temps d'exécution des programmes et de la puissance consommée par les configurations associées dans le journal [118].

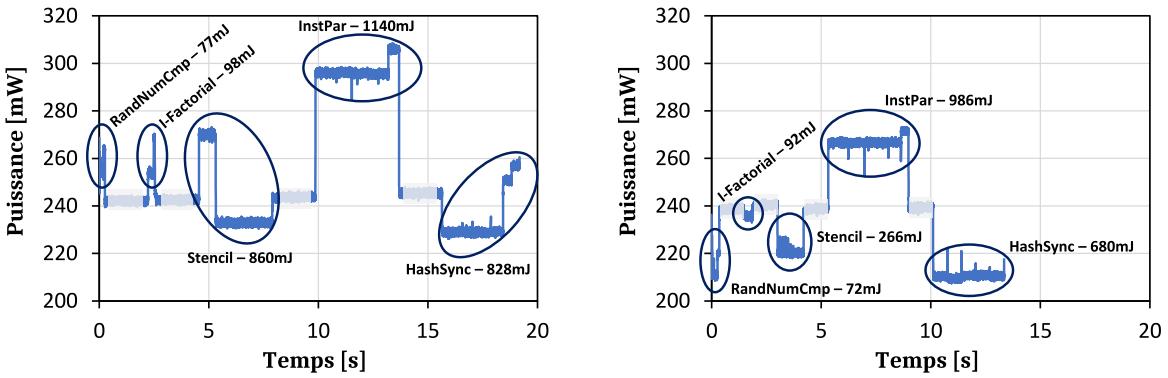
Mise en évidence de l'efficacité d'une utilisation adaptée des cœurs hétérogènes.

À partir de l'évaluation faite dans la section 4.4.2, des enseignements peuvent en être tirés. On a pu observer que tous les types cœurs n'offrent pas le même niveau de performance en fonction de la nature de l'algorithme exécuté. En effet, pour les programmes comme I-Matmul ou InstPar, des gains d'énergie sont observés en sélectionnant des configurations avec un nombre de cœurs plus élevé. Les configurations d'architecture incluant des cœurs BCF offrent la plus faible consommation d'énergie en présence de calculs en virgule flottante, comme c'est le cas pour les programmes F-Matmul, FFT et Mpeg. Il est intéressant de noter que la personnalisation du FPU implémentée dans le cœur BCF par la société Cortus est suffisamment puissante pour devenir un choix alternatif par rapport au cœur HP. Au vu de la consommation énergétique de ce dernier, son utilisation peut même être nettement handicapante pour les programmes liés à la mémoire tels que KNN et Stencil. Cependant, dans le cas du benchmark Bitonic, qui est également lié à la mémoire mais sans calculs en virgule flottante, l'augmentation du nombre de cœurs BC contribue à la minimisation de l'énergie, même sans l'aide du cœur HP. De plus, l'utilisation du cœur HP pour exécuter des programmes avec une forte présence de synchronisation s'avère pénalisante. Une remarque similaire concerne les charges de travail avec des taux élevés de mauvaise prédiction de branchement, comme illustré par le programme RandNumCmp. Le cœur HP devient trop coûteux en énergie en raison des opérations complexes effectuées dans sa microarchitecture lors des erreurs de prédiction de branchement. Plus généralement, les cœurs à faible consommation s'avèrent meilleurs dans les deux situations ci-dessus.

Les observations de la section précédente résultent d'expériences réalisées en utilisant l'ordonnancement des tâches par défaut de l'architecture. Cet ordonnancement consiste à attribuer des tâches à traiter à tous les cœurs disponibles. L'affinité des cœurs vis-à-vis de la

nature de l'algorithme n'est pas prise en compte. Forts des enseignements des expériences précédentes, nous pouvons assigner les cœurs aux tâches pour lesquelles ils sont les plus performants (ou en tout cas le moins pénalisant).

Pour ce faire, la charge de travail sera composée d'une succession de tâches différentes issues des algorithmes présentés dans le tableau 4.4.



(a) Répartition par défaut des tâches.

(b) Répartition guidée des tâches.

Figure 4.7 – Exécution du multi-programmes.

La Figure 4.7 illustre les consommations d'énergie respectives pour chaque sous-partie de la charge de travail, correspondant à un algorithme avec une nature d'algorithmique spécifique. Pour améliorer la lisibilité des résultats produits, nous avons inséré une tâche fictive (représentée par les parties bleu clair des graphiques des Figure 4.7a et Figure 4.7b) entre les différentes sous-parties de la charge de travail. Cette tâche fictive a une consommation d'énergie constante dans chaque scénario d'exécution.

Nous observons que le temps d'exécution global du programme synthétique avec la prise en compte de l'affinité entre les cœurs et la nature des algorithmes est réduit par rapport à la répartition par défaut. La configuration de cœur de l'architecture choisie pour la répartition guidée des tâches est indiquée dans le tableau 4.4. La valeur de la réduction du temps d'exécution s'élève à 26%. De même pour la puissance consommée, nous observons une diminution globale de 13% avec la prise en compte de l'affinité entre les cœurs et la nature des algorithmes est réduite par rapport à la répartition par défaut. En outre, pour chaque sous-partie (à l'exclusion des parties de la tâche fictive), la consommation d'énergie correspondante est annotée dans les figures.

De manière générale, l'adaptation de l'exécution des applications sur une architecture hétérogène comme celle proposée ici peut tirer parti d'analyses en-ligne pour décider de

Tableau 4.4 – Composition de la charge de travail synthétique.

Programmes	Configurations sélectionnées
RandNumCmp	1F 2I
I-Factorial	1H
Stencil	2F
InstPar	1H 1F 2I
HashSync	1F 1I

la meilleure configuration matérielle. Des travaux comme ceux abordés dans [130–134] promeuvent cette idée en exploitant notamment des techniques de machine learning.

4.5 Potentiel des architectures proposées pour le ML

4.5.1 Algorithmes de *machine learning* évalués

Après avoir testé les architectures multicœurs hétérogènes avec des programmes de nature différente, nous allons à présent utiliser des programmes de ML. Les programmes de ML présente la particularité d'être à la fois de nature *memory-intensive* et *computing-intensive*. C'est cette particularité qui rend leurs utilisations dans les systèmes embarqués aussi contraignantes. Pour les expérimentations qui vont suivre, nous avons considéré trois types d'algorithmes de *machine learning* résumés dans le tableau 4.5 : regroupement, classification et régression.

Tableau 4.5 – Algorithmes de *machine learning* utilisés

Programmes	Type	Flottant	Jeu de données
K-means	Regroupement	Oui	2000
LogReg	Classification	Oui	(2000, 80)
Backprop	Régression	Oui	(260, 59)

- **Algorithmes de regroupement:** K-means est un algorithme populaire pour l'analyse donnée non supervisée. À partir d'un ensemble de données non étiquetées (c'est-à-dire sans groupe ou catégorie prédéfinie), l'algorithme va procédé itérativement a un partitionnement des données en k *clusters* (c'est-à-dire des sous-ensembles de données) en minimisant la distance au carrée entres les données et un centre du *cluster* déduit. À chaque itération, le centre du *cluster* est ajusté afin d'en identifier le meilleur.

- **Algorithmes de classification:** L'algorithme de régression logistique (LogReg) permet la classification de donnée supervisée. Dans un premier temps, l'algorithme va analyser

des données étiquetées (c'est-à-dire donnée dont leurs classes est connu) communément appelées d'entraînement ou d'apprentissage. Dans un second temps, l'algorithme va procéder à la classification (ou étiquetage) de données non étiquetées, généralement nommée inférence. Le résultat de cette classification est une probabilité d'appartenance d'une donnée aux classes de données possible.

- **Algorithmes de régression:** Le dernier modèle ML considéré est un réseau de neurones artificiel (ANN) avec rétropropagation (Backprop), qui exploite la descente de gradient pour réaliser les tâches d'entraînement et d'inférence. Un processus itératif est appliqué pour ajuster les paramètres (par exemple, ses poids) de l'ANN jusqu'à ce que la précision de la tâche d'entraînement atteigne un seuil acceptable. Par souci de simplicité, nous considérons un réseau neuronal à trois couches : couches d'entrée, cachée et de sortie. Le réseau est composé de 30, 10 et 1 neurone dans ses couches d'entrée, cachée et de sortie respectivement. Son nombre total de poids est de 310. La taille de ce réseau est délibérément réduite en raison des ressources matérielles limitées disponibles sur la carte FPGA cible.

Pour réaliser nos expériences avec les algorithmes ci-dessus, nous considérons un ensemble de données de 2000 éléments pour l'algorithme de regroupement K-means. Pour les algorithmes de classification LogReg et régression Backprop, les données sont partitionnées en deux sous-ensembles (T, I), où le sous-ensemble T est utilisé pour l'entraînement du modèle, et le sous-ensemble I est utilisé pour l'inférence basée sur le modèle entraîné. Nous utilisons les partitionnements (260, 59) et (2000, 80) pour Backprop et LogReg respectivement.

Notez que pour chaque algorithme, nous avons considéré autant d'éléments de données que la mémoire disponible dans le prototype FPGA le permet. La taille du programme correspondant à Backprop est la plus grande parmi les trois, laissant ainsi moins d'espace pour le stockage des données d'entraînement et d'inférence.

4.5.2 Optimisations au niveau logiciel

Communément, la mise en œuvre de système embarqué est confrontée à des contraintes en termes de surface et de consommation d'énergie ce qui est particulièrement vrai dans le cas du prototype FPGA introduit précédemment dans la partie 4.3. Un programme écrit dans un langage de programmation tel que le langage C et C++ est exécuté sur un processeur après avoir été compilé. La compilation permet la création d'un exécutable qui est une

traduction d'un langage de haut niveau vers le langage machine. Le compilateur ne se contente pas de simplement traduire le programme donné, il va réaliser une succession d'opérations diverses telle que preprocessing, analyse lexicale, analyse syntaxique, analyse sémantique et des optimisations. Ces optimisations ont un impact significatif sur l'exécutable résultant affectant aussi bien la taille de l'exécutable que le temps d'exécution ou bien l'empreinte mémoire. Dans un contexte de système embarqué, il est donc pertinent d'évaluer l'impact des optimisations du compilateur GNU Compiler Collection (GCC) [135] sur les algorithmes de ML présentés précédemment dans la partie 4.5.1.

Le rôle des options d'optimisations

Le compilateur GCC donne accès à plusieurs optimisations de code possible qui peut être individuellement sélectionné par l'utilisateur. Cependant, le compilateur a défini 7 niveaux d'optimisation possible: -O0, -O1, -O2, -O3, -Ofast, -Os et -Og [136].

Le niveau d'optimisation de base -O0 de GCC n'est pas pris en compte lors des expérimentations. C'est le niveau d'optimisation par défaut de GCC qui a la particularité de ne pas activer d'optimisation pour permettre une compilation rapide. L'objectif de nos expérimentations est de justement évaluer les optimisations permettant des améliorations de performance lors de l'exécution de programme dans un cadre de système embarqué. Pour la même raison, le niveau d'optimisation -Og n'est pas pris en compte dans nos expérimentations, car il est essentiellement destiné au débogage. Ci-dessous, les niveaux d'optimisation utilisés lors des expérimentations sont succinctement décrits.

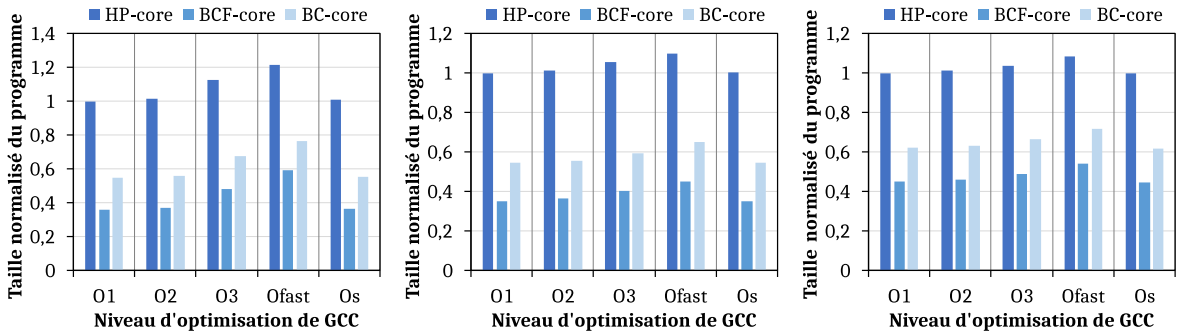
- **-O1**: Ce niveau d'optimisation vise à réduire le temps d'exécution et la taille du code. Il s'agit de l'option d'optimisation la plus légère.
- **-O2**: Ce niveau augmente le nombre d'optimisations effectuées par rapport à l'option précédente. Visé a augmenter les performances sans contrainte de taille de code. Souvent considérée comme l'une des options les plus sûres, il est généralement recommandé de l'envisager en premiers.
- **-O3**: C'est le plus haut niveau d'optimisation contenant des optimisations des niveau -O1 et -O2 ainsi que les optimisations les plus agressives pour le code
- **-Ofast**: Il s'agit d'une option alternative qui active toutes les optimisations -O3. De plus, elle applique des optimisations qui ne sont pas valables pour tous les programmes

conformes aux standards. Il réduit agressivement le temps d'exécution, mais peut altérer les résultats et la reproductibilité du comportement du programme.

- **-Os**: Cet optimiseur vise principalement à minimiser la taille du code reprenant des optimisations de O2 sans celle pouvant entraîner une augmentation de la taille du code. Il est particulièrement intéressant pour les systèmes disposant de ressources mémoires limitées comme les FPGA.

Impact des optimisations du compilateur

L'évaluation de l'impact des optimisations de GCC précédemment présenté est réalisée avec les algorithmes de ML K-means, LogReg et Backprop introduit dans la section 4.5.1. Dans les résultats ci-dessous nous comparons l'impact des optimisations de GCC respectif sur la taille du code pour chaque type de cœur dans l'architecture heptacœur, afin d'économiser les ressources FPGA. De plus, nous exécutons les algorithmes de ML sur les différents types de cœurs pour évaluer les résultats énergétiques correspondants. Ces évaluations sont présentées dans les figures 4.8 et 4.9.



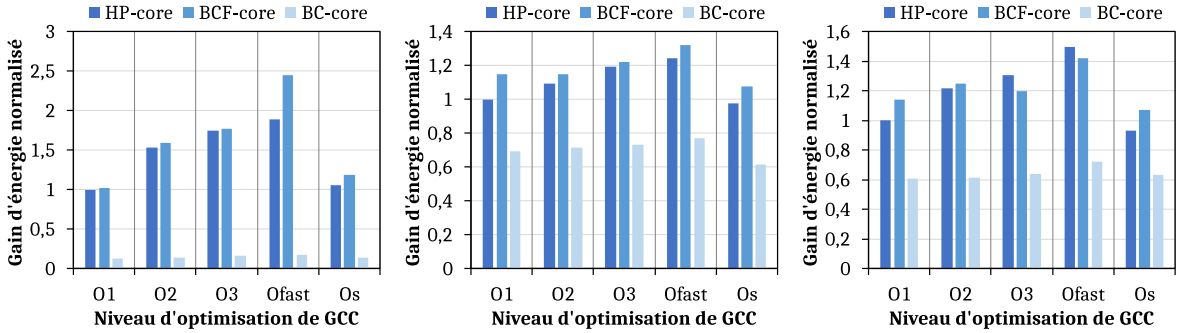
(a) Backprop. (b) LogReg. (c) K-means.

Figure 4.8 – Les tailles des programmes obtenues pour différentes options d'optimisations du compilateur, pour chaque type de cœur.

En excluant l'option -Os, nous observons que le passage à des niveaux d'optimisation plus agressive entraîne une augmentation de la taille du code (voir figure 4.8). Cependant, il est surprenant de constater que l'option d'optimisation -Os ne permet pas d'obtenir une taille de code significativement inférieure à celle des niveaux d'optimisation -O1 et -O2.

En ce qui concerne la consommation d'énergie, les options -O1 et -Os n'apportent aucun gain par rapport aux options -O3 et -Ofast (voir Fig. 4.9). Cette dernière option, c'est-à-dire

-Ofast, présente même un gain d'environ 135 % par rapport à -O1 dans le cas de l'algorithme Backprop exécuté par le cœur BCF.



(a) Backprop. (b) LogReg. (c) K-means.

Figure 4.9 – Les gains d'énergies obtenues lors de l'exécution des programmes compilés avec différentes options d'optimisations du compilateur, pour chaque type de cœur.

Ici, la consommation d'énergie rapportée couvre à la fois les phases d'entraînement et d'inférence. Plus globalement, il apparaît que -O1 et -Os n'offrent pas une efficacité énergétique suffisante au regard de la réduction de la taille du code qu'ils permettent. De plus, -Ofast a l'inconvénient de conduire à des résultats de performance difficilement reproductibles, ce qui rendrait la comparaison un peu difficile dans nos expériences ultérieures. Par conséquent, pour le reste de nos expériences, nous ne conserverons que l'option d'optimisation -O3. En nous concentrant sur les scénarios de l'option -O3 dans la Fig. 4.9, nous observons que le cœur HP et le cœur BCF surpassent de loin le cœur BC en termes de consommation d'énergie. Cela est dû en grande partie à l'absence de support matériel de l'unité à virgule flottante dans le cœur BC, ce qui nuit fortement au temps d'exécution des algorithmes considérés. Les gains énergétiques apportés par le cœur BCF par rapport au cœur HP sont respectivement de +5%, +8.5% et -2% pour Backprop, LogReg et K-means respectivement. Un premier constat intéressant est que le microcontrôleur cœur BCF représente a priori une alternative très pertinente au processeur applicatif cœur HP du point de vue énergétique. De plus, des gains potentiels supplémentaires pourraient être attendus du cœur BCF en ce qui concerne la consommation statique d'énergie puisque sa surface est 15 fois plus petite que celle du cœur HP.

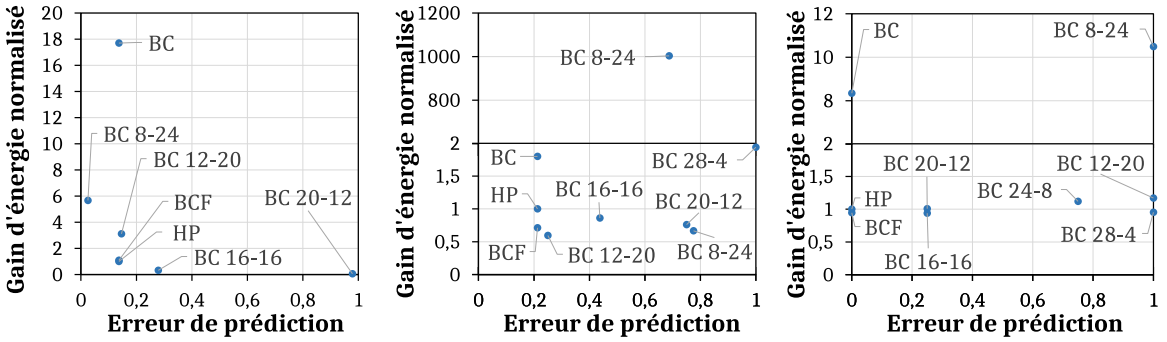
4.5.3 Optimisations au niveau matériel

L'architecture présentée dans la partie 4.3.1 est basée sur l'association des caractéristiques intrinsèques de trois types de cœurs pour l'exécution d'algorithme de ML. On a pu voir dans la partie 4.4.2 que les trois types de cœurs affichaient des performances différentes en fonction de la nature de l'algorithme en cours d'exécution.Nous allons à présent nous intéresser à deux cas de figure pour utiliser aux mieux l'architecture multicœur hétérogène.Dans une première partie, nous allons voir comment compenser l'absence de FPU et dans un second temps comment exploiter le parallélisme pour un algorithme aux départs séquentiels comme K-means.

Compensation du manque d'unité de calcul en virgule flottante

L'absence de FPU dans les cœurs BC pour traiter les calculs en virgule flottante est un inconvénient évident par rapport aux cœurs HP et BCF. Pour pallier cette limitation, nous envisageons une représentation des nombres en virgule fixe pour l'exécution des programmes sur les cœurs BC. Une telle représentation des nombres réels définit un nombre fixe de chiffres après la virgule, induisant ainsi une réduction de la précision arithmétique par rapport à la représentation en virgule flottante. Nous avons utilisée une version à virgule fixe pour les trois algorithmes ML (Backprop, LogReg et K-means) avec des représentations en virgule fixe de 32 bits. Dans nos expériences, six instances de représentation en virgule fixe sont évaluées sur le cœur BC. Elles sont désignées par "BC X-Y", où X représente le nombre de chiffres avant la virgule et Y représente le nombre de chiffres après la virgule (c'est-à-dire X + Y = 32). En d'autres termes, la partie entière des unités arithmétiques à virgule fixe de 32 bits est composée de X chiffres, tandis que la partie fractionnaire est composée de Y chiffres. Par exemple, dans l'instance en virgule fixe désignée par BC 8-24, les 8 premiers chiffres sont dédiés à la partie entière du nombre réel, tandis que les 24 chiffres restants sont dédiés à la partie décimale. Ensuite, nous évaluons les instances à virgule fixe suivantes : BC 8-24, BC 12-20, BC 16-16, BC 20-12, BC 24-8 et BC 28-4. Chacune de ces instances est exécutée sur un seul cœur BC, et comparée aux variants à virgule flottante des algorithmes ML sur les cœurs BC, BCF et HP.

La figure 4.10 résume cette comparaison en termes de compromis entre énergie et erreur de prédiction : plus l'énergie est faible (c'est-à-dire proche de 0), meilleure est l'instance évaluée ; et plus l'erreur de prédiction est faible (c'est-à-dire proche de 0), plus l'instance est précise. Notez que la consommation d'énergie rapportée est normalisée par rapport à celle du cœur HP. Dans les figures 4.10b et 4.10c, nous avons ajouté un zoom sur certaines régions des graphiques afin de mieux mettre en évidence la différence entre les scénarios concernés.



(a) Backprop. (b) LogReg. (c) K-means.

Figure 4.10 – Évaluation de l'énergie normalisée et de la précision : précision en virgule flottante sur les cœurs HP et BCF contre précision en virgule fixe sur les cœurs BC.

Tout d'abord, par rapport aux variants d'algorithmes à virgule flottante exécutée sur le cœur BC, les variants à virgule fixe réduisent globalement la consommation d'énergie d'au moins 200 %, au prix d'une perte de précision d'entrainement acceptable, c'est-à-dire moins de 20 % de différence d'erreur. Elles présentent même des améliorations énergétiques par rapport aux variants d'algorithmes à virgule flottante exécutée sur les cœurs HP et BCF, qui incluent un FPU dans leur microarchitecture. Pour Backprop et LogReg, la réduction d'énergie atteint 20%. Les meilleurs candidats pour la précision en virgule fixe varient d'un algorithme à l'autre. Par exemple, pour LogReg, BC 12-20 apparaît comme le plus avantageux, tandis que pour l'ANN BC 16-16 est la meilleure instance en virgule fixe. Seul l'algorithme K-means ne présente pas d'amélioration énergétique, car il manipule moins d'arithmétique en virgule flottante. La précision en virgule fixe représente donc une alternative pertinente au manque de support FPU dans les cœurs BC. En outre, les FPGA sont connus pour leur efficacité énergétique traditionnelle pour les calculs en virgule fixe [137] (même si les FPGA de pointe telle que le Stratix 10 2 d'Altera/Intel intègre désormais des FPU compétitifs). Nous notons que dans toutes les expériences ci-dessus, l'estimation de la consommation d'énergie du système FPGA considéré ne varie que de 1,3W à 1,5W, où la partie statique représente environ 80%.

Les tableaux 4.6 et 4.7 résume certains scores de performance clé obtenus lors de ces expériences. Il rapporte exclusivement les meilleurs scores d'inférence obtenus par chaque type de cœur sur les algorithmes Backprop et LogReg. Les métriques présentées indiquent le nombre d'inférences (ou de prédictions) par seconde et par joule. Il est intéressant de noter l'amélioration du score fournie par la précision en virgule fixe combinée à l'optimisation du débit permise par l'exécution parallèle sur trois cœurs BC à faible puissance.

Tableau 4.6 – Performances en matière d'inférence des cœurs Cortus.

Configurations matérielles	cœur HP		cœur BCF		cœur BC	
Algorithmes	Backprop	LogReg	Backprop	LogReg	Backprop	LogReg
Meilleure perf. (#inferences/sec)	161.64	41.38	149.36	46.83	0.47	2.87
Meilleure energie-eff. (#inferences/J)	806.20	173.16	865.14	275.03	2.53	16.03

Tableau 4.7 – Performances en matière d'inférence des cœurs Cortus BC, en virgule fixe.

Configurations matérielles	cœur BC (virgule fixe)		3x cœurs BC (virgule fixe)	
Algorithmes	Backprop	LogReg	Backprop	LogReg
Meilleure perf. (#inferences/sec)	842.85	47.50	**2023.56**	**169.13**
Meilleure energie-eff. (#inferences/J)	4617.13	254.86	**10749.33**	**891.77**

Parallélisation de l'implantations de K-means

L'intérêt d'une architecture multicœur hétérogène peut rapidement être perdu si les algorithmes utilisés n'exploitent pas pleinement la capacité de parallélisation de l'architecture. Pour répondre à cette difficulté, nous envisageons maintenant la parallélisation de l'algorithme K-means pour accélérer son exécution.

L'idée est de diviser la charge de travail en plusieurs morceaux distincts et d'appliquer sur chaque morceau l'algorithme des K-means. De cette façon, chaque cœur utilisé ne traitera qu'une partie du jeu de données initial. Une fois que chaque algorithme K-means a terminé l'exécution de son morceau correspondant, nous obtenons les centroïdes des différents morceaux. Ensuite, nous appliquons à nouveau l'algorithme K-means sur les centroïdes obtenus pour produire le regroupement final autour des centroïdes finaux. Comme indiqué précédemment, nous considérons un jeu de données de 2000 éléments de données, divisé et distribué sur les quatre cœurs supportant le calcul en virgule flottante : HP et 3 BCF. Il s'agit d'un choix arbitraire à des fins d'illustration. Chaque cœur exécute K-means sur un chunk et produit 4 centroïdes. Nous obtenons donc 16 centroïdes au total qui sont ensuite traités par le dernier algorithme K-means pour déduire les quatre centroïdes finaux.

La figure 4.11 illustre l'ensemble du jeu de données selon sa distribution en *clusters* distincts, lors de l'application des versions séquentielle et parallèle de K-means. Les partitions C1, C2, C3 et C4 représentent les morceaux considérés pour l'exécution parallèle. Au centre

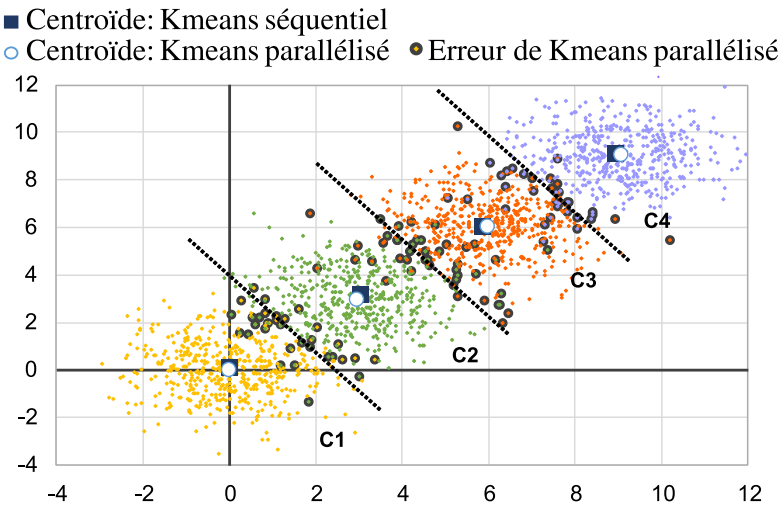


Figure 4.11 – Visualisation des données et des centroïdes.

de chaque *cluster* de points sont représentés les deux centroïdes déterminés par les deux versions de K-means. Les carrés pleins et les cercles vides indiquent les centroïdes finaux obtenus par les versions séquentielle et parallèle de K-means. On observe la similarité entre les centroïdes obtenus avec les deux versions. Cependant, les points représentés par des cercles gris apparaissant à la frontière d'une paire de *clusters*, correspondent aux points qui ne sont pas classés dans le même *cluster* par les deux versions. Cette différence entre les deux versions est d'environ 5%.

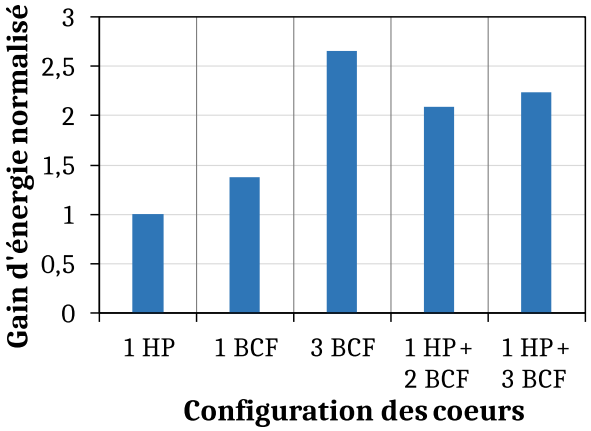


Figure 4.12 – Les gains d'énergies normalisées pour la version parallélisée de K-means.

La consommation d'énergie correspondante est reportée dans la Fig. 4.12. On peut observer que la parallélisation permet un gain énergétique notable par rapport à la version séquentielle

exécutée sur des cœurs uniques. Ce gain provient principalement de la division par deux du temps d'exécution obtenu avec la version parallèle de K-means sur trois cœurs BCF.

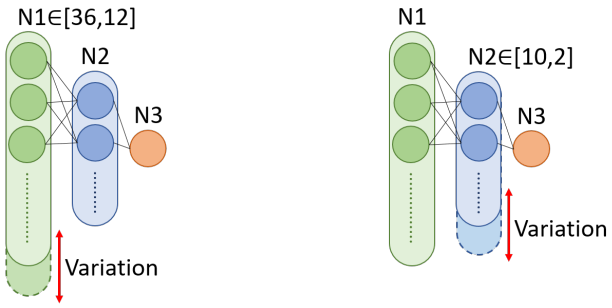
4.5.4 Exploration de paramètres de modèles

L'exécution d'algorithmes de ML tel que les réseaux de neurones sont sensibles a de multiples paramètres. Ces paramètres influencent aussi bien le temps d'exécution, la mémoire nécessaire et l'énergie consommée que l'exactitude des résultats du réseau de neurones. Nous allons donc voir l'impact du nombre de neurones sur un réseau, et visualiser les variations des erreurs obtenues et de l'impact du taux d'apprentissage sur la consommation.

Paramètre d'architecture

Parmi les approches existantes pour optimiser les ANNs, on peut citer l'utilisation de l'algorithme d'évaluation des hyper-paramètres [138] dont le principe général est de réaliser une exploration exhaustive des paramètres du réseau. Mais cette exploration exhaustive nécessite souvent beaucoup de temps et d'énergie. Ici, nous considérons l'exploration des paramètres du réseau ANN en nous concentrant sur le dimensionnement de la taille du réseau basé sur le nombre de neurones par couche (voir Fig.4.13). Nous partons d'une configuration de réseau arbitraire composée de 30, 10 et 1 neurones dans ses première, deuxième et dernière couches respectivement. Puis, par souci de simplicité, nous faisons varier séparément le nombre de neurones dans la première et la deuxième couche, comme le montre la figure 4.13, et nous explorons les améliorations énergétiques possibles tout en préservant une précision de prédiction acceptable. En fin de compte, le nombre de neurones dans ces deux couches conduisant au meilleur résultat en termes d'énergie et d'erreur quadratique moyenne normalisée (NMSE) sera sélectionné comme la meilleure configuration d'un bon réseau candidat.

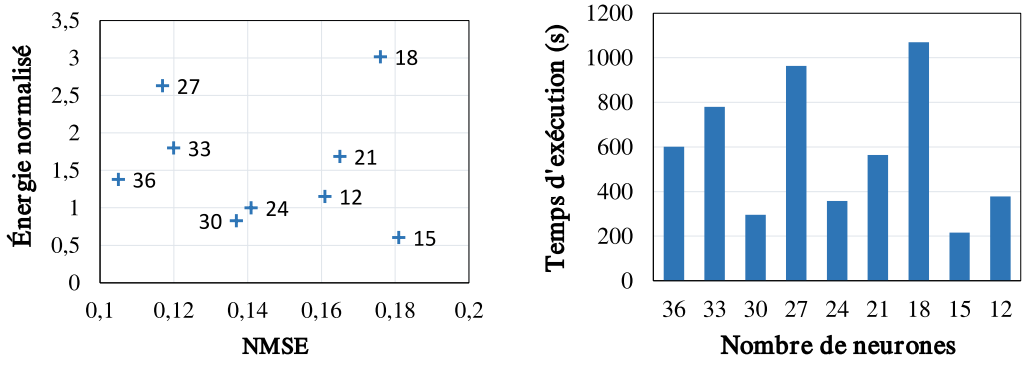
Cette exploration est menée sur le prototype FPGA. Nous profitons de la nature multicœur de son architecture embarquée pour paralléliser cette exploration. Chaque cœur effectue une tâche d'entraînement avec une configuration de réseau différente, et le résultat obtenu est comparé à celui des autres cœurs pour identifier la meilleure configuration de réseau dans l'espace exploré. Les cœurs HP et BCF sont utilisés pour cette exploration, car nous considérons l'algorithme Backprop, qui contient des calculs lourds en virgule flottante. Les résultats de cette exploration sur les première et deuxième couches sont présentés respectivement dans les figures 4.14 et 4.15. Les chiffres mis en évidence dans les graphiques représentés sur les figures 4.14a et 4.15a correspondent aux nombres évalués de neurones dans les première et deuxième



(a) Couche 1 (b) Couche 2

Figure 4.13 – Variation du nombre de neurones dans les couches du réseau de neurones.

couches respectivement. D'après ces résultats, la configuration du réseau avec 15 neurones et 2 neurones sur la première et deuxième couches fournit la meilleure efficacité énergétique, avec une précision raisonnable, c'est-à-dire un NMSE inférieur à 0,2. Après l'exécution de ce réseau, le gain énergétique est d'environ x3,5 par rapport au réseau neuronal initial. Le temps total d'exploration des dix-huit réseaux sur l'architecture est d'environ une heure sur le FPGA. Il s'agit d'une complexité d'exploration raisonnable pour les nœuds périphériques.

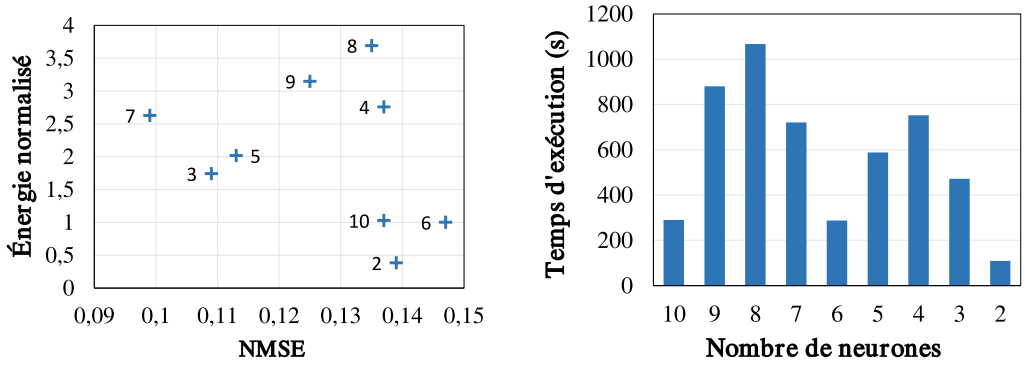


(a) Énergie normalisée et précision. (b) Temps d'exécution.

Figure 4.14 – Évaluation du réseau en fonction de la variation du nombre de neurones de la couche 1.

Paramètre de taux d'apprentissage

Nous explorons le compromis entre la précision et l'efficacité énergétique de LogReg en faisant varier son paramètre de taux d'apprentissage. Ce paramètre influence directement la précision de l'apprentissage. Il est généralement compris entre 0 et 1. Une valeur proche de 0 suggère intuitivement un apprentissage plus précis. Cependant, cette précision induit souvent



(a) Énergie normalisée et précision.

(b) Temps d'exécution.

Figure 4.15 – Évaluation du réseau en fonction de la variation du nombre de neurones de la couche 2.

une surcharge en temps d'exécution et en dissipation d'énergie. Nous avons donc exploré différentes valeurs de taux d'apprentissage afin de déterminer le meilleur compromis entre énergie et précision d'apprentissage, comme le montre la figure 4.16.

Comme prévu, l'augmentation du taux d'apprentissage entraîne une perte de précision, mais aussi un gain en énergie. Le meilleur compromis est avec un taux d'apprentissage de 0.2, qui permet d'obtenir un gain d'énergie de 15% par rapport au taux d'apprentissage le plus précis (i.e. 0.1), avec une perte de précision de 10%. Dans l'exploration ci-dessus, nous notons que le temps d'évaluation de chaque valeur de taux d'apprentissage ne nécessite pas plus de 3 minutes. Ceci peut être considéré comme une complexité d'exploration raisonnable.

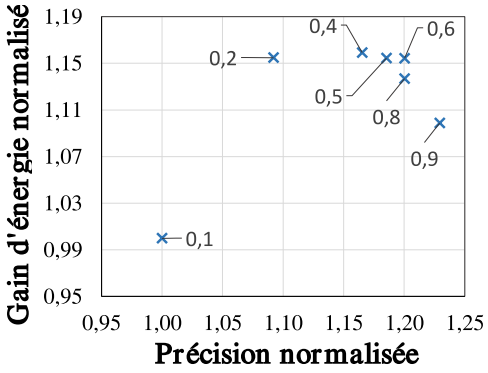


Figure 4.16 – Précision en fonction de l'énergie pour différents taux d'apprentissage.

4.6 Résumé

Ce chapitre a été consacré à l'étude d'une architecture multicœur hétérogène efficace en énergie. L'hétérogénéité de cette architecture est caractérisée par la présence de trois types de cœur différents : cœur de haute performance, cœur de basses consommations et un cœur de basse consommation supportant les opérations flottantes. L'évaluation de la consommation et des performances de ce prototype d'architecture est permise grâce à une implantation sur carte FPGA.

Tout d'abord, douze algorithmes de nature différente ont été évalués sur l'architecture multicœur hétérogène. Cette évaluation permet la mise en avant de l'intérêt de considérer une architecture de calcul de cœurs différents. En effet, chaque type de cœurs montre des performances différentes en fonction de la nature de la charge de travail. Ces résultats encourageants ont montré l'intérêt d'utiliser cette architecture multicœur hétérogène dans le domaine de l'informatique en périphérie. La première exécution a utilisé l'intégralité des cœurs disponibles dans l'architecture, sans la prise en compte des types de cœurs disponibles et la deuxième exécution à utiliser exclusivement les types de cœurs les plus efficaces pour chaque algorithme composant l'application synthétique. Cette deuxième exécution a permis une réduction de 26% du temps d'exécution et de 13% de la puissance consommée. Ces résultats encourageants ont montré l'intérêt d'utiliser cette architecture multicœur hétérogène dans le domaine de l'informatique en périphérie.

Dans un second temps, l'évaluation de l'architecture proposée s'est orientée sur l'utilisation d'algorithmes typiques de ML. Au vu des contraintes entourant l'utilisation les algorithmes de ML, nous avons étudié l'influence de différentes d'optimisation sur leurs exécutions sur l'architecture. Si le caractère généraliste de cette dernière permet de traiter une grande variété d'algorithmes, il ne permet pas cependant d'adresser les spécificités de réseaux de neurones courants telles que les CNN. Dans le chapitre suivant, nous abordons une approche architecturale complémentaire, reposant sur l'accélération de traitements spécifiques à des réseaux de type CNN.

Chapitre 5

Unité de calcul flexible dédiée aux algorithmes d'apprentissage profond

Table des matières du chapitre

Ce chapitre est destiné à présenter la conception d'une unité multiplication-accumulation permettant des multiplications asymétriques pour des tailles d'opérandes en puissance de 2, à partir de 2 bits jusqu'à 32 bits [139].

Le chapitre est organisé essentiellement en deux parties, la première partie présente la conception de l'unité MAC et une deuxième partie concernant son évaluation.

5.1 Motivations: importance des unités MAC dans les algorithmes de ML

Dans la section 2.3, nous avons introduit l'importance des opérations MAC dans les techniques de machine learning et plus particulièrement dans les DNN. À présent, nous allons rentrer plus dans les détails de ce type d'opération et de son implantation matérielle.

Une opération MAC est couramment utilisée en informatique, et elle suit l'équation suivante

$$C = A \times B + C \tag{5.1}$$

Cette opération est la succession d'une multiplication et d'une addition. Bien qu'elle puisse paraître anodine, l'opération MAC est fréquente. L'un des algorithmes les plus emblématiques est la multiplication de matrice. La réalisation d'une multiplication de matrice est largement composée d'opérations MAC. L'illustration en Figure 5.3 montre deux matrices A et B qui sont multipliées donnant la matrice résultante C.

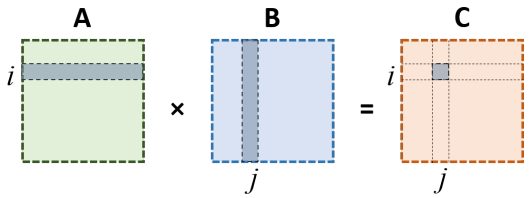


Figure 5.1 – Illustration de la multiplication de matrice.

Le calcul de chaque cellule de la matrice résultante est défini par l'équation 5.2. Comme le montre l'équation, le résultat d'une cellule nécessité de multiple multiplication dont les résultats seront additionnés entre eux. Le nombre d'opérations de multiplication est de n^3 pour une matrice carrée de largeur n. Par exemple, avec une multiplication de deux matrices carrées de 4, il faudrait 64 multiplications et autant d'additions. Pour obtenir le résultat de la multiplication de deux matrices carrées de 4, il faut 1024 opérations distinctes.

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj} \tag{5.2}$$

La Figure 5.2 illustre le principe de la réalisation de cette opération dans un processeur. Deux données à calculer stockées dans leurs registres respectifs vont être multipliées et le résultat obtenu sera additionné à une donnée temporaire stockée dans un troisième registre

permettant l'accumulation. Au départ, cette donnée temporaire est initialisée à zéro. Le nombre de cycles nécessaire à la réalisation de ce type d'opération dépend en partie de l'implantation matérielle. Dans le cas de la figure 5.2, le résultat de la multiplication peut directement être additionné. Un seul cycle peut alors être suffisant. Mais dans certains cas, le résultat de la multiplication devra au préalable être stocké dans un registre avant de pouvoir être additionné. Par ce stockage intermédiaire, le nombre de cycles pour réaliser l'opération MAC est augmenté.

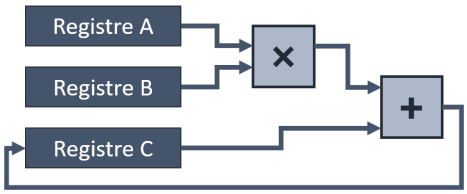


Figure 5.2 – Illustration du fonctionnement d'une opération MAC.

De manière générale, les algorithmes de ML sont composés de nombreuses opérations MAC. C'est particulièrement le cas pour les algorithmes de DNN qui se comportent comme des multiplications de matrice. Dans le cas d'un exemple concret, un CNN tel que AlexNet [20] contient 724M opérations MAC par inférence.

Une partie des enjeux de l'utilisation de DNN par un système de calcul réside dans la capacité à efficacement exécuter des opérations MAC. Comme expliqué plus haut, un cœur de processeur peut généralement exécuter 1 MAC / cycle, lorsque des DNN peuvent avoir 724M MAC comme AlexNet l'exécution devient irréalisable.

Par conséquent des *Graphics Processing Unit* et des Accelérateuurs dédiés sont fréquemment utilisés pour l'exécution de DNN. Ces deux types de matériels présentent de fortes capacités de parallélisation. Du côté des processeurs, c'est par l'ajout de systèmes *Single Instruction Multiple Data* que leurs capacités de parallélisation peut-être accrues. Généralement le fonctionnement des systèmes SIMD s'appuie sur une réduction de la précision des opérandes. Par exemple dans le cas des cœurs Cortex-M de ARM, le système SIMD permet la réalisation d'opération pour des données sur 16 bits et 8 bits [51]. Cependant, la mise en place de systèmes SIMD entraine un cout de surface et au niveau de l'*Instruction Set Architecture* (ISA).

5.2 Notre proposition : une unité MAC flexible

Face au besoin d'exécuter des algorithmes de ML dans les systèmes embarqués et aux contraintes inhérentes aux systèmes embarqués, nous proposons une nouvelle unité MAC destinée à être intégrée dans un microprocesseur 32 bits. L'objectif de cette unité MAC est de gérer les opérations de multiplication et les opérations MAC pour des données allant de 2 bits jusqu'à 32 bits en un cycle d'horloge. La particularité de cette unité MAC réside dans sa capacité à paralléliser la réalisation d'opérations de multiplication et MAC en fonction de la précision des données. La présentation de l'unité MAC est scindée en deux parties : la première est dédiée aux explications sur le principe de la multiplication binaire considérée et la seconde dédiée à la description de l'implantation de l'unité MAC.

5.2.1 Principe de la décomposition de la multiplication binaire

La réalisation de l'unité MAC décrite dans ce chapitre s'appuie la possibilité de décomposer la réalisation d'une multiplication binaire en plusieurs multiplications indépendantes de 2 bits. Pour donner une idée avec une multiplication sur 16 bits, la multiplication serait décomposable en 64 multiplications indépendantes de 2-bit. Pour illustrer la décomposition binaire, nous allons présenter un exemple sur une multiplication de deux opérandes de 4 bits.

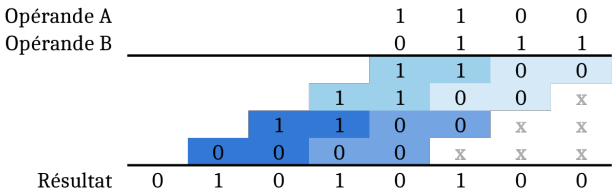


Figure 5.3 – Illustration de la multiplication binaire 4 bits.

La Figure 5.3 illustre la réalisation d'une multiplication binaire, elle est composée de deux opérandes A et B de 4 bits : le premier opérande est multiplié par chaque bit du second opérande. Les produits partiels résultant de cette multiplication sont décalés de manière appropriée avant d'être finalement additionnés.

En utilisant la décomposition de la multiplication binaire de 4bits, on obtient 4 multiplications de deux bits indépendantes. La Figure 5.4 illustre cette décomposition.

Le résultat final est obtenu en ajoutant, avec un décalage approprié, les quatre résultats des multiplieurs 2 bits × 2 bits, comme le montre la figure 5.5.

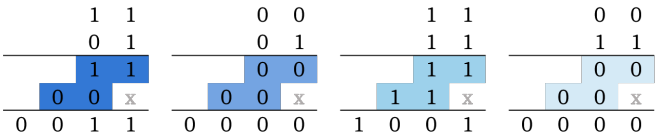


Figure 5.4 – Illustration de la décomposition de la multiplication binaire 4 bits.

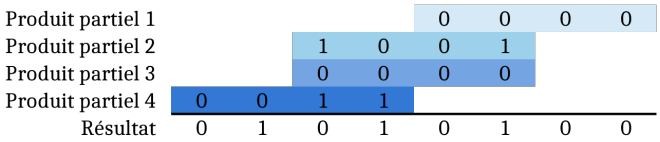


Figure 5.5 – Illustration de l'addition finale de la multiplication binaire 4 bits décomposée.

Implantation matérielle

Communément, l'opération de multiplication bit à bit est matériellement implantée à l'aide de portes ET. Ensuite, les résultats de cette opération sont additionnés par des additionneurs demi/complet pour obtenir le résultat final. La Figure 5.6 montre le circuit courant d'un multiplieur 2 bits.

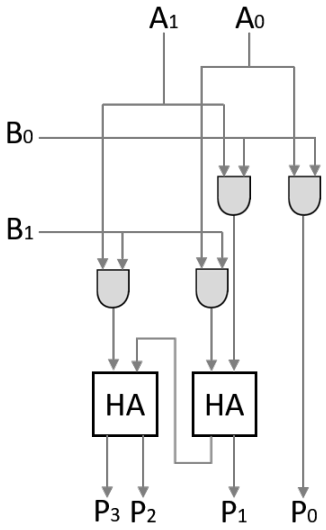


Figure 5.6 – Schéma d'un multiplieur 2 bits.

L'utilisation de l'approche décomposée est implantée matériellement suivant la Figure 5.7. Les quatre multiplieurs 2 bits indépendants mentionnés plus tôt sont bien identifiables. Leurs entrées sont alimentées par les deux opérandes A et B où seulement les 2 bits nécessaires sont récupérés. Comme le montre la Figure 5.5, trois décalages binaires sont nécessaires deux de 2

bits et un de 4 bits. Les quatre produits partiels peuvent ainsi être additionnés pour obtenir le résultat de la multiplication 4 bits.

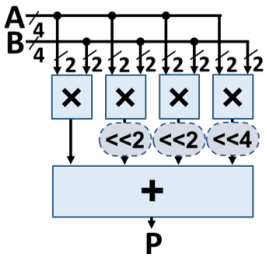


Figure 5.7 – Schéma d'un multiplieur 4 bits utilisant la décomposition binaire.

5.2.2 Description de l'unité MAC

Dans la section précédente, nous avons explicité le principe de la décomposition binaire appliqué pour des opérandes sur 4 bits. Dans cette section, la décomposition binaire sera mise en application pour un multiplieur de 32 bits. De plus, l'objectif est d'obtenir une unité MAC capable de réaliser en parallèle des opérations de multiplication et de MAC pour opérande de différente précision. Le point-clé permettant la parallélisation est illustré sur le Tableau 5.1. Les données en entrées de l'unité MAC sont préalablement stockées dans des registres de 32 bits. Ces registres peuvent être chargés avec des opérandes de 2 bits × 16 bits, ou de 4 bits × 8 bits, ou de 8 bits × 4 bits, ou de 16 bits × 2 bits comme indiqué dans le Tableau 5.1.

Tableau 5.1 – Représentation des données contenues dans un registre de 32 bits pour chaque largeur de bit de données.

32 bits															
16 bits								16 bits							
8 bits				8 bits				8 bits				8 bits			
4 bits		4 bits		4 bits		4 bits		4 bits		4 bits		4 bits		4 bits	
2b	2b	2b	2b	2b	2b	2b	2b	2b	2b	2b	2b	2b	2b	2b	2b

L'architecture de l'unité MAC proposée est composée de trois parties distinctes : multiplieurs, additionneurs et accumulateur. La figure 5.8 montre un schéma simplifié de cette architecture : une ligne de multiplieurs 2 bits dans la partie supérieure, les additionneurs et les décaleurs au milieu, le contrôle du décalage binaire à gauche, le multiplexeur de sortie à droite et l'accumulateur en bas. Pour supporter une multiplication de 32 bits en 1 cycle d'horloge, notre unité MAC est composée de 256 multiplieurs indépendants de 2 bits. En effet, dans la

Section 5.2.1 nous avons expliqué qu'une multiplication sur 4 bits nécessite 4 multiplieurs indépendants sur 2 bits $((4/2)^2)$. Par conséquent, pour une multiplication de 32 bits, 256 multiplieurs indépendants de 2 bits sont nécessaires $((32/2)^2)$.

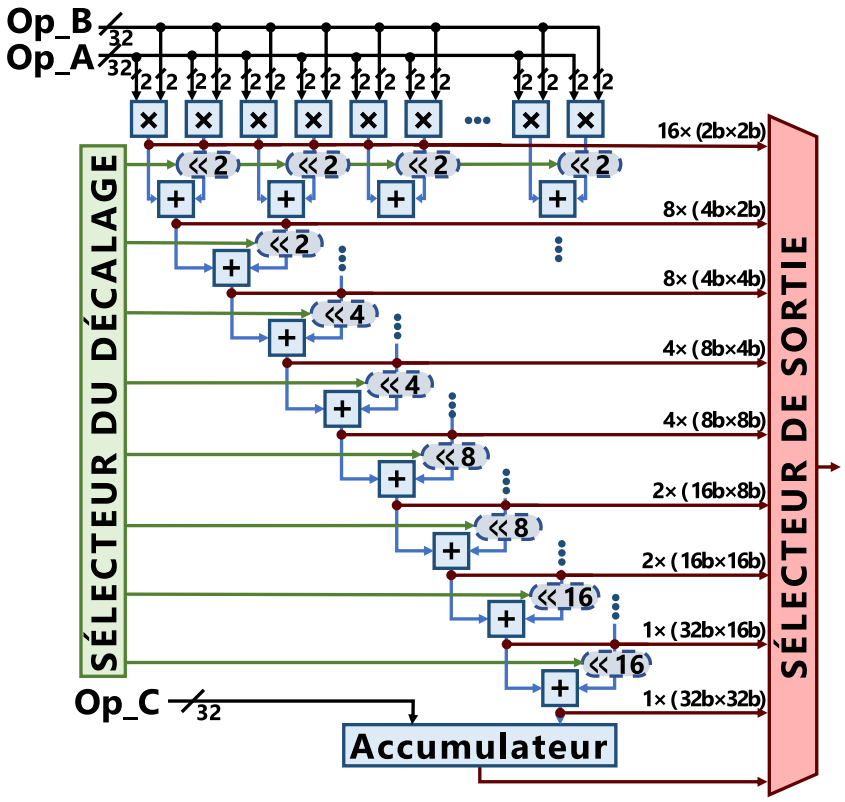


Figure 5.8 – Représentation schématique du multiplieur.

La connexion entre l'entrée des multiplieurs 2 bits et les registres d'entrée est fixe. La configurabilité de l'unité MAC par rapport à la précision des données est réalisée dans le reste du circuit. Pour éviter une consommation dynamique inutile des multiplieurs qui ne sont pas concernés par une opération et un blocage des données entrantes est présent. La sortie de 16 des multiplieurs est reliée au multiplexeur de sortie dans les cas où les données à calculer sont sur 2 bits.

Pour faciliter la configurabilité et l'adaptabilité des multiplieurs, l'addition des produits partiels est effectuée par des additionneurs à 2 entrées. Comme mentionné dans la section 5.2.1, pendant l'étape d'addition des produits partiels, un décalage de bits doit être effectué. Pour réduire le nombre de décalages, pour chaque additionneur, une de ces entrées est décalée au préalable. Comme le montre la figure 5.8, un maximum de 8 niveaux d'additionneurs est

nécessaire pour effectuer les multiplications. Les résultats des multiplications sont capturés aux sorties des multiplieurs 2 bits et de chaque niveau d'additionneur.

Enfin, l'accumulateur est utilisé pour effectuer l'opération MAC. Un registre dédié stocke la valeur intermédiaire de l'accumulation précédente. La valeur intermédiaire est renvoyée à l'accumulateur via l'Op_C illustré sur la figure 5.8.

Le Tableau 5.2 résume le nombre des opérations réalisable en parallèle en fonction des précisions des données d'entrée.

Tableau 5.2 – Le nombre d'opérations disponibles pour chaque largeur de bit de données.

Taille des données (bits)	Nombre d'opérations MAC
32 et 32 × 16	1
16 × 16 et 16 × 8	2
8 × 8 et 8 × 4	4
4 × 4 et 4 × 2	8
2 × 2	16

5.3 Évaluation de l'unité MAC

5.3.1 Cadre expérimental

L'unité MAC présentée dans la section précédente est développée dans le langage de description matériel SystemVerilog. Les outils utilisés pour réaliser la simulation et la synthèse sont respectivement ModelSim et Synopsys Design Compiler. La simulation permet de vérifier la fonctionnalité de l'architecture MAC proposée. Elle permet également l'acquisition de l'activité de commutation, nécessaire pour une estimation précise de la puissance. La synthèse fournit les coûts de surface et de puissance dans la technologie sélectionnée, qui est la technologie FD-SOI de 28nm à 200MHz. Pour émuler des opérandes quantifiés à précision mixte, l'évaluation de l'unité MAC est réalisée avec des données de différentes largeurs de bits selon le tableau 5.2. Les valeurs des opérandes sont générées aléatoirement par des scripts Python. Ces données sont ensuite chargées dans les registres d'entrée 32 bits des architectures MAC avec un *benchmark* SystemVerilog. Pour chaque largeur de bit testée, les registres d'entrée des architectures MAC sont chargés 1000 fois avec des valeurs différentes.

Notre unité MAC sera comparée à l'unité MAC présente dans le cœur RI5CY [5]. La raison du choix de ce cœur est double: c'est un cœur connu dans la communauté entourant l'ISA RI5CY et les opérations supportées sont similaires à celles visées par notre unité MAC. Ce cœur

est également développé en SystemVerilog, il est disponible sur sa page Github dédiée [140]. Illustrée par la Figure 5.9, l'unité MAC est composée de cinq types de multiplieurs dont chacun correspond à une précision. La capacité de parallélisation est similaire au Tableau 5.2. Les outils ModelSim et Synopsys Design Compiler sont aussi utilisés pour effectuer la simulation et la synthèse avec la technologie FD-SOI de 28nm à 200MHz. Initialement, l'unité MAC du cœur RI5CY ne supporte pas les configurations 4 bits × 4 bits et 2 bits × 2 bits. Nous les avons ajoutées en respectant la description [6] et la syntaxe utilisée [140]. Contrairement à notre approche reconfigurable, cette unité MAC est conçue par redondance matérielle, c'est-à-dire, le support d'une nouvelle précision est permis grâce à des multiplieurs supplémentaires dédiés.

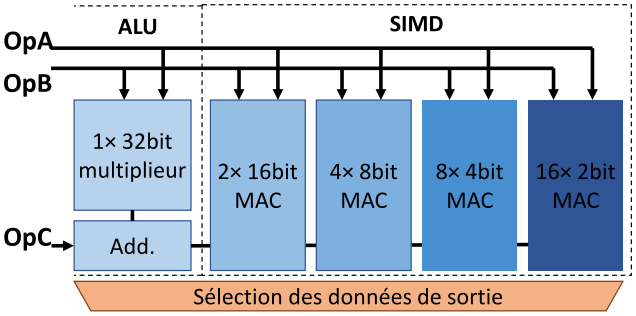


Figure 5.9 – Illustration de l'unité MAC du cœur RI5CY basée sur les descriptions [5, 6].

5.3.2 Estimation de surface occupée

La première information donnée par la synthèse de l'architecture est la surface. La valeur de surface obtenue est de $9930 \mu m^2$. La Figure 5.10 détaille la répartition de la surface en fonction des différentes parties de l'unité MAC. Le plus flagrant sur la figure est la partie contenant les additionneurs et les décalages qui représentent 75% du circuit et les 256 multiplieurs représentent 20% du circuit. La surface occupée par l'accumulateur est inférieure à 1% du circuit et les 4% restants sont occupés par les connexions et le multiplexeur de sortie. L'occupation de 75% de la surface par la partie contenant les additionneurs et les décalages s'explique par les faibles possibilités d'optimisation réalisée lors de la synthèse. Comme montré sur la Figure 5.8, entre les huit niveaux d'additionneur une captation des données vers la sortie est réalisée. Cette captation permet la récupération de résultat d'opération de multiplication inférieur à 32 bits. Empêchant l'outil de synthèse d'effectuer des optimisations consistant à réduire les nombres de portes logiques.

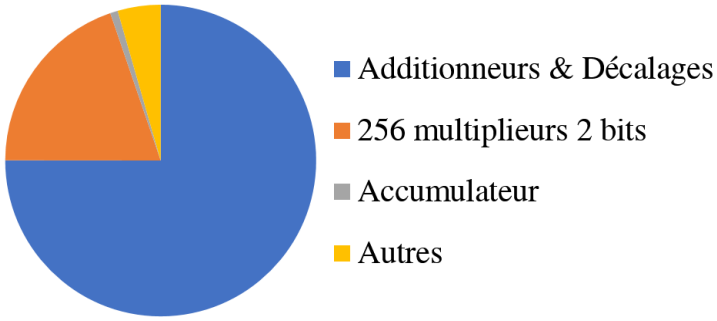


Figure 5.10 – La répartition de la surface occupée par les principales fonctions de l'unité MAC.

La surface de l'unité MAC du cœur RI5CY est de 10830μm^2. Notre proposition permet une réduction de 10%, par rapport à l'unité MAC de RI5CY. Cette réduction de surface est un point positif sachant que la surface de l'unité MAC du RI5CY représente environ 40% de la surface totale du cœur.

5.3.3 Estimation de puissance et efficacité énergétique

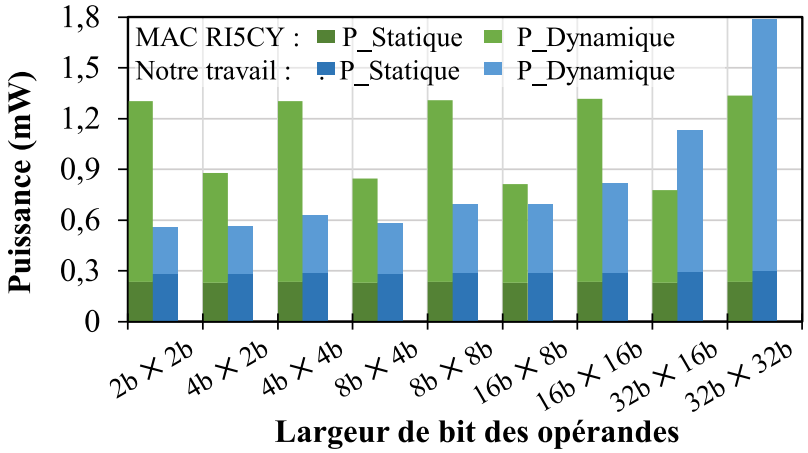


Figure 5.11 – Puissance dynamique fournie par Synopsys Design Compiler, incluant les activités de commutation du *testbench* simulé.

La figure 5.11 montre l'évolution de la consommation statique et dynamique pour les deux architectures MAC. La puissance dynamique de notre solution augmente progressivement en fonction de la largeur des bits de données. En effet, notre solution est très adaptée et n'active que le matériel requis pour une largeur de bit spécifique. C'est le principe de Diviser et Conquérir que nous utilisons qui permet d'activer seulement la logique nécessaire pour effectuer une multiplication est proportionnel à la largeur de bit des opérandes.

La puissance dissipée est directement liée au nombre de portes logiques activées. C'est pourquoi les opérations 32 bits qui utilisent toute la logique du multiplieur consomment plus d'énergie. Par rapport à RI5CY, la consommation d'énergie est plus élevée pour les opérations 32 bits × 16 bits et 32 bits × 32 bits. Notre explication présumée est un placement des composants logiques moins optimisé lors de la synthèse. En effet, notre unité MAC est composée de plusieurs connexions réparties sur le circuit, réduisant ainsi l'espace d'optimisation possible. Cela explique aussi la puissance statique légèrement plus élevée par rapport à RI5CY malgré une surface plus petite. Quant à l'unité MAC du cœur RI5CY, nous remarquons deux niveaux de puissance dynamique. Le niveau le plus élevé correspond aux valeurs des opérandes avec la même largeur de bit, et le niveau le plus bas correspond aux valeurs des opérandes avec une largeur de bit asymétrique. Cela s'explique par la présence de zéros supplémentaires pour s'adapter à la largeur de bit asymétrique. En moyenne, la puissance dynamique dissipée par notre architecture MAC est inférieure de 25 % à celle mesurée pour le RI5CY.

Pour comparer l'efficacité énergétique des deux solutions, la figure 5.12 indique le nombre d'opérations par mW. Pour les deux architectures MAC, la meilleure efficacité énergétique est obtenue avec les largeurs de bits les plus faibles. Une baisse de l'efficacité énergétique se produit à partir de données de 16 bits. En moyenne, l'efficacité énergétique est supérieure de 50 % pour l'unité MAC proposée par rapport à celle du cœur RI5CY.

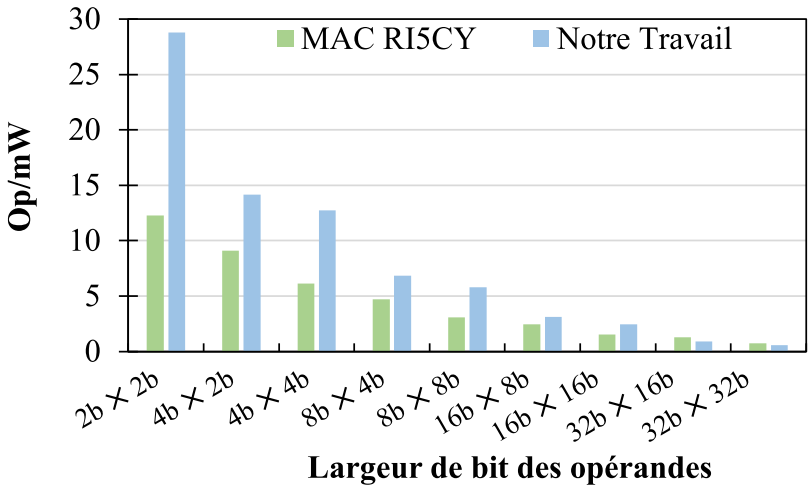


Figure 5.12 – Efficacité énergétique déterminée à partir de la puissance dynamique et du nombre d'opérations réalisable en parallèle par les unités MAC.

5.4 Résumé

Dans ce chapitre, nous avons présenté une unité de calcul MAC flexible destiné à augmenter les capacités de parallélisation des opérations MAC de microcontrôleur. L'exécution de réseau de neurones profond sur des microcontrôleurs de basse consommation est un réel défi. Les caractéristiques *compute-intensive* et *memory-intensive* des réseaux de neurones profonds sont opposées à l'utilisation de microcontrôleur. L'une des solutions pour réduire leur empreinte mémoire s'appuie sur la quantification des réseaux de neurones. La quantification consiste à réduire le nombre de bits codant les données. Au niveau des microcontrôleurs, l'utilisation de données représentées sur un nombre de bits moindre est exploitée grâce à des systèmes SIMD basés sur de la redondance matérielle des unités MAC présents et les précisions supportées sont souvent limitées 16 bits et 8 bits.

L'unité MAC présentée dans ce chapitre prend en charge toutes les précisions en puissance de 2 possibles, de 2 à 32 bits. L'unité MAC du cœur open-source RI5CY est utilisée comme référence pour évaluer notre solution proposée. Nous avons constaté une réduction de 25% de la puissance dynamique et également une amélioration de l'efficacité en matière d'énergie de 50%. De plus une réduction de la surface de 10% est à souligner par rapport à l'unité MAC du cœur RI5CY.

Dans le chapitre suivant, nous évaluons l'impact de la nouvelle unité MAC sur l'efficacité énergétique dans des architectures d'accélérateurs pour du *machine learning*.

Chapitre 6

Étude d'architectures d'accélérateurs pour les algorithmes d'apprentissage

Table des matières du chapitre

6.1 Introduction

Dans ce chapitre, l'objectif est d'évaluer différentes architectures matérielles accueillant notre unité MAC. Pour permettre une implantation et une évaluation rapide, nous avons choisi d'évaluer notre unité MAC à un niveau d'abstraction plus élevé contrairement aux chapitres précédents. Cette unité MAC offre une flexibilité de calcul en supportant différentes précisions de données. Elle permet aussi d'augmenter les capacités de parallélisation d'opérations effectuées par un microcontrôleur en évitant le recours à une solution SIMD avec de la redondance matérielle [51, 5].

6.2 Approche d'exploration : principes et choix

Le chapitre 4 a abordé l'implantation d'une architecture de processeur sur une puce FPGA tandis que le chapitre 5 a exploité une synthèse de l'unité MAC proposée, via un fichier RTL. Aucun de ces deux procédés ne facilite l'exploration d'espaces de conception potentiellement larges. La synthèse de fichier RTL est un processus demandant un temps de conception important et l'utilisation de cartes FPGA est une solution à privilégier lorsque les options de conceptions sont suffisamment restreintes pour une implantation matérielle. Ces approches demandent la manipulation d'un lourd écosystème rallongeant le temps d'exploration.

Une solution est donc d'effectuer l'exploration matérielle par simulation haut niveau d'abstraction. Le choix du simulateur est guidé par le niveau d'abstraction et le compromis temps/précision recherché. Dans notre cas, l'utilisation de l'outil Timeloop [108] permet de rapidement explorer différents choix matériels. Cette rapidité d'exploration est permise par son haut niveau d'abstraction. De plus, il permet de se concentrer sur plusieurs aspects architecturaux pertinents.

6.2.1 Cadre de modélisation Timeloop

Timeloop est un outil permettant d'explorer et d'évaluer des architectures d'accélérateurs destinées à exécuter des réseaux de neurones profonds. Cet outil permet une description de la topologie d'une architecture basée sur quelques éléments clés tels que les composants matériels arithmétiques et des mémoires. Il peut émuler des architectures pour estimer leurs performances et efficacités énergétiques selon une charge de travail donnée. En fonction du nombre d'opérations à réaliser, Timeloop déterminera le nombre de mouvements de données

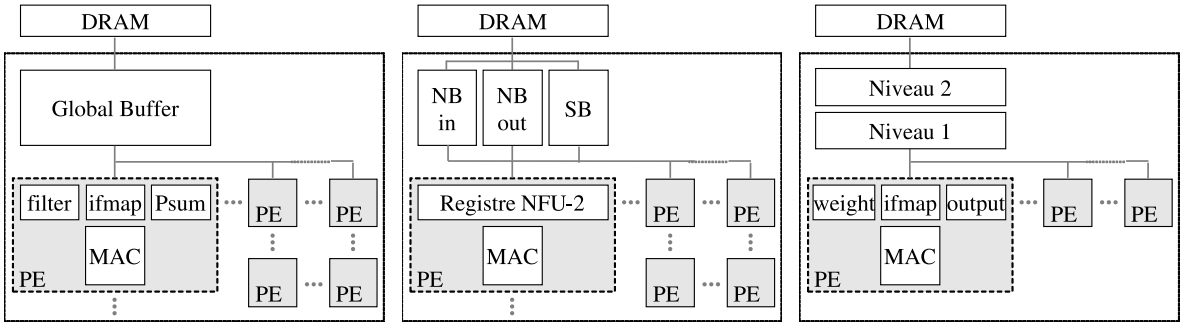
dans chaque mémoire et le nombre d'opérations effectuées par chaque unité arithmétique. Il estime aussi la consommation énergétique.

Les charges de travail supportées par Timeloop sont des réseaux de type CNN. Ces derniers étant basés sur des structures matricielles, Timeloop peut ainsi supporter tout type de charges de travail correspondant à ce type de structure. Les couches de convolution sont évidemment supportées comme charge de travail. Mais aussi, la charge de travail peut être simplement une multiplication de matrice.

L'intégration de notre unité MAC dans des architectures qui sont décrites dans la section suivante est réalisée avec l'utilisation de l'outil Accelergy [141]. Il s'agit d'un outil utilisé en complément de Timeloop pour affiner l'estimation de la consommation. Nous avons apporté les modifications nécessaires aux librairies de composants utilisées par Accelergy pour qu'il puisse avoir accès aux données de consommations des unités MAC présentées dans le chapitre 5 et du cœur RI5CY. La principale difficulté est la bonne prise en compte des différentes précisions supportées par les unités MAC. De plus, des modifications mineures du code dans Timeloop ont été réalisées pour supporter le calcul de plusieurs données simultanément avec une seule unité MAC.

6.2.2 Architectures matérielles évaluées

Nous considérons trois architectures dans notre étude : Eyeriss [63], DianNao [75] et GAP8 [112], respectivement représentées sur les Figures 6.1a, 6.1b et 6.1c. Ces Figures représentent une version simplifiée de chaque architecture afin qu'elles puissent être ensuite modélisées dans Timeloop.



(a) Eyeriss. (b) DianNao. (c) GAP8.

Figure 6.1 – Schéma simplifié des architectures.

L'architecture de **Eyeriss** est composée de 168 *processing elements* (PE), dont chaque PE dispose de trois registres pouvant accueillir respectivement les poids des filtres de CNN, les données entrées et les résultats des opérations MACs. Chaque PE est connecté à une mémoire tampon SRAM (*Global Buffer*) pouvant contenir les données des poids et des cartes des caractéristiques d'entrée (input feature map - ifmaps) entre les PE et la DRAM. Les données de sortie sont directement envoyées dans la mémoire principale DRAM. Les PE de l'architecture de **DianNao** contiennent un registre unique pouvant accueillir les données de poids. Les PE seront alimentés en données par trois mémoires partagées pouvant accueillir chacune un type de données: NBin = des cartes des caractéristiques d'entrée; NBout = des cartes des caractéristiques de sortie et SB = les poids. Pour ces deux architectures, les PE sont organisés sous la forme d'un tableau x et y. La troisième architecture modélise celle de la puce **GAP8**. Elle se distingue des deux autres architectures par une organisation de ces PE en parallèle. Comme Eyeriss, les PE de GAP8 sont composés de trois mémoires pouvant stocker un poids des filtres de CNN, une donnée d'entrée et un résultat des opérations MACs. Les PE reçoivent des données issues de la mémoire principale par deux niveaux de mémoires distincts, notés Niveau 1 et Niveau 2. Pour les trois architectures, les mémoires se comportent comme des mémoires *scratchpad*.

Lors de la modélisation de ces architectures dans Timeloop, le nombre de leurs PE sera fixé à 96. Ce nombre de PE est choisi en prenant en compte la mémoire disponible, afin qu'elles ne soient pas un facteur limitant.

La procédure de modélisation des architectures dans Timeloop est identique pour les trois architectures. Dans la section suivante, la modélisation de l'architecture de la puce GAP8 y sera détaillée. Nous avons privilégié la modélisation de cette architecture, car nous avons en notre possession la carte physique GAPuino accueillant la puce GAP8. Cela nous donne l'opportunité de valider la consistance du modèle Timeloop correspondant.

6.3 Modélisation et validation d'une architecture : exemple de la GAP8

Contrairement aux architectures Eyeriss et DianNao couramment citées dans l'état de l'art, le choix de modéliser GAP8 est guidé par sa composition avec des cœurs RI5CY. Dans le chapitre 5, l'unité MAC décrite a pour objectif premier d'être intégrée dans un cœur 32 bits tel que le cœur RI5CY. Ayant choisi une approche avec un haut niveau d'abstraction dans ce

chapitre, il est pertinent de modéliser une architecture d'accélérateur composé du cœur RI5CY (et d'autant plus que ce type de cœur est présent dans la carte GAPuino).

6.3.1 Modélisation de l'architecture GAP8 dans Timeloop

La puce GAP8 est embarquée sur la carte GAPuino, elle est destinée à l'exécution efficace d'algorithmes de ML. Dans la section 3.3, nous avons introduit cette carte plus en détail lors d'une évaluation concise. Avec son haut niveau d'abstraction, la modélisation dans Timeloop se concentre sur certains composants clés d'architecture d'accélérateur. Comme nous l'avons évoqué dans la section 2.3, au moins deux composants fondamentaux sont à prendre en compte dans la conception d'architecture exécutant efficacement des techniques de ML : la mémoire et l'unité arithmétique réalisant les opérations MAC. Ces deux composants sont justement au cœur des modèles dans Timeloop. La modélisation de GAP8 par Timeloop est concentrée sur les mémoires et les unités MAC, comme l'illustre son schéma en Figure 6.1c.

Pour la modélisation de chaque composant du modèle, il est possible de définir le noeud technologique désiré. Actuellement, Timeloop supporte les technologies de 40nm et 65nm. Pour modéliser des mémoires, Timeloop prend en compte différentes caractéristiques telles que la géométrie (largeur, hauteur), la taille des données y transitant et le débit. La modélisation des unités arithmétiques est caractérisée par le type d'opération à réaliser, la taille des données supportées et le type de composants.

Chacune des propriétés citées ci-dessus est utilisée pour déterminer l'activité de l'architecture en fonction d'une charge de travail donnée. Par activité, on entend les différentes transactions mémoires et le nombre d'opérations réalisées par les unités de calcul. La consommation d'énergie est déterminée en fonction des caractéristiques de l'architecture et de son activité.

6.3.2 Validation de la modélisation

L'une des questions qui se posent lors de l'utilisation de simulateur avec un haut niveau d'abstraction est sa capacité à fournir des données pertinentes vis-à-vis d'une implantation matérielle. Pour répondre à cette question, nous reprenons les données de consommation d'énergie présentées dans la section 3.3. Notre objectif est de vérifier si les tendances de consommation correspondent entre ceux obtenus avec Timeloop et la carte GAPuino. Nous parlons seulement de tendance, car les niveaux de précision sont très différents. Comme pour les données d'énergie issues précédemment de la carte GAPuino, la charge de travail

utilisée dans Timeloop est le CNN MNIST présenté dans la section 2.1.1. Elle permet une consommation dynamique suffisamment importante par rapport à la consommation statique de la carte GAPuino.

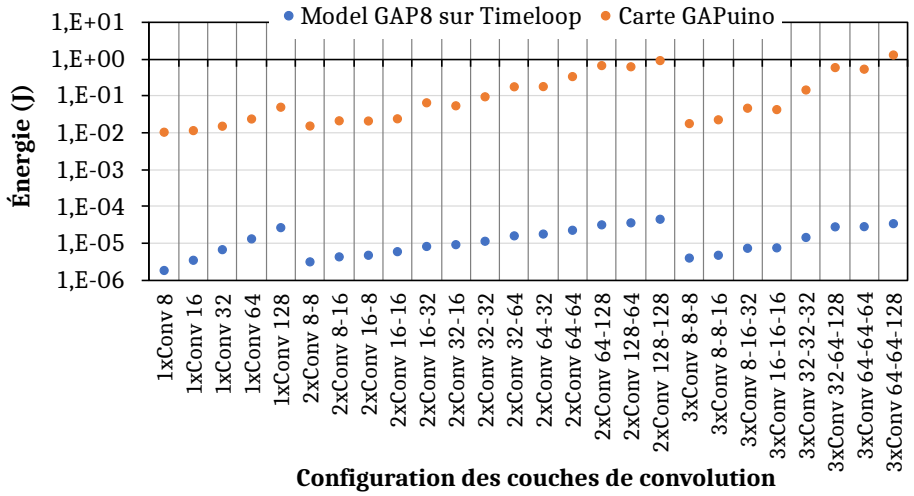


Figure 6.2 – Comparaison entre l'énergie mesurée sur la carte GAPuino et l'énergie estimée du modèle Timeloop inspiré de la puce GAP8.

La Figure 6.2 montre les consommations d'énergie obtenues par le modèle Timeloop et par la carte GAPuino. Comme, nous l'avons mentionné précédemment, la consommation d'énergie de la carte GAPuino est supérieure au modèle de Timeloop étant compris entre une dizaine de mJ aux joules. Tandis que le modèle de Timeloop est compris entre 1.8 uJ à 50 uJ. On peut constater que pour chaque configuration du CNN MNIST la tendance de consommation est similaire. On peut noter même que certaines singularités de consommation sont observables sur les deux "courbes", notamment pour les configurations 3xConv 32-64-128 et 3xConv 16-16-16 dans la figure.

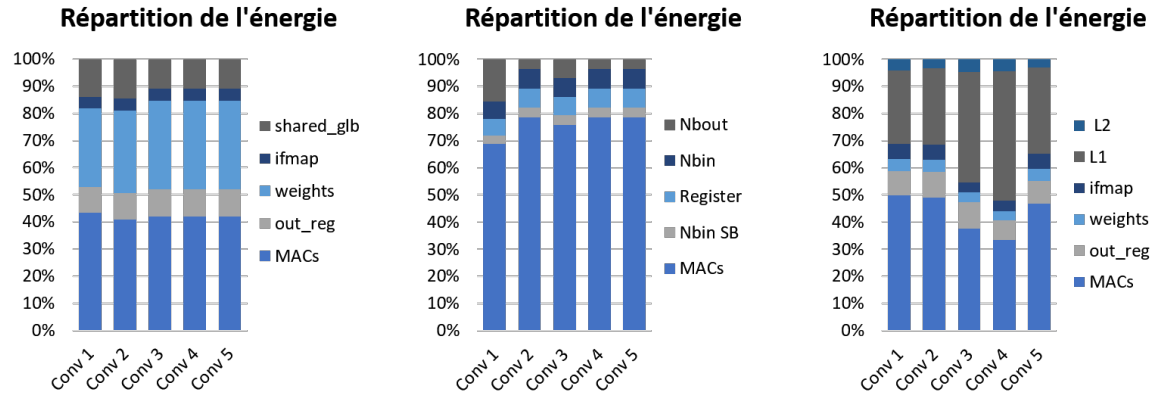
Cette similitude des tendances suggère d'une certaine façon la validité du modèle de l'architecture de GAP8 que nous avons construit dans Timeloop. En suivant la même méthodologie de modélisation dans Timeloop, nous définissons ainsi les équivalents des deux autres architectures, à savoir Eyeriss et DianNao.

6.4 Exploration d'architectures selon le type de MAC

6.4.1 De la pertinence d'optimiser le MAC

Avant de commencer à inclure notre unité MAC dans les trois architectures modélisées, il est intéressant de se demander si une marge d'optimisation est possible. Sachant que les architectures d'accélérateurs sont composées de plusieurs niveaux de mémoires différents, on pourrait s'attendre à une consommation d'énergie nettement supérieure de la part des mémoires des architectures.

Nous analysons donc la répartition de la consommation d'énergie sur les différents composants des trois architectures. Cela permet de disposer d'un clair aperçu de l'impact des éléments considérés. Pour l'analyse de la répartition d'énergie ainsi que pour l'ensemble des résultats obtenus lors de l'exploration présentée dans les prochaines sections, la charge de travail est définie par les cinq couches de convolutions du CNN AlexNet [20].



(a) Eyeriss. (b) DianNao. (c) GAP8.

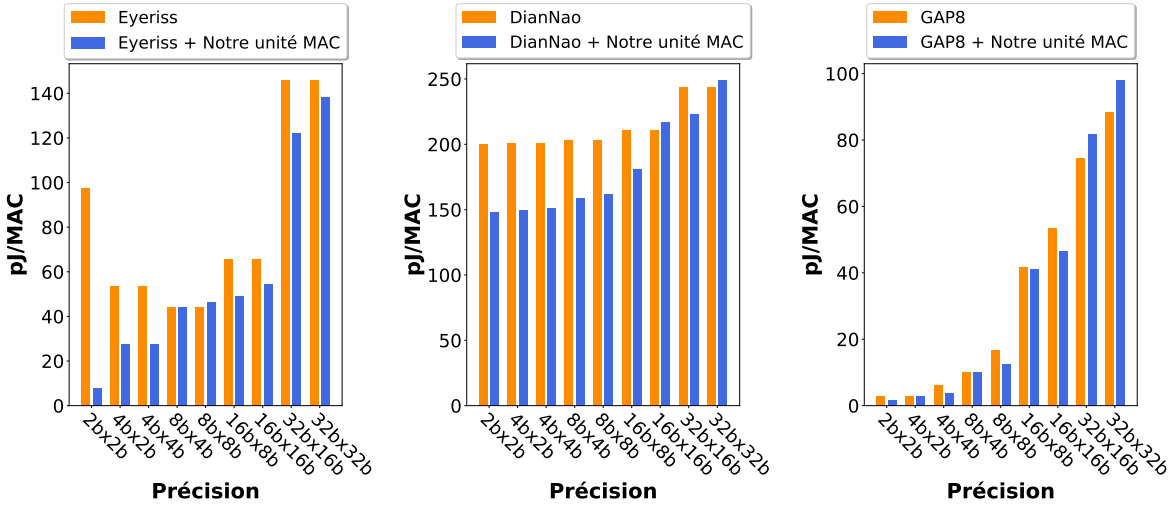
Figure 6.3 – Répartition de la consommation d'énergie entre les composants des architectures.

La répartition de l'énergie obtenue pour les trois architectures est représentée sur la Figure 6.3. La consommation des unités MAC se révèle être non négligeable pour chacune des architectures. Elle représente près de 41% de la consommation d'énergie pour Eyeriss et GAP8. Tandis que pour DianNao elle atteint les 75% en moyenne. Les répartitions d'énergies représentées sur la Figure 6.3 se concentrent sur les composants que l'on peut considérer sur puce, excluant ainsi la mémoire DRAM. Nous pouvons tout de même mentionner que la consommation d'énergie de la DRAM représente 60%, 95% et 23% de la consommation totale des architectures Eyeriss, DianNao et GAP8 respectivement.

La répartition de la consommation d'énergie montre un impact non négligeable des unités MAC sur les systèmes. Par conséquent, celles-ci sont des cibles intéressantes pour de possibles optimisations en vue d'améliorer l'efficacité énergétique.

6.4.2 Impact de notre unité MAC sur l'efficacité énergétique

Notre objectif à présent est d'analyser l'impact de la présence de notre unité MAC dans chacune des trois architectures considérées. La Figure 6.4 présente l'efficacité énergétique obtenue pour ces dernières. Nous utilisons comme charge de travail les cinq couches de convolution du CNN AlexNet.



(a) l'architecture Eyeriss. (b) l'architecture DianNao. (c) l'architecture GAP8.

Figure 6.4 – Impact de notre unité MAC sur l'efficacité énergétique des architectures.

L'amélioration apportée par notre unité MAC est le plus marquée sur les architectures Eyeriss et DianNao. L'explication est dans la capacité de notre unité MAC à paralléliser les opérations pour des données de précision réduite. Les unités MAC par défaut d'Eyeriss et DianNao peuvent réaliser seulement une opération MAC à la fois. On peut constater une amélioration moyenne de l'efficacité énergétique de 32 % et 19% respectivement pour les architectures Eyeriss et DianNao.

Pour l'architecture GAP8, on retrouve les tendances observées dans le chapitre 5, notre unité est plus efficace pour les précisions réduites. Des précisions 2 bits × 2 bits à 16 bits × 16 bits, la consommation d'énergie par opération MAC est réduite de 23% avec la présence de notre unité MAC. Comme attendu pour les opérations impliquant des données en 32 bits,

notre unité MAC augmente la consommation d'énergie par opération MAC de 10% et 2.5% pour l'architecture GAP8 et DianNao respectivement. Pour l'architecture Eyeriss, l'efficacité énergétique est améliorée de 10 % en moyenne pour les opérations MAC impliquant des données sur 32 bits.

À présent que nous avons regardé l'efficacité énergétique de chaque précision obtenue pour le CNN AlexNet sur les trois architectures, nous abordons l'efficacité énergétique obtenue pour chaque couche de convolution du CNN AlexNet. Les résultats seront présentés couche par couche, pour l'architecture cible GAP8.

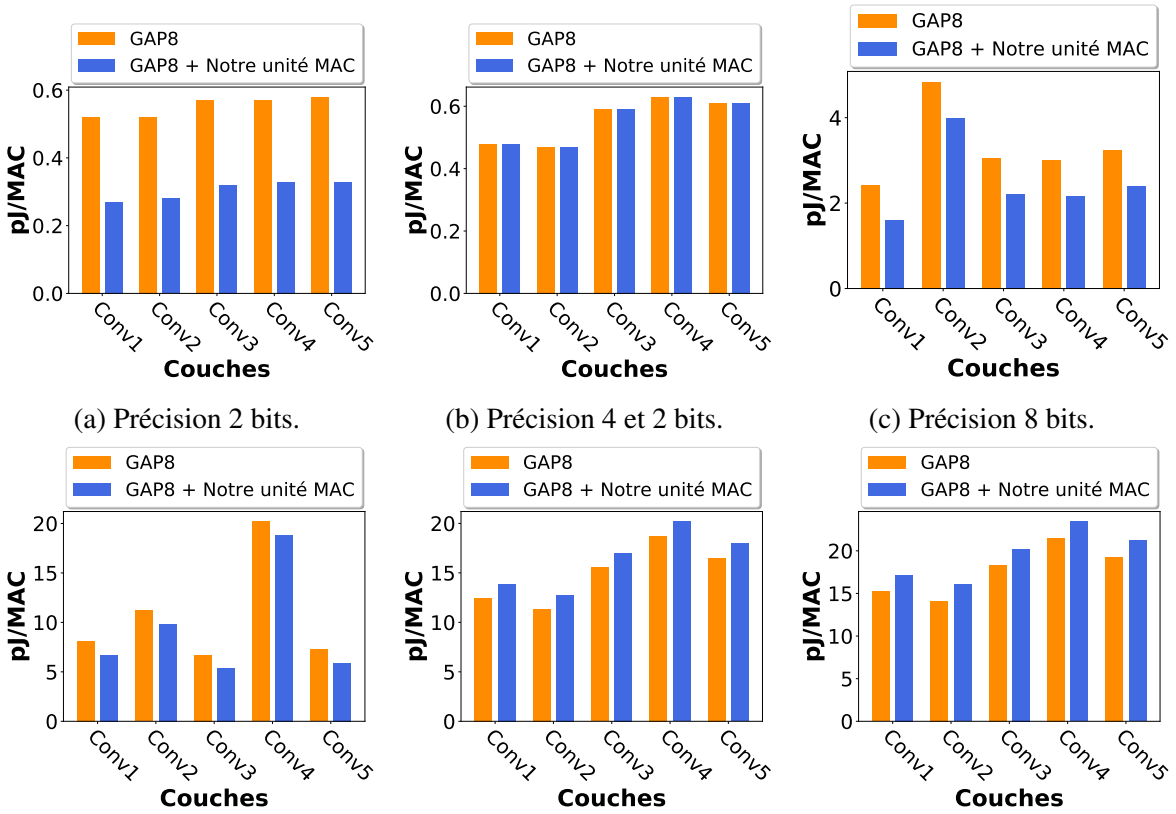
La Figure 6.5 regroupe les données d'efficacité énergétique obtenues couche par couche par des précisions sélectionnées par rapport aux observations faites pour la Figure 6.4. On peut remarquer que l'efficacité énergétique n'est pas uniforme entre les couches et entre les précisions. La variation du coût énergétique par opération des couches est attendue. Effectivement, le coût énergétique global d'un CNN n'est pas uniformément réparti sur ses couches le constituant. Le nombre d'opérations à réaliser et le volume de données à manipuler varient d'une couche à l'autre. Cependant, on peut remarquer que d'une précision à l'autre les variations sont différentes. En effet, la couche de convolution 4 est la plus coûteuse pour les précisions au-dessus de 16 bits. Mais pour la précision en 8 bits, c'est la couche de convolution 2 qui est clairement la plus coûteuse. Pour l'architecture Eyeriss des observations similaires sont faites. En revanche, pour DianNao ce n'est pas le cas. La tendance d'efficacité énergétique ne semble pas être impactée par la précision.

Globalement, notre unité MAC a un impact positif sur les trois architectures considérées. Cependant, pour certaines précisions des données, l'apport de notre unité MAC apparait atténué. Une des principales différences entre les architectures considérées est leurs mémoires. Dans la section suivante, évaluons l'impact sur l'efficacité énergétique de notre unité MAC dans ces architectures du point de vue de la mémoire.

6.4.3 Influence sur l'activité mémoire

Nous nous concentrons sur l'activité en lecture et en écriture de la mémoire DRAM. Comme déjà mentionné dans la section 6.4.1, la mémoire DRAM représente 60%, 95% et 23% de la consommation totale d'énergie des architectures Eyeriss, DianNao et GAP8 respectivement.

Sur la Figure 6.6, on peut voir une variation du nombre de lectures et d'écritures au niveau de la mémoire DRAM en fonction de la précision. Pour l'architecture DianNao, la Figure 6.6h est représentative des autres précisions. Précédemment, nous avons mentionné que l'efficacité



(a) Précision 2 bits. (b) Précision 4 et 2 bits. (c) Précision 8 bits.

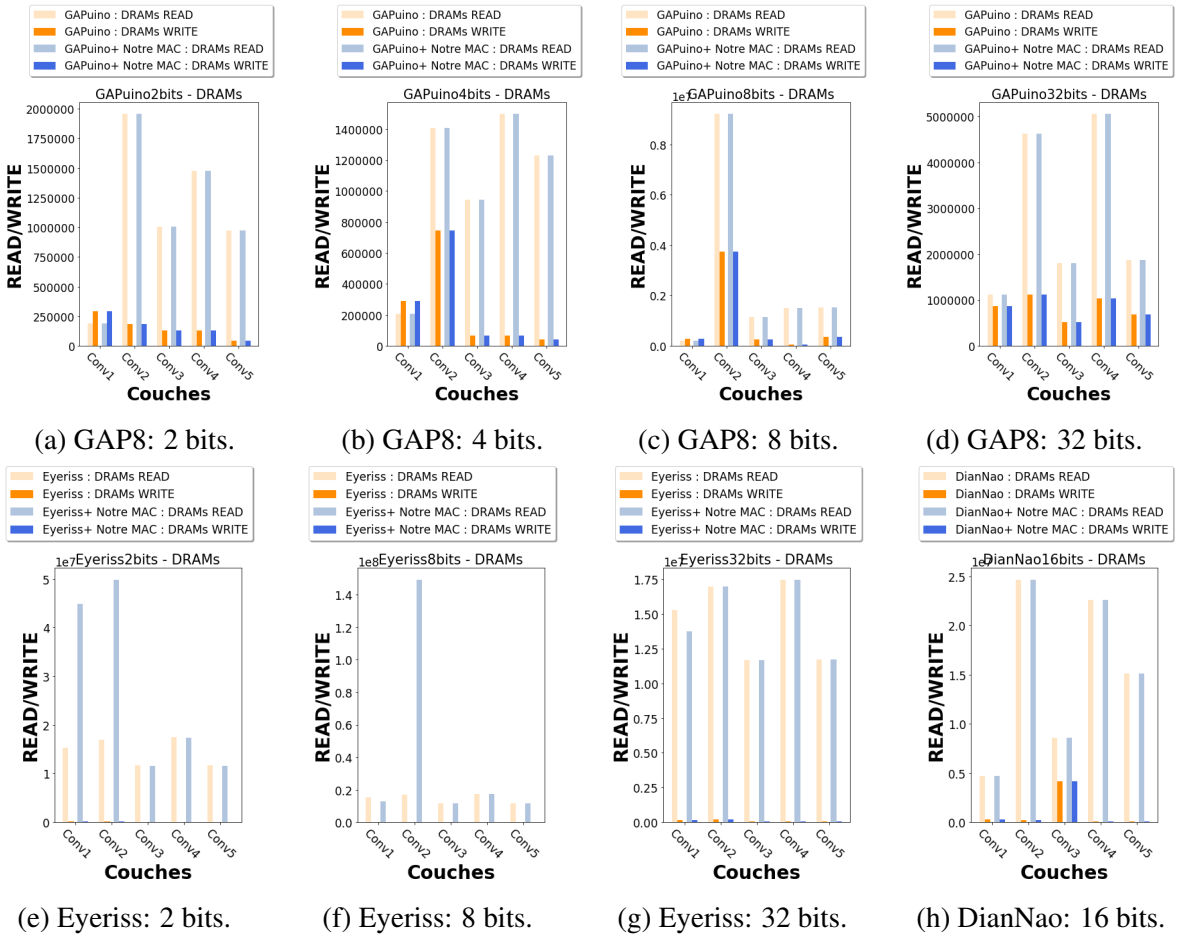
(d) Précision 16 bits. (e) Précision 32 et 16 bits. (f) Précision 32 bits.

Figure 6.5 – Détails de l'efficacité énergétique de l'architecture GAP8 pour différentes précisions des données.

énergétique de l'architecture DianNao semble être indépendante de la précision des données. Cette remarque est confirmée par le nombre de lectures et écritures sur la DRAM qui est identique, quelle que soit la précision. Pour les architectures Eyeriss et GAP8, leurs nombres de lectures et écritures en mémoire DRAM varient aussi en fonction de la précision des données.

6.4.4 Impact sur l'utilisation des PE

Lors de la modélisation des architectures Eyeriss, DianNao et GAP8, nous avons défini le même nombre de PE pour qu'ils puissent être comparables. Jusqu'à présent nous n'avons pas abordé la question du nombre de PE utilisés lors de la réalisation d'une charge de travail. Pourtant cette question est essentielle, généralement, l'objectif est d'adapter l'utilisation des ressources matérielles, dont les PE, pour parvenir à atteindre la meilleure efficacité énergétique pour une charge de travail donnée.

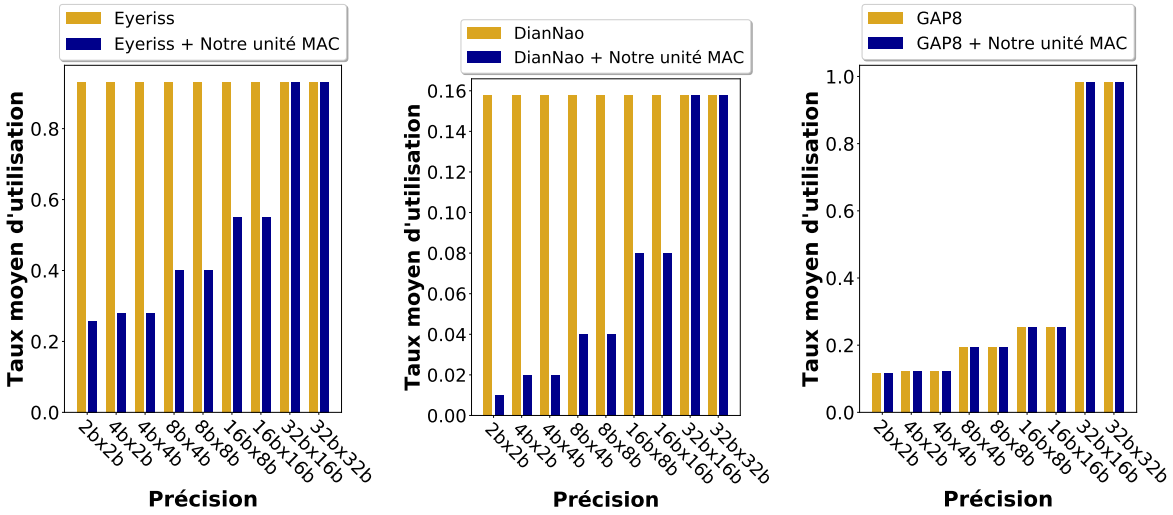


(a) GAP8: 2 bits. (b) GAP8: 4 bits. (c) GAP8: 8 bits. (d) GAP8: 32 bits.

(e) Eyeriss: 2 bits. (f) Eyeriss: 8 bits. (g) Eyeriss: 32 bits. (h) DianNao: 16 bits.

Figure 6.6 – Nombre de lectures et d'écritures de la mémoire DRAM pour différentes précisions de données pour les architectures GAP8, Eyeriss et DianNao.

La figure 6.7 décrit le taux d'utilisation des PE disponibles pour les trois architectures avec ou sans notre unité MAC. Pour Eyeriss et DianNao leur taux d'utilisation de PE est supérieur à 80% avec leurs unités MAC à précision unique. Une fois que notre unité MAC est installée, on peut observer une variation du taux d'utilisation en fonction de la précision utilisée. L'explication est liée aux capacités croissantes de parallélisation de notre unité MAC en réduisant la précision. Par conséquent, pour une charge de travail donnée, Timeloop tente de minimiser la consommation d'énergie par une optimisation de la répartition des données entre les mémoires et les PE. Ainsi, pour une charge de travail donnée et des PE capables d'effectuer des opérations en parallèle en basse précision, le nombre de PE requis est réduit. Nous pouvons le confirmer à travers la Figure 6.7c, où le taux d'utilisation de PE est similaire avec ou sans notre unité MAC du fait que l'unité MAC embarquée dans l'architecture GAP8 supporte différentes précisions. Comme, nous l'avons montré précédemment dans la section



(a) l'architecture Eyeriss. (b) l'architecture DianNao. (c) l'architecture GAP8.

Figure 6.7 – Comparaison du taux d'utilisation des PE des architectures avec et sans notre unité MAC.

6.4.2, malgré la réduction du taux d'utilisation de PE, notre unité MAC permet tout de même de réduire la consommation d'énergie.

6.5 Analyse générale

Les architectures d'accélérateurs destinés à des systèmes embarqués supportent souvent des CNN quantifiés. L'avantage pour ces architectures est de permettre un gain en énergie et en surface. Avec la Figure 6.4, nous avons montré l'impact sur l'efficacité énergétique d'unité MAC de précision flexible. Plus largement, nous avons pu constater que munir les PE de capacités de parallélisation pour des opérations MAC de faible précision permet un gain d'efficacité énergétique. Outre le gain en efficacité énergétique observé, permettre aux accélérateurs de supporter de multiple précision leur évite de s'enfermer dans des conditions d'utilisation trop restreinte.

Différentes architectures existent pour le support de plusieurs précisions par une unité MAC. Le chapitre 3, nous a permis d'en présenter quelqu'une. Nous avons pu acquérir la carte GAPuino embarquant la puce GAP8 dont les cœurs RI5CY contiennent une architecture d'unité MAC pouvant supporter de plusieurs précisions. Confirmant les observations du chapitre 5, notre unité MAC présente un gain d'énergie supérieur à l'approche utilisée par les cœurs RI5CY. La Figure 6.5 met bien en avant la réduction d'énergie consommée par opération MAC. Une

approche basée sur une redondance matérielle telle que présente dans le cœur RI5CY n'est pas idéale en termes d'efficacité énergétique. Les approches basées plutôt sur une reconfigurabilité matérielle telle que notre unité MAC est plus prometteuse.

L'une des craintes que peut entraîner l'utilisation d'unité MAC pouvant réaliser plusieurs opérations MAC simultanément pour des précisions réduites est une augmentation significative de l'activité des mémoires. Dans la section 6.4.3, nous avons observé qu'il n'y a pas de variation significative de l'activité mémoire pour les trois architectures considérées. Cependant, avec notre unité MAC dans l'architecture Eyeriss, différentes singularités en nombre de lectures sont à noter. Malgré ces pics en nombre de lectures, la présence de notre unité MAC permet tout de même une amélioration de l'efficacité énergétique. Bien sûr, avec le haut niveau d'abstraction de Timeloop, il convient de confronter ces observations avec un support d'exploration à un niveau d'abstraction prenant en considération les unités de contrôle de mémoire, par exemple un mécanisme de type MMU.

Les architectures d'accélérateurs s'appuient sur la présence de nombreux PE pour permettre la parallélisation des opérations et la consommation d'énergie est limitée par la réduction de la précision des données supportée. La section 6.4.4 nous a permis de constater qu'aussi bien notre unité MAC ou bien celle de la puce GAP8 permet une réduction de nombre de PE utilisé lors de la résolution d'une charge de travail. Hormis les réflexions faites précédemment sur les bienfaits de doter les PE de capacité de parallélisation, cette réduction du nombre de PE utilisés associés à une fonctionnalité de *power gating* sur les PE inutilisés peut être une opportunité de réduction de l'énergie consommée par les accélérateurs.

6.6 Résumé

Ce chapitre nous a permis d'évaluer notre unité MAC sur 3 architectures d'accélérateurs différentes grâce au simulateur Timeloop. L'utilisation de Timeloop est guidée par son haut niveau d'abstraction permettant une rapide modélisation de différentes architectures et de notre unité MAC. Préalablement, à toute autre expérimentation, nous avons évalué la validité de la modélisation d'une architecture par rapport à une carte correspondante en notre possession. Nous avons pu observer une similitude entre les tendances d'énergie consommée par la carte et d'énergie estimée par le simulateur. Validant ainsi le modèle et nous permettant de poursuivre l'évaluation notre unité MAC.

Avant d'implanter notre unité MAC dans les trois architectures considérées, une question légitime se pose : est-ce que la modification de l'unité MAC utilisée par ces architectures est pertinente ? En ne considérant pas la consommation de la mémoire DRAM, nous avons pu constater que la consommation des unités MAC des architectures atteignait le 40% à même 75%. Avec une réelle opportunité de réduction de la consommation d'énergie des unités MAC dans les architectures, nous avons ensuite évalué l'impact de la présence de notre unité MAC du point de vue de l'efficacité énergétique et l'utilisation des mémoires des architectures considérées. Le constat général est une amélioration manifeste du coût par opérations des architectures avec notre unité MAC. Sans pour autant avoir un impact significativement négatif sur les mémoires. De plus, une réduction du nombre de PE a été observée ouvrant une opportunité d'optimisation de l'utilisation des ressources matérielles des architectures.

Chapitre 7

Conclusion et perspectives

Table des matières du chapitre

7.1 Conclusion

La mise en œuvre embarquée de techniques d'intelligence artificielle et plus particulièrement de *Machine Learning* (ML) représente plus que jamais un défi. De nombreuses recherches s'attachent ainsi à répondre à la problématique d'exécuter efficacement des algorithmes de ML sur des systèmes embarqués contraints en ressources matérielles. Dans cette thèse, nous avons étudié des pistes visant à identifier des ingrédients architecturaux répondant à cette attente.

Au travers du chapitre 3, nous avons rappelé les caractéristiques de base des techniques de ML, ainsi que les choix courants d'optimisations logicielles et matérielles. Les réseaux de neurones occupent particulièrement une grande partie de la littérature étudiée. En effet, ils montrent d'excellentes capacités à exploiter de grandes masses de données. Néanmoins, leur utilisation implique la réalisation de nombreuses opérations de multiplication-accumulation (MAC). À titre d'exemple, le réseau de neurones convolutif AlexNet [20] nécessite 724 millions d'opérations MAC comme indiqué dans le tableau 2.1 pour une inférence. Les optimisations logicielles visent à réduire le nombre d'accès mémoire et d'opérations MAC impliquées lors des exécutions. Les optimisations matérielles quant à elles, promeuvent un maximum de parallélisme des opérations MAC et une organisation mémoire évitant d'être le goulot

d'étranglement. Nous avons également abordé différentes approches de conception pour l'exploration d'architecture matérielle. Celles-ci reposent sur une divers niveaux d'abstraction selon les besoins en précision et en rapidité des raisonnements. Enfin, le chapitre est cloturé par une évaluation que nous avons menée sur deux cartes dédiées au ML: la Coral Dev Board [142] et la GAPuino [143]. La première carte s'appuie sur l'utilisation d'un accélérateur nommé Edge TPU [114] tandis que la seconde s'appuie sur l'utilisation de huit cœurs RI5CY [5]. Entre autres, cette évaluation nous a permis de constater que la puissance consommée par l'architecture multicœur de la carte GAPuino est proportionnelle à la charge de travail. Contrairement à l'accélérateur de la carte Coral Dev Board qui une fois active à une puissance consommée stable, quelle que soit la charge de travail.

Le chapitre 4 a été consacré à la conception et à l'évaluation d'une architecture multicœur hétérogène efficace en énergie. Ce travail a montré l'intérêt d'exploiter des cœurs hétérogènes au sein d'une architecture, notamment par une gestion des différents types de cœurs de l'architecture adaptée à la charge de travail exécutée. Nous avons pu observer un gain d'énergie moyen de 22% lorsque nous adaptons l'utilisation des cœurs à la nature des tâches à exécuter. Cependant, selon les observations issues de nos expérimentations, l'utilisation de ce type d'architecture de calcul a des limitations pour l'exécution performante de techniques de ML reposant sur des réseaux de neurones en particulier. Ce constat est conforté par l'évaluation faite dans le chapitre 3, malgré une puissance consommée plus réduite et proportionnée à la charge de travail, les temps d'exécution montrés par l'accélérateur de CNN Edge TPU restent nettement plus faibles. En conséquence, l'énergie consommée par la carte Coral Dev Board est inférieure.

En guise de piste complémentaire, nous nous sommes focalisés par la suite sur une approche mettant en avant l'utilisation de mécanismes d'accélérations dédiés aux réseaux de neurones. Dans le chapitre 5, nous avons ainsi travaillé à améliorer les capacités d'exécution d'opérations MAC de cœurs de microprocesseur en définissant une nouvelle unité MAC à précision flexible. L'architecture de cette unité MAC vise à à adapter l'utilisation de l'unité et à donner aux cœurs la capacité à paralyser les opérations MAC lorsque les données sont représentées sur moins de 32 bits. Par rapport à une approche SIMD basée sur de la redondance matérielle du cœur RI5CY [5, 6], l'unité MAC proposée dans ce chapitre a montré une baisse de la puissance dynamique intéressante, de l'ordre de 25%.

Enfin, dans le chapitre 6, nous avons étudié l'intégration de la nouvelle unité MAC proposée au chapitre précédent dans des architectures typiques d'accélérateurs de réseaux de neurones.

Nous avons exploité l'outil de simulation à haut niveau d'abstraction Timeloop [108] afin de faciliter l'exploration de larges espaces de conception. Nous avons modélisé l'unité MAC développée dans le chapitre 5 puis nous l'avons incluse dans trois architectures d'accélérateur populaires. La simulation réalisée par Timeloop nous a permis de mettre en évidence des gains en efficacité énergétique au niveau de l'architecture globale, par rapport aux unités MAC de base.

7.2 Quelques perspectives

Plus généralement, une perspective à considérer est la possible extension du système hétérogène proposé dans le chapitre 4 avec des accélérateurs de ML. L'évaluation de ce système a mis en lumière le potentiel de ce type d'architecture pour le traitement de charge de travail de natures différentes. Une nouvelle version de l'architecture est envisageable en considérant d'autres types de cœurs. Actuellement, l'un des cœurs de l'architecture se distingue par la présence d'un FPU. On pourrait considérer d'autres fonctionnalités telles que l'utilisation d'accélérateurs d'opérations de type MAC ou type transformée de Fourier rapide (Fast Fourier Transform, FFT). Ce type d'accélérateur peut s'avérer tout à fait pertinent pour une architecture ciblant la résolution de technique de ML sachant que des méthodes de résolution de CNN sont basées sur la FFT. Néanmoins, la définition de tels systèmes hétérogènes généralisés nécessitera une réflexion sur un modèle de programmation adapté, permettant une gestion efficace des tâches sur les différents éléments de calcul du système.

D'autre part, pour répondre aux besoins de parallélisation d'opérations MAC de données de faible nombre de bits, nous avons présenté une proposition dans le chapitre 5. L'impact de celle-ci a été évalué sur des architectures d'accélérateurs dans le chapitre 6. Il semble que cela ouvre une opportunité de réflexion concernant son impact particulier sur les accès mémoires (du niveau cache au niveau mémoire principale) pour de possibles optimisations de l'efficacité énergétique globale des architectures.

Pour aller plus loin, nous pourrions mentionner la prise en compte des technologies de mémoire émergentes. Les mémoires non volatiles (NVM) ont des caractéristiques intéressantes pour les systèmes de calcul embarqués dédiées au ML. En effet, la mise en œuvre de réseau de neurones demande un stockage des données issues de la phase d'apprentissage du réseau de neurones. L'opportunité de stocker ces données sur une mémoire non volatile ayant une consommation statique très faible permettrait une économie d'énergie substantielle. De nom-

breux travaux ont déjà abordé une question similaire indépendamment du contexte du ML, à savoir l'implantation de NVM dans des systèmes de calcul que ce soit au niveau des registres d'un processeur [144], ou bien de la hiérarchie mémoires [145–149]. Pour tirer avantage des mémoires NVM, il convient de limiter le nombre d'écritures qui pour de nombreuses technologies de NVM consomment bien plus d'énergie (STT-MRAM, ReRAM, PC-RAM). Pour ce faire, des analyses de programmes peuvent y contribuer [148, 150, 151]. Malheureusement, les limitations actuelles du simulateur Timeloop dans la modélisation des latences de lecture et d'écriture des mémoires non volatiles ne permettent pas d'aborder le problème. Cette piste de travail reste ouverte avec l'utilisation des simulateurs alternatifs.

Enfin, d'autres opportunités technologiques existent pour permettre une exécution efficace en énergie de technique de ML. L'une de ces opportunités est le rapprochement des données stockées en mémoire et des unités arithmétiques. D'une approche basée sur l'intégration de mémoire en 3D au Near Processing Memory, le Processing In-Memory semble être une approche prometteuse pour une utilisation sur des architectures au niveau du edge computing [152–155].

Liste des publications

G. Devic, M. France pillois, J. Salles, G. Sassatelli, and A. Gamatié, "Highly-Adaptive Mixed-Precision MAC Unit for Smart and Low-Power Edge Computing," in 19th IEEE Interregional NEWCAS Conference (NEWCAS 2021), (Toulon (virtual), France), June 2021.

G. Devic, G. Sassatelli, and A. Gamatié, "Energy-Efficient Machine Learning on FPGA for Edge Devices: a Case Study," in ComPAS: Conférence en Parallélisme, Architecture et Système, (Lyon, France), June 2020.

G. Devic, A. Gamatié, and G. Sassatelli, "Évaluation de deux architectures matérielles dédiées à l'inférence basée sur des réseaux de neurones convolutifs," in Conférence francophone d'informatique en Parallélisme, Architecture et Système (Compas'2020), (Lyon, France), June 2020.

M. Mirka, **G. Devic**, F. Bruguier, G. Sassatelli, and A. Gamatié, "Automatic Energy-Efficiency Monitoring of OpenMP Workloads," in ReCoSoC: Reconfigurable Communication-centric Systems-on-Chip, (York, United Kingdom), July 2019.

A. Gamatié, **G. Devic**, G. Sassatelli, S. Bernabovi, P. Naudin, and M. Chapman, "Towards Energy-Efficient Heterogeneous Multicore Architectures for Edge Computing, IEEE Access, vol. 7, pp. 49474–49491, Apr. 2019.

Bibliographie

[1] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey and benchmarking of machine learning accelerators," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9, 2019.

[2] J. Qiu, Q. Wu, G. Ding, Y. Xu, and S. Feng, "A survey of machine learning for big data processing," *EURASIP Journal on Advances in Signal Processing*, vol. 2016, 05 2016.

[3] A. Kienhuis, *Design space exploration of stream-based dataflow architectures - methods and tools.* PhD thesis, Delft University of Technology, 1999.

[4] "Xilinx University Program/Vivado." [en ligne] consulté le 06/07/2021 — https://www.xilinx.com/support/university/vivado.html.

[5] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "A near-threshold RISC-V core with DSP extensions for scalable iot endpoint devices," *CoRR*, vol. abs/1608.08376, 2016.

[6] A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, and L. Benini, "Xpulpnn: Accelerating quantized neural networks on risc-v processors through isa extensions," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 186–191, 2020.

[7] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: hardware-aware automated quantization," *CoRR*, vol. abs/1811.08886, 2018.

[8] O. Bonnaud, "The challenges of microelectronics for the future digital society: The roles of thin film technologies and of the higher education," *Journal of Materials Science and Chemical Engineering*, vol. 07, pp. 47–56, 01 2019.

[9] S. Verma, Y. Kawamoto, Z. M. Fadlullah, H. Nishiyama, and N. Kato, "A survey on network methodologies for real-time analytics of massive iot data and open research issues," *IEEE Communications Surveys Tutorials*, vol. 19, no. 3, pp. 1457–1477, 2017.

[10] H. Li, K. Ota, and M. Dong, "Learning iot in edge: Deep learning for the internet of things with edge computing," *IEEE Network*, vol. 32, no. 1, pp. 96–101, 2018.

[11] D. Georgakopoulos, P. P. Jayaraman, M. Fazia, M. Villari, and R. Ranjan, "Internet of things and edge cloud computing roadmap for manufacturing," *IEEE Cloud Computing*, vol. 3, pp. 66–73, July 2016.

[12] N. Verma, H. Jia, H. Valavi, Y. Tang, M. Ozatay, L.-Y. Chen, B. Zhang, and P. Deaville, "In-memory computing: Advances and prospects," *IEEE Solid-State Circuits Magazine*, vol. 11, no. 3, pp. 43–55, 2019.

[13] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware," *CoRR*, vol. abs/1705.06963, 2017.

[14] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015.

[15] J. Qiu, Q. Wu, G. Ding, Y. Xu, and S. Feng, "A survey of machine learning for big data processing," *EURASIP Journal on Advances in Signal Processing*, vol. 2016, p. 67, May 2016.

[16] L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, *Scaling Learning Algorithms toward AI*, pp. 321–359. 2007.

[17] T. Yang and V. Sze, "Design considerations for efficient deep neural networks on processing-in-memory accelerators," *CoRR*, vol. abs/1912.12167, 2019.

[18] S. Dong, P. Wang, and K. Abbas, "A survey on deep learning and its applications," *Computer Science Review*, vol. 40, p. 100379, 2021.

[19] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[21] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[23] A. Z. Karen Simonyan, "Very deep convolutional networks for large-scale image recognition," *iclr*, 2015.

[24] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in Neural Information Processing Systems* (D. Touretzky, ed.), vol. 2, Morgan-Kaufmann, 1990.

[25] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. Lecun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *International Conference on Learning Representations (ICLR) (Banff)*, 12 2013.

[26] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018.

[27] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size," *arXiv:1602.07360*, 2016.

[28] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," *CoRR*, vol. abs/1707.01083, 2017.

[29] "The mnist database of handwritten digits." [en ligne] consulté le 22/07/2021 — http://yann.lecun.com/exdb/mnist/.

[30] Y. Le Cun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov 1998.

[31] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *Proceedings of the International Conference on Learning Representations*, 2017.

[32] E. Cai, D. Juan, D. Stamoulis, and D. Marculescu, "Neuralpower: Predict and deploy energy-efficient convolutional neural networks," *CoRR*, vol. abs/1710.05420, 2017.

[33] D. Stamoulis, E. Cai, D. Juan, and D. Marculescu, "Hyperpower: Power- and memory-constrained hyper-parameter optimization for neural networks," *CoRR*, vol. abs/1712.02446, 2017.

[34] S. Vadera and S. Ameen, "Methods for pruning deep neural networks," *CoRR*, vol. abs/2011.00241, 2020.

[35] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *CoRR*, vol. abs/1506.02626, 2015.

[36] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *Proceedings of the 32nd International Conference on Machine Learning* (F. Bach and D. Blei, eds.), vol. 37 of *Proceedings of Machine Learning Research*, (Lille, France), pp. 2285–2294, PMLR, 07–09 Jul 2015.

[37] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2015. Published as a conference paper at ICLR 2016 (oral).

[38] D. Mittal, S. Bhardwaj, M. M. Khapra, and B. Ravindran, "Studying the plasticity in deep convolutional neural networks using random pruning," *CoRR*, vol. abs/1812.10240, 2018.

[39] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[40] "The Arm Cortex-M0 processor is the smallest Arm processor available.." [en ligne] consulté le 02/09/2021 — https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0.

[41] "The Arm Cortex-M1 processor targets FPGA devices.." [en ligne] consulté le 02/09/2021 — https://developer.arm.com/ip-products/processors/cortex-m/cortex-m1.

[42] "The Arm Cortex-M3 processor is the industry-leading 32-bit processor for highly deterministic real-time applications.." [en ligne] consulté le 02/09/2021 — https://developer.arm.com/ip-products/processors/cortex-m/cortex-m3.

[43] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, (New York, NY, USA), p. 26–35, Association for Computing Machinery, 2016.

[44] B. Moons, K. Goetschalckx, N. V. Berckelaer, and M. Verhelst, "Minimum energy quantized neural networks," *CoRR*, vol. abs/1711.00215, 2017.

[45] Q. Jin, L. Yang, and Z. Liao, "Adabits: Neural network quantization with adaptive bit-widths," *CoRR*, vol. abs/1912.09666, 2019.

[46] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," *CoRR*, vol. abs/1602.01528, 2016.

[47] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, 2014.

[48] A. Dundar, J. Jin, V. Gokhale, B. Martini, and E. Culurciello, "Memory access optimized routing scheme for deep networks on a mobile coprocessor," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, 2014.

[49] "Gap8 hardware reference manual." [en ligne] consulté le 22/07/2021 — https://gwt-website-files.s3.amazonaws.com/gap8_datasheet.pdf.

[50] "Esperanto Technologies to Reveal Chip with 1000+ Cores at RISC-V Summit." [en ligne] consulté le 15/08/2021 — https://www.esperanto.ai/esperanto-technologies-to-reveal-chip-with-1000-cores-at-risc-v-summit/.

[51] "DSP for Cortex-M." [en ligne] consulté le 31/07/2021 — https://developer.arm.com/architectures/instruction-sets/dsp-extensions/dsp-for-cortex-m.

[52] T. Adegbija, A. Rogacs, C. Patel, and A. Gordon-Ross, "Microprocessor optimizations for the internet of things: A survey," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, pp. 7–20, Jan 2018.

[53] "Technologies arm big.little." [en ligne] consulté le 06/07/2021 — https://www.arm.com/why-arm/technologies/big-little.

[54] Y. Lu, L. Liu, J. Zhu, S. Yin, and S. Wei, "Architecture, challenges and applications of dynamic reconfigurable computing," *Journal of Semiconductors*, vol. 41, p. 021401, feb 2020.

[55] I. R. Quadri, H. Yu, A. Gamatié, É. Rutten, S. Meftali, and J. Dekeyser, "Targeting reconfigurable FPGA based socs using the UML MARTE profile: from high abstraction levels to code generation," *Int. J. Embed. Syst.*, vol. 4, no. 3/4, pp. 204–224, 2010.

[56] D. J. M. Moss, E. Nurvitadhi, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. W. Leong, "High performance binary neural networks on the xeon+fpga™ platform," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, 2017.

[57] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, (New York, NY, USA), p. 26–35, Association for Computing Machinery, 2016.

[58] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 268–281, 2020.

[59] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: A framework for mapping convolutional neural networks on fpgas," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 40–47, 2016.

[60] D. Moolchandani, A. Kumar, and S. R. Sarangi, "Accelerating cnn inference on asics: A survey," *Journal of Systems Architecture*, vol. 113, p. 101887, 2021.

[61] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "14.2 dnpu: An 8.1tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 240–241, 2017.

[62] K. Ueyoshi, K. Ando, K. Hirose, S. Takamaeda-Yamazaki, M. Hamada, T. Kuroda, and M. Motomura, "Quest: Multi-purpose log-quantized dnn inference engine stacked on 96-mb 3-d sram using inductive coupling technology in 40-nm cmos," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 1, pp. 186–196, 2019.

[63] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[64] Y. Chen, J. S. Emer, and V. Sze, "Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks," *CoRR*, vol. abs/1807.07928, 2018.

[65] S. Yin, P. Ouyang, S. Tang, F. Tu, X. Li, L. Liu, and S. Wei, "A 1.06-to-5.09 tops/w reconfigurable hybrid-neural-network processor for deep learning applications," in *2017 Symposium on VLSI Circuits*, pp. C26–C27, 2017.

[66] B. Moons and M. Verhelst, "A 0.3–2.6 tops/w precision-scalable processor for real-time large-scale convnets," in *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pp. 1–2, 2016.

[67] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 246–247, 2017.

[68] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "Unpu: A 50.6tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pp. 218–220, 2018.

[69] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," *CoRR*, vol. abs/1512.04295, 2015.

[70] K. Ando, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, M. Ikebe, T. Asai, S. Takamaeda-Yamazaki, T. Kuroda, and M. Motomura, "Brein memory: A 13-layer 4.2 k neuron/0.8 m synapse binary/ternary reconfigurable in-memory deep neural network accelerator in 65 nm cmos," in *2017 Symposium on VLSI Circuits*, pp. C24–C25, 2017.

[71] J. Sim, J.-S. Park, M. Kim, D. Bae, Y. Choi, and L.-S. Kim, "14.6 a 1.42tops/w deep convolutional neural network recognition processor for intelligent ioe systems," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 264–265, 2016.

[72] P.-H. Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun, and E. Culurciello, "Neuflow: Dataflow vision processing system-on-a-chip," in *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 1044–1047, 2012.

[73] Z. Yuan, Y. Yang, J. Yue, R. Liu, X. Feng, Z. Lin, X. Wu, X. Li, H. Yang, and Y. Liu, "14.2 a 65nm 24.7µj/frame 12.3mw activation-similarity-aware convolutional neural network video processor using hybrid precision, inter-frame data reuse and mixed-bit-width difference-frame data codec," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 232–234, 2020.

[74] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," *SIGARCH Comput. Archit. News*, vol. 43, p. 92–104, June 2015.

[75] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *SIGARCH Comput. Archit. News*, vol. 42, p. 269–284, Feb. 2014.

[76] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "Circnn: Accelerating and compressing deep neural networks using block-circulantweight matrices," *CoRR*, vol. abs/1708.08917, 2017.

[77] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[78] G. Desoli, N. Chawla, T. Boesch, S.-p. Singh, E. Guidetti, F. De Ambroggi, T. Majo, P. Zambotti, M. Ayodhyawasi, H. Singh, and N. Aggarwal, "14.1 a 2.9tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 238–239, 2017.

[79] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks," *CoRR*, vol. abs/1712.01507, 2017.

[80] S. Sharify, A. D. Lascorz, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," *CoRR*, vol. abs/1706.07853, 2017.

[81] S. Yin, P. Ouyang, S. Tang, F. Tu, X. Li, L. Liu, and S. Wei, "A 1.06-to-5.09 tops/w reconfigurable hybrid-neural-network processor for deep learning applications," *2017 Symposium on VLSI Circuits*, pp. C26–C27, 2017.

[82] J. Yue, R. Liu, W. Sun, Z. Yuan, Z. Wang, Y.-N. Tu, Y.-J. Chen, A. Ren, Y. Wang, M.-F. Chang, X. Li, H. Yang, and Y. Liu, "7.5 a 65nm 0.39-to-140.3tops/w 1-to-12b unified neural network processor using block-circulant-enabled transpose-domain acceleration with 8.1 × higher tops/mm2and 6t hbst-tram-based 2d data-reuse architecture," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 138–140, 2019.

[83] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA," *CoRR*, vol. abs/1808.04311, 2018.

[84] P. Colangelo, N. Nasiri, A. K. Mishra, E. Nurvitadhi, M. Margala, and K. Nealis, "Exploration of low numeric precision deep learning inference using intel fpgas," *CoRR*, vol. abs/1806.11547, 2018.

[85] V. Camus, L. Mei, C. Enz, and M. Verhelst, "Review and benchmarking of precision-scalable multiply-accumulate unit architectures for embedded neural-network processing," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 4, pp. 697–711, 2019.

[86] A. D. Pimentel, "Exploring exploration: A tutorial introduction to embedded systems design space exploration," *IEEE Des. Test*, vol. 34, no. 1, pp. 77–90, 2017.

[87] L. Apvrille and A. Bécoulet, "Prototyping an Embedded Automotive System from its UML/SysML Models," in *Embedded Real Time Software and Systems (ERTS2012)*, (Toulouse, France), Feb. 2012.

[88] J.-L. Dekeyser, A. Gamatié, A. Etien, R. Ben Atitallah, and P. Boulet, "Using the UML Profile for MARTE to MPSoC Co-Design," in *First International Conference on Embedded Systems Critical Applications (ICESCA'08)*, (Tunis, Tunisia), May 2008.

[89] A. Gamatié, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser, "A model-driven design framework for massively parallel embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 10, Nov. 2011.

[90] I. R. Quadri, A. Gamatié, P. Boulet, and J.-L. Dekeyser, "Modeling of Configurations for Embedded System Implementations in MARTE," in *1st workshop on Model Based Engineering for Embedded Systems Design - Design, Automation and Test in Europe (DATE 2010)*, (Dresden, Germany), Mar. 2010.

[91] H. Yu, A. Gamatié, É. Rutten, and J. Dekeyser, "Safe design of high-performance embedded systems in an MDE framework," *Innov. Syst. Softw. Eng.*, vol. 4, no. 3, pp. 215–222, 2008.

[92] E. Senn, D. Chillet, O. Zendra, C. Belleudy, S. B. Bilavarn, R. B. Atitallah, C. Samoyeau, and A. Fritsch, "Open-people: Open power and energy optimization platform and estimator," in *2012 15th Euromicro Conference on Digital System Design*, pp. 668–675, 2012.

[93] M. Breuer, A. Friedman, and A. Iosupovicz, "A survey of the state of the art of design automation," *Computer*, vol. 14, pp. 58–75, oct 1981.

[94] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 249–261, 2007.

[95] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, p. 1–7, Aug. 2011.

[96] A. Butko, A. Gamatié, G. Sassatelli, L. Torres, and M. Robert, "Design Exploration for next Generation High-Performance Manycore On-chip Systems: Application to big.LITTLE Architectures," in *ISVLSI: International Symposium on Very Large Scale Integration*, (Montpellier, France), pp. 551–556, IEEE, July 2015.

[97] A. Butko, R. Garibotti, L. Ost, V. Lapotre, A. Gamatie, G. Sassatelli, and C. Adeniyi-Jones, "A trace-driven approach for fast and accurate simulation of manycore architectures," in *The 20th Asia and South Pacific Design Automation Conference*, pp. 707–712, 2015.

[98] A. Butko, F. Bruguier, A. Gamatié, G. Sassatelli, D. Novo, L. Torres, and M. Robert, "Full-system simulation of big.little multicore architecture for performance and energy exploration," in *2016 IEEE 10th International Symposium on Embedded Multicore/Manycore Systems-on-Chip (MCSOC)*, pp. 201–208, 2016.

[99] A. Nocua, F. Bruguier, G. Sassatelli, and A. Gamatié, "Elasticsimmate: A fast and accurate gem5 trace-driven simulator for multicore systems," in *12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC 2017, Madrid, Spain, July 12-14, 2017*, pp. 1–8, IEEE, 2017.

[100] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer US, 2006.

[101] K. Latif, C. E. Effiong, A. Gamatié, G. Sassatelli, L. B. Zordan, L. Ost, P. Dziurzanski, and L. Soares Indrusiak, "An Integrated Framework for Model-Based Design and Analysis of Automotive Multi-Core Systems," in *FDL: Forum on specification Design Languages*, Work-in-Progress Session, (Barcelona, Spain), Sept. 2015.

[102] K. Latif, M. Selva, C. Effiong, R. Ursu, A. Gamatie, G. Sassatelli, L. Zordan, L. Ost, P. Dziurzanski, and L. S. Indrusiak, "Design space exploration for complex automotive applications: An engine control system case study," in *Proceedings of the 2016 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[103] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel simulation of systemc tlm 2.0 compliant mpsoc on smp workstations," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pp. 606–609, 2010.

[104] G. Schirner and R. Dömer, "Quantitative analysis of the speed/accuracy trade-off in transaction level modeling," *ACM Trans. Embed. Comput. Syst.*, vol. 8, Jan. 2009.

[105] X. An, A. Gamatié, and E. Rutten, "High-level design space exploration for adaptive applications on multiprocessor systems-on-chip," *Journal of Systems Architecture*, vol. 61, no. 3, pp. 172–184, 2015.

[106] X. An, S. Boumedien, A. Gamatié, and E. Rutten, "Classy: A clock analysis system for rapid prototyping of embedded applications on mpsocs," in *Proceedings of the 15th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '12, (New York, NY, USA), p. 3–12, Association for Computing Machinery, 2012.

[107] G. V. Caliri, "Introduction to analytical modeling," in *Int. CMG Conference*, pp. 31–36, Citeseer, 2000.

[108] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 304–315, 2019.

[109] G. Devic, A. Gamatié, and G. Sassatelli, "Évaluation de deux architectures matérielles dédiées à l'inférence basée sur des réseaux de neurones convolutifs," in *Conférence francophone d'informatique en Parallélisme, Architecture et Système (Compas'2020)*, (Lyon, France), June 2020.

[110] A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[111] T. Delobelle, P.-Y. Péneau, A. Gamatié, F. Bruguier, S. Senni, G. Sassatelli, and L. Torres, "Magpie: System-level evaluation of manycore systems with emerging memory technologies," in *2nd International Workshop on Emerging Memory Solutions - Technology, Manufacturing, Architectures, Design and Test at Design Automation and Test in Europe (DATE'2017), Lausanne, Switzerland*, March 2017.

[112] E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini, "Gap-8: A risc-v soc for ai at the edge of the iot," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 1–4, July 2018.

[113] "What is mendel linux?." [en ligne] consulté le 22/07/2021 — https://coral.googlesource.com/docs/+/refs/heads/master/ReadMe.md.

[114] "Edge tpu: Puce asic conçue sur mesure par google pour exécuter des inférences en périphérie." [en ligne] consulté le 22/07/2021 —https://cloud.google.com/edge-tpu/.

[115] "Cloud tpu: L'entraînement et l'exécution des modèles de machine learning n'ont jamais été aussi rapides." [en ligne] consulté le 22/07/2021 — https://cloud.google.com/tpu.

[116] "Matériel d'acquisition de données multifonction." [en ligne] consulté le 22/07/2021 —https://www.ni.com/fr-fr/support/model.usb-6009.html.

[117] J. C. R. da Silva, F. M. Q. Pereira, M. Frank, and A. Gamatié, "A compiler-centric infra-structure for whole-board energy measurement on heterogeneous android systems,"

in *13th Int'l Symp. on Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC'18, Lille, France, July 9-11, 2018* (S. Niar and M. A. R. Saghir, eds.), pp. 1–8, IEEE, 2018.

[118] A. Gamatié, G. Devic, G. Sassatelli, S. Bernabovi, P. Naudin, and M. Chapman, "Towards Energy-Efficient Heterogeneous Multicore Architectures for Edge Computing," *IEEE Access*, vol. 7, pp. 49474–49491, Apr. 2019.

[119] G. Devic, G. Sassatelli, and A. Gamatié, "Energy-Efficient Machine Learning on FPGA for Edge Devices: a Case Study," in *ComPAS: Conférence en Parallélisme, Architecture et Système*, (Lyon, France), June 2020.

[120] "Cortus – soc / asic design," March 2021. [online] https://www.cortus.com/.

[121] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*. O'Reilly & Associates, Inc., 1996.

[122] "Genesys 2 Kintex-7 FPGA Development Board." https://www.xilinx.com/products/boards-and-kits/1-cfdwjq.html, July 2017.

[123] "Xilinx Virtex UltraScale FPGA VCU108 Evaluation Kit." https://www.xilinx.com/products/boards-and-kits/ek-u1-vcu108-g.html, July 2017.

[124] "7 Series Product Selection Guide." [en ligne] consulté le 31/07/2021 — https://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf.

[125] "Ultrascale FPGA Product Selection Guide." [en ligne] consulté le 31/07/2021 — https://www.xilinx.com/support/documentation/selection-guides/ultrascale-fpga-product-selection-guide.pdf.

[126] J. C. R. da Silva, F. M. Q. Pereira, M. Frank, and A. Gamatié, "A compiler-centric infra-structure for whole-board energy measurement on heterogeneous android systems," in *13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC 2018, Lille, France, July 9-11, 2018*, pp. 1–8, 2018.

[127] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," (Brussels, Belgium), pp. 137–147, OCG, July 2010.

[128] L. Ma, L. Lavagno, M. T. Lazarescu, and A. Arif, "Acceleration by inline cache for memory-intensive algorithms on fpga via high-level synthesis," *IEEE Access*, vol. 5, pp. 18953–18974, 2017.

[129] N. S. Mokhtari, *Performance Optimization of Memory-Bound Programs on Data Parallel Accelerators*. PhD thesis, The Ohio State University, 2016.

[130] M. Mirka, G. Devic, F. Bruguier, G. Sassatelli, and A. Gamatié, "Automatic energy-efficiency monitoring of openmp workloads," in *14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC 2019, York, United Kingdom, July 1-3, 2019*, pp. 43–50, IEEE, 2019.

[131] M. Mirka, G. Sassatelli, and A. Gamatié, "Online learning for dynamic control of openmp workloads," in *9th International Conference on Modern Circuits and Systems Technologies, MOCAST 2020, Bremen, Germany, September 7-9, 2020*, pp. 1–6, IEEE, 2020.

[132] M. Novaes, V. Petrucci, A. Gamatié, and F. M. Q. Pereira, "Compiler-assisted adaptive program scheduling in big.little systems: poster," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019* (J. K. Hollingsworth and I. Keidar, eds.), pp. 429–430, ACM, 2019.

[133] M. Novaes, V. Petrucci, A. Gamatié, and F. M. Q. Pereira, "Compiler-assisted adaptive program scheduling in big.little systems," *CoRR*, vol. abs/1903.07038, 2019.

[134] A. Gamatié, X. An, Y. Zhang, A. Kang, and G. Sassatelli, "Empirical model-based performance prediction for application mapping on multicore architectures," *J. Syst. Archit.*, vol. 98, pp. 1–16, 2019.

[135] "The GNU Compiler Collection." [en ligne] consulté le 06/07/2021 — https://gcc.gnu.org/.

[136] "Options That Control Optimization of GCC." [en ligne] consulté le 06/07/2021 — https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[137] A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[138] D. Stamoulis, E. Cai, D. Juan, and D. Marculescu, "Hyperpower: Power- and memory-constrained hyper-parameter optimization for neural networks," *CoRR*, vol. abs/1712.02446, 2017.

[139] G. Devic, M. France pillois, J. Salles, G. Sassatelli, and A. Gamatié, "Highly-Adaptive Mixed-Precision MAC Unit for Smart and Low-Power Edge Computing," in *19th IEEE Interregional NEWCAS Conference (NEWCAS 2021)*, (Toulon (virtual), France), June 2021.

[140] "Page Github du coeur cv32e40p (RI5CY)." [en ligne] consulté le 20/08/2021 — https://github.com/openhwgroup/cv32e40p.

[141] Y. N. Wu, J. S. Emer, and V. Sze, "Accelergy: An architecture-level energy estimation methodology for accelerator designs," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2019.

[142] "Coral Dev Board : A development board to quickly prototype on-device ML products.." [en ligne] consulté le 06/07/2021 — https://coral.ai/products/dev-board/.

[143] "GAPuino development board." [en ligne] consulté le 06/07/2021 — https://greenwaves-technologies.com/product/gapuino/.

[144] S. Senni, L. Torres, G. Sassatelli, A. Gamatie, and B. Mussard, "Emerging non-volatile memory technologies exploration flow for processor architecture," in *2015 IEEE Computer Society Annual Symposium on VLSI*, pp. 460–460, 2015.

[145] S. Senni, T. Delobelle, O. Coi, P.-Y. Peneau, L. Torres, A. Gamatie, P. Benoit, and G. Sassatelli, "Embedded systems to high performance computing using stt-mram," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 536–541, 2017.

[146] P.-Y. Péneau, D. Novo, F. Bruguier, L. Torres, G. Sassatelli, and A. Gamatié, "Improving the Performance of STT-MRAM LLC through Enhanced Cache Replacement Policy," in *ARCS: Architecture of Computing Systems*, vol. LNCS, (Braunschweig, Germany), pp. 168–180, Apr. 2018.

[147] P.-Y. Péneau, R. Bouziane, A. Gamatié, E. Rohou, F. Bruguier, G. Sassatelli, L. Torres, and S. Senni, "Loop optimization in presence of stt-mram caches: A study of performance-energy tradeoffs," in *2016 26th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pp. 162–169, 2016.

[148] R. Bouziane, E. Rohou, and A. Gamatié, "Energy-efficient memory mappings based on partial wcet analysis and multi-retention time stt-ram," in *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, RTNS '18, (New York, NY, USA), p. 148–158, Association for Computing Machinery, 2018.

[149] A. Gamatié, A. Nocua, J. W. Weloli, G. Sassatelli, L. Torres, D. Novo, and M. Robert, "Emerging NVM Technologies in Main Memory for Energy-Efficient HPC: an Empirical Study." working paper or preprint, May 2019.

[150] F. M. Q. a. Pereira, G. V. Leobas, and A. Gamatié, "Static prediction of silent stores," *ACM Trans. Archit. Code Optim.*, vol. 15, Nov. 2018.

[151] R. Bouziane, E. Rohou, and A. Gamatié, "Compile-time silent-store elimination for energy efficiency: An analytic evaluation for non-volatile cache memory," in *Proceedings of the Rapido'18 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[152] P. Siegl, R. Buchty, and M. Berekovic, "Data-centric computing frontiers: A survey on processing-in-memory," in *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, (New York, NY, USA), p. 295–308, Association for Computing Machinery, 2016.

[153] S. Yin, B. Zhang, M. Kim, J. Saikia, S. Kwon, S. Myung, H. Kim, S. J. Kim, M. Seok, and J.-S. Seo, "Pimca: A 3.4-mb programmable in-memory computing accelerator in 28nm for on-chip dnn inference," in *2021 Symposium on VLSI Technology*, pp. 1–2, 2021.

[154] S. Yin, Y. Kim, X. Han, H. Barnaby, S. Yu, Y. Luo, W. He, X. Sun, J.-J. Kim, and J.-s. Seo, "Monolithically integrated rram- and cmos-based in-memory computing optimizations for efficient deep learning," *IEEE Micro*, vol. 39, no. 6, pp. 54–63, 2019.

[155] J. Zhang, Z. Wang, and N. Verma, "In-memory computation of a machine-learning classifier in a standard 6t sram array," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 915–924, 2017.