BRICOUT Thaddée
Électronique et Informatique des systèmes embarqués (IESE)
POLYTECH GRENOBLE

Rapport de stage 2022

Evaluation de l'inférence d'un réseau de neurones sur une architecture C-SRAM (Computing SRAM)

Tome Principal
Et
Annexes

Année universitaire 2021-2022
Stage du 4 avril au 2 septembre 2022

I. Remerciements

Ce stage de fin d'études a été effectué dans l'équipe mémoire du laboratoire LFIM du CEA Grenoble. Je souhaite donc remercier toute l'équipe pour son accueil ainsi que pour les nombreux conseils apportés tout au long du stage ; en particulier mon Tuteur, Henri-Pierre CHARLES, qui m'a guidé afin de mener à bien ce projet tout au long du stage, ainsi que Kevin MAMBU son Thésard pour tous les conseils techniques tout au long du stage.

Je souhaite également remercier mon Enseignant référent, Frédéric ROUSSEAU qui m'a rendu visite et suivi tout au long de ce stage pour s'assurer de son bon déroulement.

II. Table des matières

I. Remerciements 1

II. Table des matières 2

III. Table des figures 3

IV. Glossaire 4

V. Le CEA 5

1. Présentation générale 5

2. L'environnement de mon stage 6

VI. Contexte 7

1. La C-SRAM 8

2. Les réseaux de neurones 9

1. L'apprentissage 10
2. L'inférence 11

VII. Environnement / outils 12

1. Hybrogen et Hybrolang 12

2. QEMU 13

3. TensorFlow Lite pour microcontrôleur 14

VIII. Travail réalisé 15

1. Prise en main de la C-SRAM 15

2. Analyse de TensorFlow 16

3. Profilage d'exemples 17

4. Implémentation de la convolution optimisée 18

1. Analyse de la convolution scalaire 18
2. Organisation de la convolution optimisée 20
3. Flow de développement de la convolution optimisée 21

5. Performance de la convolution scalaire et C-SRAM 22

1. Performance de la convolution 22
2. Optimisation potentielle 25

6. Ajout de la convolution dans TFLM 26

1. Ajout de la convolution. 26
2. Compilation croisée pour processeur RISCV 26

IX. Déroulement du stage 28

X. Compétences ingénieur IESE 29

1. Situation professionnelle métier ____ 29

2. Situation professionnelle transversale ____ 29

XI. Conclusion et perspectives ____ *31*

1. Conclusion ____ 31

2. Perspectives ____ 31

3. Contributions ____ 32

XII. Références ____ *33*

1. Bibliographie ____ 33

2. Sources ____ 33

3. Lien logiciel et contributions ____ 33

XIII. Annexes ____ *34*

III. Table des figures

Figure 1: Carte des centres CEA en France 6

Figure 2: Organigramme du laboratoire LFIM 6

Figure 3: Représentation du "Memory Wall" 7

Figure 4: Hiérarchie mémoire classique 7

Figure 5: Comparaisons des architectures scalaire, vectorielle et C-SRAM 8

Figure 6: Architecture d'un réseau de neurones 9

Figure 7: Description d'un neurone 10

Figure 8: Architecture standard d'un réseau de neurones à convolutions 10

Figure 9: Langage de programmation Hybrolang 12

Figure 10: Schéma de fonctionnement de Hybrogen 13

Figure 11: Vue d'ensemble de TensorFlow 14

Figure 12: Architecture de la bibliothèque C++ de TensorFlow Lite 16

Figure 13: Extrais du rapport Gprof pour les 3 exemples 18

Figure 14: Calcul de la convolution par canal 19

Figure 15: Organisation des données dans la C-SRAM pour le calcul de convolution 20

Figure 16: Gain en énergie et en vitesse de la convolution optimisé par rapport à la convolution scalaire en fonction de l'optimisation utilisée lors de la compilation GCC (Os et O3) 24

Figure 17: Organisation temporelle du stage 28

IV. Glossaire

CEA: Commissariat à l'Energie Atomique et aux Energies Alternatives.

IMC : In Memory Computing

NMC : Near Memory Computing

DRT : Direction de la recherche technologique

DRF : Direction de la recherche fondamentale

DEN : Direction des énergies

DAM : direction des applications militaires

CPU : Central Processing Unit

RAM : Random Access Memory

SRAM : Static RAM

DRAM : Dynamic RAM

ALU : Unité Arithmétique et Logique

IA : Intelligence artificielle

TF : TensorFlow

TFLM : TensorFlow Lite pour microcontrôleur

API : Application Programming Interface

MAC: multiplication accumulation

V. Le CEA

1. Présentation générale

Le Commissariat à l'énergie atomique et aux énergies alternatives (CEA) est un établissement public de recherche à caractère scientifique, technique et industriel créé en 1945. Acteur majeur de la recherche, du développement et de l'innovation dans le monde, le CEA est aujourd'hui structuré autour de quatre grandes missions :

- **La défense et la sécurité**, regroupées sous la direction des applications militaires (DRT). Le CEA a pour mission de concevoir, fabriquer, maintenir en condition opérationnelle, puis démanteler les têtes nucléaires qui équipent les forces aéroportées et océaniques françaises. Le CEA apporte aussi son expertise dans le domaine de l'armement conventionnel.
- **Les énergies bas carbone (nucléaires et renouvelables)**, regroupées sous la direction des énergies (DEN). Sa mission est d'apporter au pouvoir public et aux industriels les éléments d'expertise et d'innovation pour la mise en œuvre d'un système énergétique bas carbone.
- **La recherche fondamentale**, regroupée sous la direction de la recherche fondamentale (DRF). Sa mission est d'œuvrer pour la recherche dans les domaines des biotechnologies, des sciences de la matière et de l'univers, la santé, la physique et des nanosciences.
- **La recherche technologique pour l'industrie**, regroupée sous la direction de la recherche technologique (DRT). Le CEA participe au redressement industriel du pays à travers la valorisation et le transfert des connaissances, des compétences et des technologies vers l'industrie. Entrent dans ce champ, les technologies pour les énergies renouvelables, les technologies de l'information et les technologies pour la santé.

Le CEA se compose de 10 centres de recherche ainsi que de 6 plateformes régionales de transferts technologiques dans toute la France. L'ensemble de ces sites (Figure 1) emploie plus de 20 000 personnes.

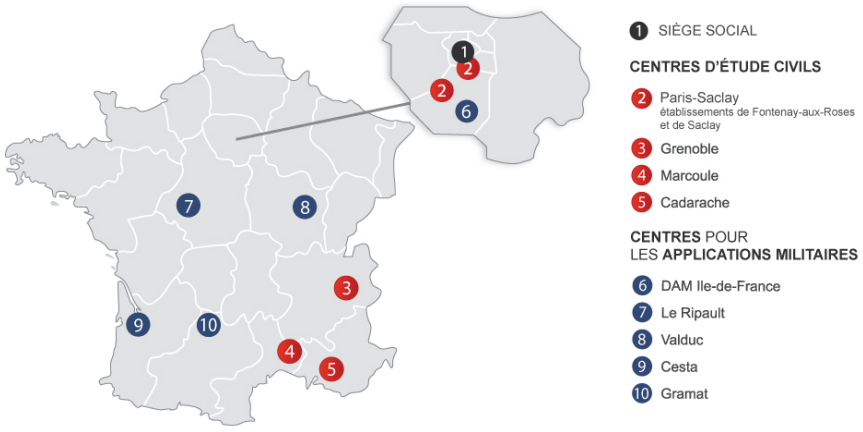


Figure 1: Carte des centres CEA en France

2. L'environnement de mon stage

Mon stage se déroule dans le centre de Grenoble. Dans ce centre, les 3 instituts de la DRT sont présents :

- **Le LITEN**, créé en 2005, travaille sur la réalisation des technologies des nanomatériaux et de l'énergie.
- **Le LETI**, créé en 1967, est spécialisé dans l'étude des micros et nanotechnologies et la manière de les intégrer dans des systèmes plus complexes.
- **Le LIST**, créé en 2003, est le laboratoire d'intégration des systèmes et des technologies. Il est spécialisé dans les systèmes numériques intelligents.

Mon stage s'est déroulé dans l'un des 5 départements de l'institut LIST, le **DSCIN** (Département des systèmes et circuits intégrés numériques) qui est décomposé en 6 laboratoires.

Le laboratoire dans lequel j'effectue mon stage est le **LFIM** (Laboratoire des Fonctions Innovantes pour les circuits Mixtes). Les missions du laboratoire se focalisent sur l'étude et la conception des circuits mixtes et numériques, pour répondre aux besoins des applications pour l'internet des objets, l'intelligence artificielle et des systèmes cyber-physiques.

Figure 2: Organigramme du laboratoire LFIM

VI. Contexte

Depuis les deux dernières décennies, les différences de performance entre la mémoire principale (DRAM) et le processeur ne cessent de s'accentuer (Figure 3). Cette différence grandissante, connue sous le nom de « Memory Wall », est due à un temps d'accès à la mémoire principale plutôt lent comparé à la fréquence de fonctionnement du CPU.

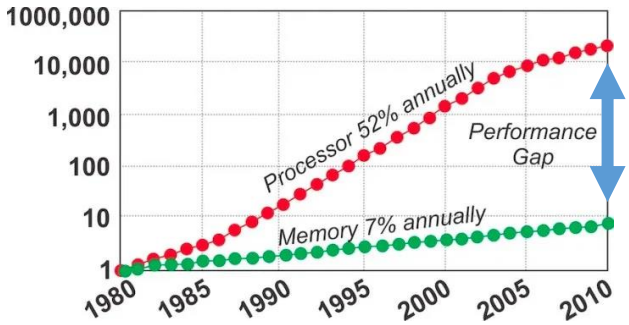
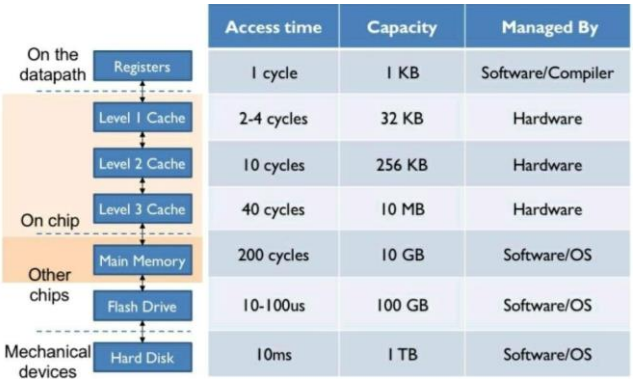


Figure 3: Représentation du "Memory Wall".
Différence de performance entre mémoires principales et processeurs

Une architecture de calcul classique, dite de Von Neumann, consiste à séparer la partie stockage (mémoire) de la partie calcul (processeur). Cela engendre une consommation et une latence importantes lors du transfert des données entre le processeur et la mémoire. Pour limiter cette latence, sur une architecture classique, on ajoute des mémoires à proximité du processeur sous forme de caches. On obtient donc la hiérarchie mémoire présentée sur la Figure 4.



Access time	Capacity	Managed By
1 cycle	1 KB	Software/Compiler
2-4 cycles	32 KB	Hardware
10 cycles	256 KB	Hardware
40 cycles	10 MB	Hardware
200 cycles	10 GB	Software/OS
10-100us	100 GB	Software/OS
10ms	1 TB	Software/OS

Figure 4: Hiérarchie mémoire classique

Le concept de calcul en mémoire et proche mémoire (IMC/NMC) est une approche différente dans laquelle on vient effectuer des tâches de calcul directement dans la mémoire, ce qui permettrait de résoudre des problèmes cités précédemment. Cette technologie, en voie de recherche, n'est pas encore disponible de manière

industrielle. L'équipe dans laquelle je travaille développe un module de calcul en mémoire autour d'une mémoire statique SRAM. Ce module est donc appelé C-SRAM pour Computing SRAM.

L'équipe qui développe ce module est composée d'un ensemble de spécialités différentes pour effectuer le design de la mémoire, développer la partie numérique qui effectue les calculs à côté de celle-ci, développer des applications afin d'évaluer les performances du module, ainsi que développer la chaine de compilations dans le but de faciliter sa programmation.

Mon stage s'inscrit dans la partie développement d'applications afin d'évaluer les performances du module. L'objectif du stage est donc d'évaluer l'inférence d'un réseau de neurones sur une architecture contenant la C-SRAM.

1. La C-SRAM

La C-SRAM est une mémoire SRAM dans laquelle a été ajoutée une unité arithmétique et logique (ALU) comme on peut le voir sur la Figure 5 en comparaison de deux architectures plus classiques.

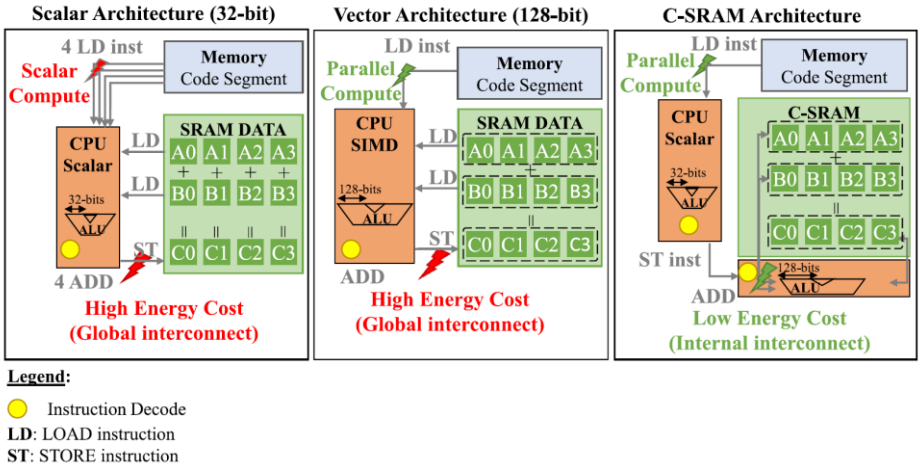


Figure 5: Comparaisons des architectures scalaire, vectorielle et C-SRAM

Donc pour la C-SRAM, le CPU joue uniquement le rôle de chef d'orchestre pour les calculs en envoyant des instructions à la mémoire qui effectuera elle-même les calculs. Cela a deux effets majeurs, le premier est que les données n'ont plus besoin d'être transférées de la mémoire vers le CPU, ce qui diminue la consommation d'énergie ainsi que la latence. L'autre effet est le fait que la C-SRAM effectue des opérations vectorielles. En effet, toutes les opérations sont effectuées sur les lignes

complètes. Les données stockées dans la mémoire peuvent être de 8, 16 ou 32 bits. La taille des données détermine la quantité qui peut être stockée sur chaque ligne et de ce fait la quantité d'opérations effectuées en parallèle. Par exemple, pour une mémoire avec une ligne de 128 bits, on effectuera, en fonction de la taille des données, 4 opérations 32 bits, 8 opérations 16 bits ou 16 opérations 8 bits. Le fait de faire plusieurs opérations en parallèles permet donc de réduire la latence. On peut également voir que la taille des données à également une grande influence sur la latence, en parallèle de la place qu'elles utilisent dans la mémoire.

2. Les réseaux de neurones

Un réseau neuronal est un ensemble de neurones qui prennent des informations en entrée et, en conjonction avec des informations provenant d'autres nœuds, développent des sorties sans règles programmées. Essentiellement, ils résolvent les problèmes par essais et erreurs. Cela fonctionne par la mise en réseau de différents neurones.

Lorsque que l'on met en entrée du réseau des données, l'ensemble des neurones du réseau vont effectuer des calculs qui lui sont associés afin de fournir une sortie qui correspond à une prédiction pour laquelle le réseau a été entrainé. On peut voir l'architecture d'un réseau de neurones sur la Figure 6.

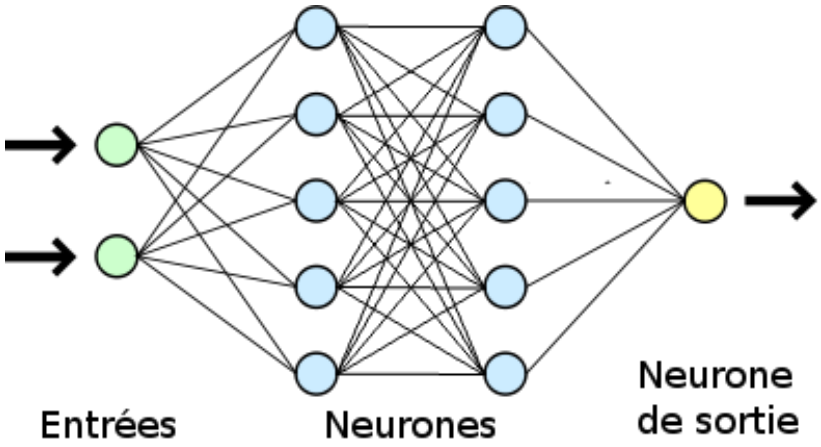


Figure 6: Architecture d'un réseau de neurones

Lorsque les données arrivent sur un neurone, elles sont ,dans un premier temps, multipliées avec un ensemble de valeurs ajustées lors de l'entrainement du réseau, que l'on appelle les poids du réseau de neurones. On ajoute ensuite l'ensemble des entrées.

Pour finir, on ajoute une fonction d'activation afin de pouvoir reformater les données pour l'entrée dans le neurone suivant (voir Figure 7).

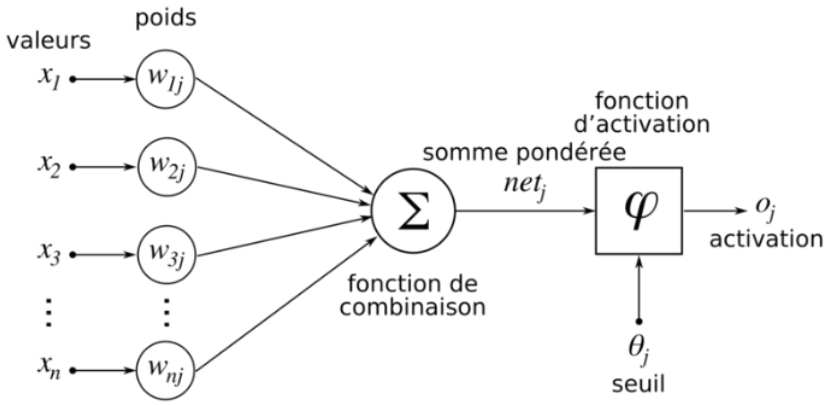


Figure 7: Description d'un neurone

L'utilisation d'un réseau de neurones se fait en deux étapes. La première est l'apprentissage. Dans cette étape, nous allons construire le réseau puis l'entrainer dans le but d'obtenir les meilleures performances pour la fonction souhaitée. La seconde est l'inférence. Cela correspond à l'utilisation du réseau de neurones. Effectuer une prédiction sur de nouvelles données. C'est donc la performance de l'exécution de l'inférence dont mon stage fait l'objet.

1. L'apprentissage

La première étape de l'apprentissage est la création du réseau. Il faut définir la suite d'opérateurs qui vont se succéder pour obtenir une sortie optimale. Comme on peut le voir sur la Figure 8, avec les opérations de convolution et de sous échantillonnages. L'ensemble de ces étapes permet d'extraire des attributs qui permettront ensuite au réseau de classifier les données.

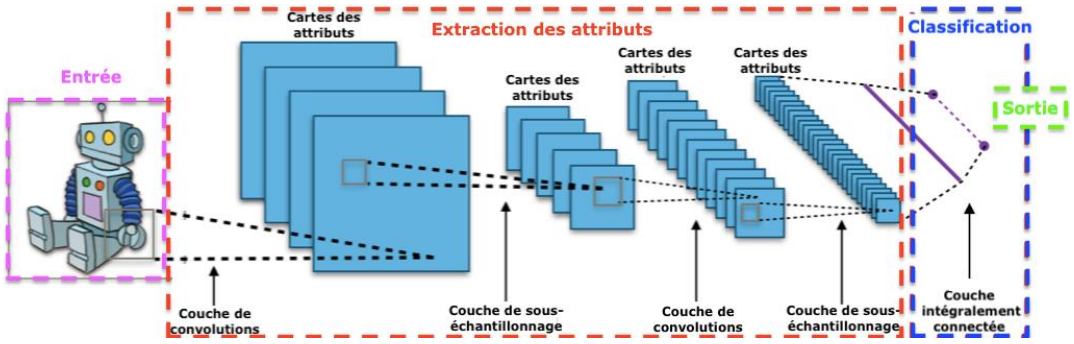


Figure 8: Architecture standard d'un réseau de neurones à convolutions

Une fois que le réseau est construit, nous avons besoins de l'entrainer afin d'extraire les meilleurs attributs et obtenir une classification optimale. Pour cela, il faut

ajuster les poids à l'entrée de chaque neurone. Pour effectuer cette opération d'ajustement, nous avons besoin de créer une base de données contenant de nombreuses données d'entrée pour notre réseau ainsi qu'une classification faite à la main en fonction de ce que l'on souhaite classifier. Nous allons ensuite classifier les données issues de la base avec le réseau. Après cette opération, on compare les résultats issus du réseau avec ceux que nous avons définis au préalable. En fonction du résultat de la comparaison, on adapte les poids en entrée de chaque neurone. On répète ensuite l'opération un grand nombre de fois pour ajuster au mieux le réseau.

2. L'inférence

Une fois que le réseau est entrainé, il n'y a plus qu'à exécuter l'inférence. C'est-à-dire effectuer des prédictions sur des nouvelles données. Pour chaque nouvelle donnée, le réseau effectue l'ensemble des calculs prévus pour la classifier. Cette opération va donc être effectuée un nombre important de fois, la performance de son exécution est donc très importante. En effet, il peut y avoir de nombreuses contraintes. En premier lieu, la vitesse d'exécution doit être plus rapide que le flot de données sur son exécution en fonction du contexte d'utilisation du réseau de neurones. Il faut par exemple que le réseau de neurones soit capable de classifier une donnée avant que la suivante arrive, On risque sinon d'obtenir une latence de plus en plus importante avant d'obtenir la classification des données. Certains contextes temps réel peuvent nécessiter que la classification s'effectue dans un temps prédéfini. Pour finir, la consommation d'énergie est également importante du fait que l'exécution peut être répétée un grand nombre de fois. D'autant plus si elle est exécutée sur un appareil fonctionnant sur batterie tel qu'un système embarqué.

Durant ce stage, je me suis donc intéressé aux performances de vitesse et de consommation de l'inférence des réseaux de neurones et notamment quel est l'impact de l'utilisation de la C-SRAM sur les performances de l'inférence.

VII. Environnement / outils

J'ai utilisé un certain nombre d'outils dont 3 principaux, le premier est un compilateur développé dans l'équipe par mon tuteur de stage, Hybrogen, ce compilateur est depuis peu open source. Il permet notamment de générer des instructions pour la C-SRAM, le deuxième est le simulateur QEMU, il permet de simuler l'exécution des instructions générées par Hybrogen sur l'architecture contenant la C-SRAM. Le dernier est le Framework d'intelligence artificielle TensorFlow Lite qui m'a permis d'exécuter l'inférence des réseaux de neurones.

1. Hybrogen et Hybrolang

Hybrogen est un compilateur dynamique. Suite à la compilation, le code sera compilé dynamiquement afin de s'adapter à l'environnement dans lequel il est exécuté. L'objectif est de compiler un code adapté et donc optimisé à son environnement, par exemple la taille des données. De plus, le jeu d'instruction de la C-SRAM est inclu dans la base de données d'Hybrogen. Cela permet donc au compilateur de générer des instructions pour la C-SRAM qui lui seront envoyées pour s'en servir comme coprocesseur au côté du RISCV.

Un langage de programmation Hybrolang, également développé par mon tuteur de stage, est utilisé comme front-end du compilateur. C'est un langage très proche du C, mais pour lequel toutes les variables sont des vecteurs dont on définit explicitement la taille comme on peut le voir sur la Figure 9.

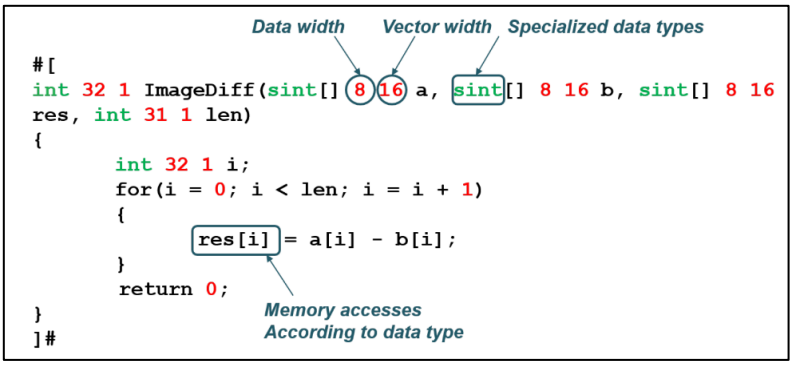


Figure 9: Langage de programmation Hybrolang

Cette manière de programmer permet de manipuler simplement la C-SRAM car ces variables permettent de manipuler des lignes complètes de la mémoire.

Hybrogen est un compilateur source à source : en entrée, nous avons des fichiers Hybrolang et en sortie, nous obtenons des fichiers C. Dans un second temps, nous compilons les fichiers C avec un compilateur classique tel que GCC. Cela permet de pouvoir mélanger des parties de code en Hybrolang avec du C dans le même fichier. En effet, seules les parties en Hybrolang seront compilées par Hybrogen et donc bénéficieront des optimisations lors de l'exécution, le reste de code sera quant à lui compilé statiquement.

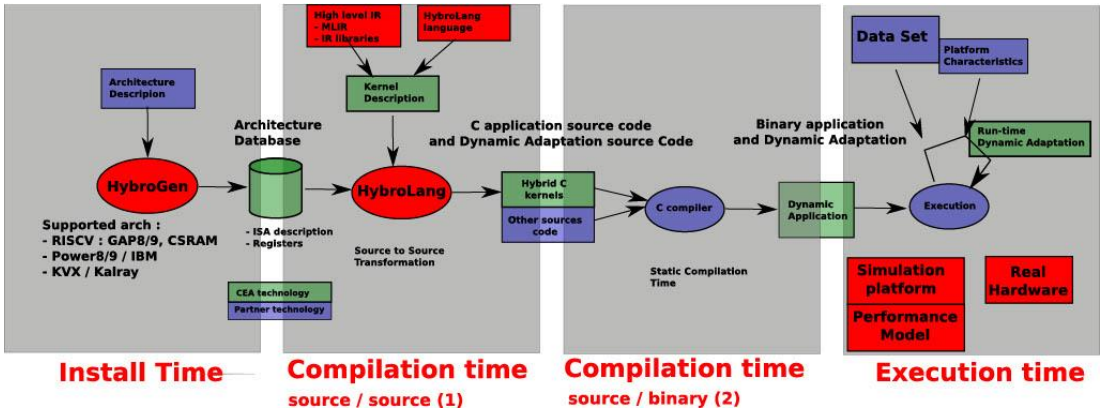


Figure 10: Schéma de fonctionnement de Hybrogen

La Figure 10 montre l'ensemble du flot de compilation d'Hyborgen avec l'installation d'une base de données contenant les différents jeux d'instruction, la compilation du code Hybrolang en code C, puis la compilation du code C en binaire exécutable. Pour finir, la compilation dynamique du code en fonction des paramètres présents au moment de l'exécution.

2. QEMU

QEMU est un simulateur d'architecture qui nous sert à simuler l'architecture contenant la C-SRAM. Le simulateur permet de placer différents modules et de les faire fonctionner entre eux. La simulation se fait au niveau des instructions. Le simulateur permet également d'obtenir des données de performance lors de la simulation des binaires. Ces données de performances dépendent évidement du modèle que l'on a défini dans le simulateur, par exemple, la présence ou non de mémoire cache au côté du processeur.

La C-SRAM modélisé dans QEMU est constituée d'une tuile de 512 lignes de mémoire SRAM. Chaque ligne fait 128 bits et peut être divisée en données de 8, 16 et 32 bits. Pour le fonctionnement de cette tuile, il y a également un processeur RISCV32,

qui envoie les instructions à la C-SRAM et effectue les calculs scalaires et les transferts de données.

3. TensorFlow Lite pour microcontrôleur

TensorFlow est un Framework d'intelligence artificielle open source développé par Google. Comme on peut le voir sur la Figure 11, TensorFlow permet, à travers de nombreux outils, d'effectuer toutes les étapes pour construire et utiliser des modèles d'intelligence artificielle. Les outils sont parfois graphiques, parfois basés sur les langages de programmation tel que le Python, C ou C++.

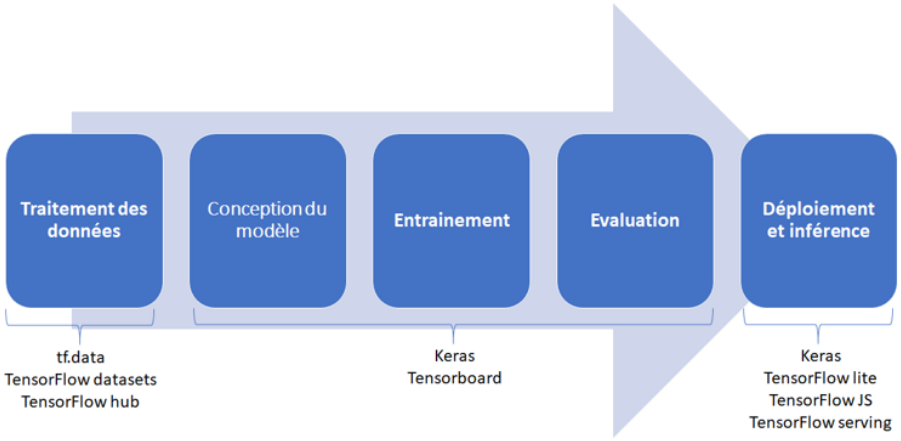


Figure 11: Vue d'ensemble de TensorFlow

Durant le stage, nous avons uniquement utilisé la TensorFlow Lite pour microcontrôleur (TFLM). Cette partie est une bibliothèque C++ adaptée au microcontrôleur. Elle est de ce fait, optimisée pour être légère, avec des opérateurs adaptés et prévus pour avoir des tailles de code réduit. C'est donc l'outil de TensorFlow qui est le plus adapté à l'architecture que nous avons, composé d'un processeur RISCV32, de la C-SRAM ainsi que des capacités mémoires très limitées.

Le fait que nous nous intéressions à l'inférence des réseaux de neurones, nous allons utiliser durant ce stage des réseaux de neurones déjà développés et entrainés avec TensorFlow. Pour cela, nous allons utiliser des modèles déjà présents dans TFLM ainsi qu'un exemple issu de l'association ML Commons qui travaille également sur les performances des réseaux de neurones, entre autres l'exécution de l'inférence sur microcontrôleur. Cela nous permet de faire des essais sur des réseaux fonctionnels sans perdre de temps à leurs développements.

VIII. Travail réalisé

Dans cette partie, nous allons voir le travail que j'ai réalisé durant mon stage. Je vais donc détailler les différentes étapes que j'ai effectuées afin de répondre au mieux au sujet de mon stage.

1. Prise en main de la C-SRAM

En tout premier lieu, j'ai dû comprendre le fonctionnement de la C-SRAM et la prendre en main. Pour cela, j'ai commencé par effectuer une bibliographie en lisant des articles écrits par l'équipe dans laquelle je travaille, dont un premier article sur la partie hardware de la mémoire [1]. Cela m'a permis d'appréhender le fonctionnellement interne de la mémoire, notamment les opérations qui peuvent être effectuées dans la mémoire et proche mémoire. Le reste de ma lecture s'est concentré sur des articles écrits par les personnes qui développent les outils informatiques à destination de la C-SRAM [2][3]. Ce qui m'a permis de comprendre comment se programme la C-SRAM et notamment les spécificités de son jeu d'instruction basé sur les instructions du RISCV.

Cette bibliographie m'a également permis de comprendre le fonctionnement d'Hybrogen, son flot de compilations en deux temps : de l'Hybrolang vers le C puis du C vers un fichier exécutable. J'ai également appréhendé le concept de compilation dynamique qui adapte le code au moment de la compilation.

Suite à cette étude bibliographique, j'ai mis en place un exemple de calcul sur la C-SRAM. Celui-ci effectue la somme de l'ensemble des pixels d'une image. Cela m'a notamment permis de mettre en pratique la méthode de programmation ainsi que de prendre en main l'utilisation du calcul vectorielle entre 2 lignes et donc l'impact de l'arrangement des données dans la mémoire ; ainsi que le fait d'être vigilant sur la taille des données, et aux possibles débordements, notamment lorsque qu'on accumule beaucoup de données dans des variables pouvant être de petite taille.

Mettre en place cet exemple m'a également permis de manipuler les données pour les transférer vers ou depuis la C-SRAM à l'aide du processeur.

2. Analyse de TensorFlow

TensorFlow va nous permettre d'exécuter l'inférence de réseaux de neurones sur la C-SRAM. Nous avons donc besoin de comprendre comment fonctionne le Framework afin de déterminer les parties les plus pertinentes à optimiser pour l'utilisation de la C-SRAM.

L'analyse de la bibliothèque C++ m'a permis, avec l'aide de la documentation, de comprendre comment était organisé le code. Comme on peut le voir sur la Figure 12, TensorFlow lite est composé de deux parties. Un noyau contenant de nombreux opérateurs qui servent pour l'exécution de l'inférence des réseaux de neurones, avec par exemple, des additions et convolutions ; et la partie consacrée aux microcontrôleurs.

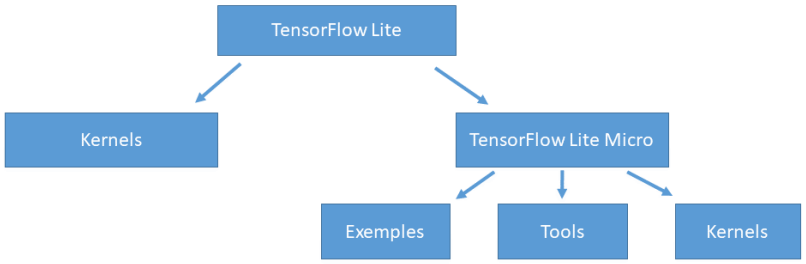


Figure 12: Architecture de la bibliothèque C++ de TensorFlow Lite

Cette sous-partie est composée d'un dossier exemple. Ce dossier contient le code utilisateur pour utiliser l'API. C'est dans cette partie que l'on va effectuer toutes les initialisations nécessaires à l'utilisation du réseau, par exemple le chargement du modèle entrainé préalablement avec TensorFlow, les opérateurs nécessaires à l'exécution de son inférence, mais également tout ce qui est requis pour l'utilisation des microcontrôleurs ; notamment les horloges et les entrées/sorties. Comme son nom l'indique, ce dossier contient des exemples qui ont pour but de montrer l'utilisation de l'API.

Cette sous-partie contient également un noyau. Les fichiers qui y sont présents font appel aux opérateurs adaptés présents dans le noyau général, en fonction du type de données par exemple. C'est aussi dans ce dossier que les opérateurs optimisés pour des architectures particulières sont placés. C'est donc à cet endroit que nous pourrons ajouter les optimisations nécessaires à l'utilisation de la C-SRAM au côté du processeur RISCV.

Le dernier dossier contient les outils qui permettront par la suite de compiler le Framework. Il contient également toutes les configurations particulières pour certaines cibles, notamment les configurations de cross compilation, ou bien la prise en compte d'opérateurs optimisés pour des cartes particulières. C'est donc un dossier très important pour pouvoir compiler le Framework sur l'architecture contenant la C-SRAM.

Toutes les données dans TensorFlow sont appelées Tenseur. Pour effectuer l'inférence d'un réseau de neurones, on place en entrée des données sous forme de tableau, c'est le tenseur d'entrée. Il sera ensuite transformé tout au long de l'exécution pour former des tenseurs intermédiaires dans le but de donner le tenseur de sortie qui contient le résultat de la prédiction.

La suite de l'analyse m'a permis de déterminer la manière avec laquelle était exécutée l'inférence. On initialise une instance de la classe MicroInterpreter. Dans cette initialisation, on vient charger le modèle de notre réseau de neurones, mais également les opérations qui vont servir au réseau. C'est cette classe qui effectuera l'inférence du réseau. Pour lancer l'exécution d'une inférence, on commence par mettre le tenseur d'entrée dans l'attribut « input » de la classe MicroInterpreter. On vient ensuite exécuter la fonction « Invoke » de la même classe. Cette fonction exécute l'inférence. On récupère à la fin de l'exécution le tenseur de sortie contenant la prédiction dans l'attribut « output » de la classe.

3. Profilage d'exemples

Suite à l'analyse de la bibliothèque TensorFlow Lite, l'objectif suivant a été d'identifier l'opérateur le plus pertinent optimisé afin de concentrer mes efforts sur une optimisation avec un maximum d'impact. Nous avons donc utilisé les deux exemples présents dans TensorFlow ainsi qu'un exemple issu de l'association ML Commons pour les microcontrôleurs qui utilisent également TFLM. Cela nous a permis d'avoir rapidement des exemples fonctionnels avec des modèles préalablement entrainés et optimisés. Ces exemples sont composés de deux réseaux travaillant avec des images en entrée : un premier, assez simple, détectant s'il y a une personne ou non dans l'image et l'autre un peu plus complexe qui classe les images en 10 catégories

telles que des voitures, avions ou encore arbres. Le dernier exemple prend en entrée des fichiers audio et classe si le fichier contient un « oui » ou un « non ».

Une fois que j'ai décidé quels modèles j'allais analyser, j'ai donc cherché à déterminer quels était les fonctions dans lesquelles l'exécution du programme passe le plus de temps. Pour cela, j'ai exécuté l'inférence de ces réseaux avec l'outil de profilage Gprof qui fait partie des outils de la suite GCC. Cet outil calcule le temps passé dans chaque fonction et donne un rapport à la fin de l'exécution. On peut alors voir sur la Figure 13 les résultats de ce rapport. Nous pouvons observer que pour chacun des exemples, ce sont les fonctions de convolutions qui occupent plus de 99% du temps, avec notamment la fonction ConvPerChannel et DepthwiseConvPerChannel.

1) person_detection

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
81.84	0.54	0.54	112	4.82	4.82	tflite::reference_integer_ops::ConvPerChannel
18.19	0.66	0.12	112	1.07	1.07	tflite::reference_integer_ops::DepthwiseConvPerChannel
0.00	0.66	0.00	21905	0.00	0.00	flatbuffers::Table::GetOptionalFieldOffset
0.00	0.66	0.00	8946	0.00	0.00	tflite::QuantizationParameters::scale

2) image_classification

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.00	0.24	0.24	18	13.33	13.33	tflite::reference_integer_ops::ConvPerChannel
0.00	0.24	0.00	4394	0.00	0.00	flatbuffers::Table::GetOptionalFieldOffset
0.00	0.24	0.00	1317	0.00	0.00	tflite::QuantizationParameters::scale
0.00	0.24	0.00	1233	0.00	0.00	tflite::QuantizationParameters::zero_point

3) micro_speech

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.02	0.04	0.04	8	5.00	5.00	tflite::reference_integer_ops::DepthwiseConvPerChannel
0.00	0.04	0.00	3360	0.00	0.00	PcanShrink
0.00	0.04	0.00	3360	0.00	0.00	WideDynamicFunction

Figure 13: Extrais du rapport Gprof pour les 3 exemples

Suite à cette analyse, j'ai donc choisi d'optimiser la fonction ConvPerChannel en effectuant les calculs sur la C-SRAM. En effet, en spécialisant cette unique fonction, on aura un impact maximal sur la performance de l'inférence du réseau de neurones.

4. Implémentation de la convolution optimisée

1. Analyse de la convolution scalaire

J'ai ensuite pris le temps d'analyser la fonction scalaire. (Voir Annexe 1). Dans un premier temps, je me suis intéressé à comprendre comment les données sont organisées pour effectuer le calcul. L'organisation des données est une étape très importante pour l'utilisation de la C-SRAM. En effet, l'organisation des données dans

la mémoire a un impact sur deux aspects : maximiser le nombre de données statiques dans la mémoire, qui permet de minimiser leurs transferts ; et effectuer un maximum d'opérations simultanées en optimisant les possibilités de parallélisme lors du calcul entre plusieurs lignes.

Concernant l'organisation des données, nous avons identifié une opération principale, c'est l'opération de multiplication accumulation (MAC). On effectue dans un premier temps une multiplication entre la première donnée du tenseur d'entrée, et la première du filtre, on accumule ce résultat avec la multiplication des deuxièmes données et ainsi de suite.

On peut donc voir sur la Figure 14 comment le calcul de la convolution par canal est organisé. Elle montre les calculs entre le tenseur d'entrée et le filtre pour former le tenseur de sortie. On peut observer en particulier qu'il y a autant de filtres que la profondeur du tenseur de sortie. On effectue donc à chaque fois cette opération de MAC sur le tenseur d'entrée avec chaque filtre pour construire ce tenseur de sortie.

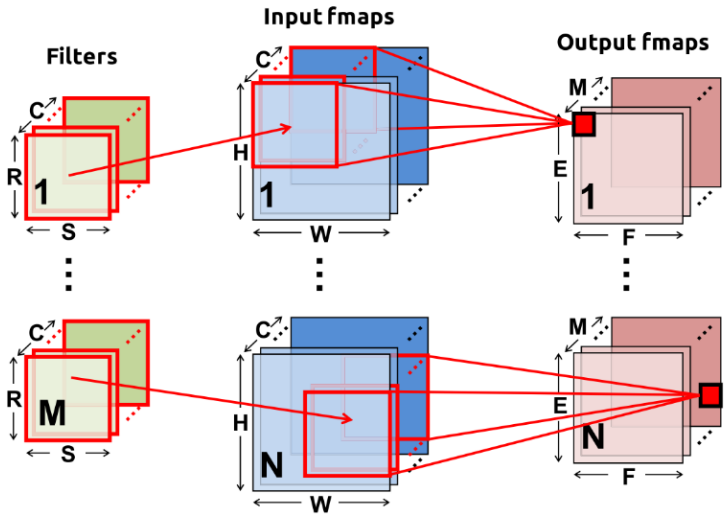


Figure 14: Calcul de la convolution par canal

En fonction de la taille du filtre (paramètres R et S) on va ou non prendre en compte les voisins du point pour lequel on calcule la convolution. On considère que tous les points qui sont à l'extérieur du tenseur d'entrée valent 0. De plus, deux paramètres peuvent être utilisés pour complexifier la convolution, le padding et le stride. Ces deux paramètres agissent sur le déplacement de la fenêtre de convolution sur le tenseur d'entrée.

2. Organisation de la convolution optimisée

Une fois que j'ai analysé comment la fonction fonctionne, je me suis attelé à imaginer le calcul dans la C-SRAM, comment organiser les données pour effectuer les MAC de manière optimale.

Pour minimiser les transferts de données, j'ai choisi de placer le filtre de convolution de manière statique dans la C-SRAM. En effet, les filtres sont les données de plus petites tailles que l'on manipule, on peut donc espérer que sa taille soit inférieure à la taille disponible dans la C-SRAM, ce sont donc les données qu'on a le plus de chance de pouvoir stocker intégralement. Si cependant il est de trop grande taille, on effectuera plusieurs fois l'ensemble du calcul de la convolution pour obtenir le tenseur de sortie en entier. On place chacun des filtres dans la mémoire sur une colonne, comme on peut voir sur la Figure 15. Cela nous permettra de faire l'ensemble des calculs de MAC en parallèle sur toutes les colonnes. On peut par exemple voir sur la Figure 15 que nous avons 2 filtres par colonne, ce qui nous donnera deux valeurs du tenseur de sortie en bas de la mémoire.

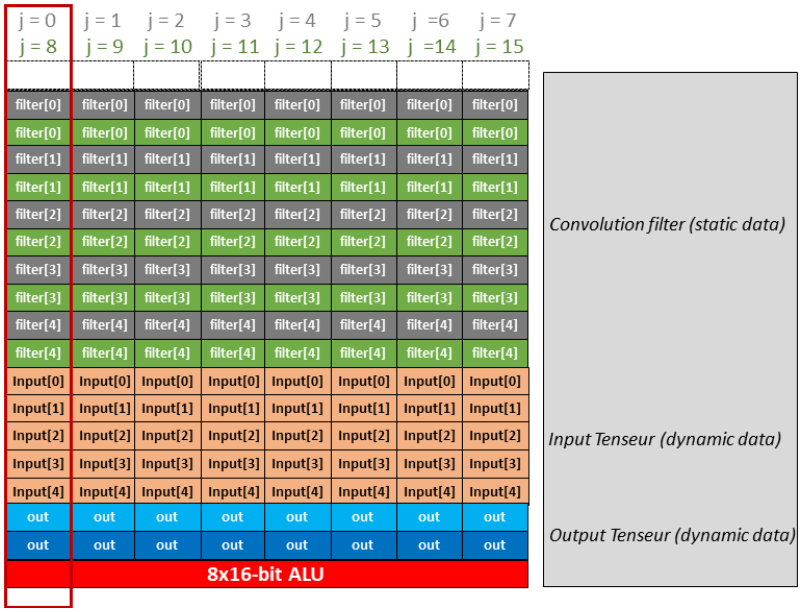


Figure 15: Organisation des données dans la C-SRAM pour le calcul de convolution

On doit ensuite appliquer les filtres sur l'ensemble des fenêtres de convolution du tenseur d'entrée afin de construire le tenseur de sortie. On vient donc placer les fenêtres de convolution les unes après les autres dans la C-SRAM. Cependant, comme on doit appliquer l'ensemble des filtres sur chaque fenêtre de convolution, on copie chaque fenêtre sur l'ensemble des colonnes de la C-SRAM. On applique ensuite les

MAC entre la fenêtre de convolution et les filtres pour obtenir les sorties. On recommence ensuite avec une nouvelle fenêtre de convolution.

Pour finaliser la convolution, il faut ensuite reformater les données, et notamment les remettre à la bonne taille. En effet, les données d'entrée sont sur 8 bits, les opérations de MAC font grandir les données qui passent sur 32 bits. Il faut ensuite les reformater pour les remettre sur 8 bits en fin de convolution. Ces opérations sont effectuées de manière scalaire par le processeur RISCV.

3. Flow de développement de la convolution optimisée

Le développement de la convolution a été effectué par étape. L'objectif était dans un premier temps de pouvoir exécuter une convolution simple, sans paramètre particulier et appliquée à un tenseur de petite taille. Une fois que cette première version est fonctionnelle, nous allons monter petit à petit en complexité pour prendre en compte l'ensemble des paramètres pris en charge par la convolution scalaire.

Pour pouvoir tester si ma convolution est bien fonctionnelle, j'ai extrait des tenseurs et filtres issus de l'exécution de la convolution scalaire dans les exemples analysés précédemment. J'ai à chaque fois extrait le tenseur d'entrée, les filtres ainsi que le tenseur de sortie. Cala me permet de mettre les tenseurs d'entrée et filtres comme paramètres d'entrée de ma convolution et ensuite vérifier que les sorties sont bien égales. Cela permet de développer une convolution optimisée bits compatible avec la convolution scalaire.

J'ai extrait en tout 4 ensembles de tenseurs avec chacun des niveaux de complexité différents. Ils me permettent chacun de valider les différentes étapes de développement de la fonction. Le premier ensemble, appelé « basic », n'a pas de paramètres particuliers et le filtre rentre en entier dans la mémoire. Le second, appelé « long », n'a pas non plus de paramètres, mais le filtre ne rentre pas en entier dans la mémoire, il faudra donc effectuer plusieurs fois l'ensemble des calculs sur différentes parties du filtre afin de construire le tenseur de sortie. Les derniers, appelés « stride » et « pad » comprennent l'utilisation des paramètres de padding et stride de la convolution.

De plus, j'ai identifié deux méthodes pour charger des données du tenseur d'entrée dans la mémoire du fait que l'on recopie sur l'ensemble des colonnes les

données de ce tenseur. Dans un premier temps, j'ai utilisé le processeur pour charger l'ensemble des données dans la mémoire. Pour la seconde méthode, j'ai chargé une fois les données dans la mémoire avec le processeur puis j'ai utilisé l'instruction Broadcast pour recopier les données sur l'ensemble des colonnes dans la mémoire. Les deux méthodes ont donc été implémentées afin de comparer leurs performances.

Tout au long du développement, j'ai testé mon travail avec le simulateur QEMU. Cela me permet de simuler l'exécution du code sur la C-SRAM et donc permettre de comparer les résultats issus du calcul en mémoire ou scalaire. L'autre avantage du simulateur est de pouvoir utiliser des instructions qui ne sont pas encore implémentées sur la version actuelle de la C-SRAM. En effet, pour pouvoir être bit compatible, j'ai besoin d'utiliser des MAC ou multiplication 32 bits, or ces opérations ne sont pas encore implémentées. J'ai donc utilisé la version de QEMU de Kevin, le Thésard qui travaille avec mon tuteur de stage, qui avait implémenté un jeu d'instruction étendu à des fins de test sur son simulateur.

En plus de cela, les opérations de MAC sont décomposées en une multiplication et une addition dans mon code. En effet, le compilateur n'est pas encore capable de générer des instructions de MAC pour la C-SRAM. Suite à l'implémentation, j'obtiens donc la fonction de convolution optimisée présentée en Annexe 2.

5. Performance de la convolution scalaire et C-SRAM

1. Performance de la convolution

Après avoir développé la convolution, nous avons besoin de comparer sa performance avec la convolution scalaire que nous avions à l'origine. Pour cela, nous utilisons encore le simulateur QEMU de l'environnement de Kevin. Celui-ci permet, avec l'ajout d'une librairie, d'obtenir des données de performance d'une exécution. En effet, suite à l'exécution du code, il rend un rapport qui nous donne un grand nombre de paramètres tel que la consommation d'énergie, le nombre de cycles, le nombre d'instructions envoyées à la mémoire ou de transferts de données.

Je me suis concentré sur deux paramètres qui sont le nombre de cycles effectués et la consommation d'énergie. Le nombre de cycles nous donne le temps d'exécution. J'ai ensuite divisé les données obtenues lors de l'exécution de la convolution scalaire

par celle obtenue lors de l'exécution de la convolution optimisée. On obtient donc le gain en énergie et en vitesse de la convolution optimisée par rapport à la référence scalaire.

Les performances de la convolution dépendent également de l'optimisation utilisée pour la compilation des fichiers C vers les fichiers objets avec GCC. J'ai donc compilé les convolutions avec 3 optimisations différentes afin de vérifier l'influence de cette optimisation sur la performance : L'optimisation O1 et O3 qui sont les premier et dernier niveau d'optimisation de GCC pour la performance. Et le niveau d'optimisation Os permet d'obtenir une taille de code la plus réduite possible.

Enfin, j'ai implémenté la convolution avec des opérations 32, 16 et 8 bits afin de comparer les performances de celles-ci. En effet, les opérations 32 bits permettent d'être bits compatibles avec la version scalaire. En plus de cela, nous souhaitons ajouter une opération de MAC qui donne une sortie 2 fois plus grande que l'entrée. Par exemple, elle prendrait en entrée des données sur 16 bits et un résultat sur 32. Cela permettra d'utiliser des opérations 16 bits tout en restant bits compatibles. On peut également imaginer utiliser des opérations qui nous donneraient un résultat sur 16 bits et par conséquent auraient un dépassement sur certaine valeur. D'après mes premières estimations, il s'agit d'un très petit nombre de valeurs ce qui ne devrait donc pas détériorer de manière significative les performances globales du réseau. Si cette hypothèse se confirme, et que nous implémentons des MAC dont le résultat est deux fois plus grand que la sortie, l'utilisation d'opérateurs 8 bits devient donc pertinent à utiliser et donc à évaluer dans cette étude.

On applique ensuite les convolutions aux 4 ensembles de données utilisées précédemment pour développer l'opérateur et on obtient les résultats montrés sur la Figure 16.

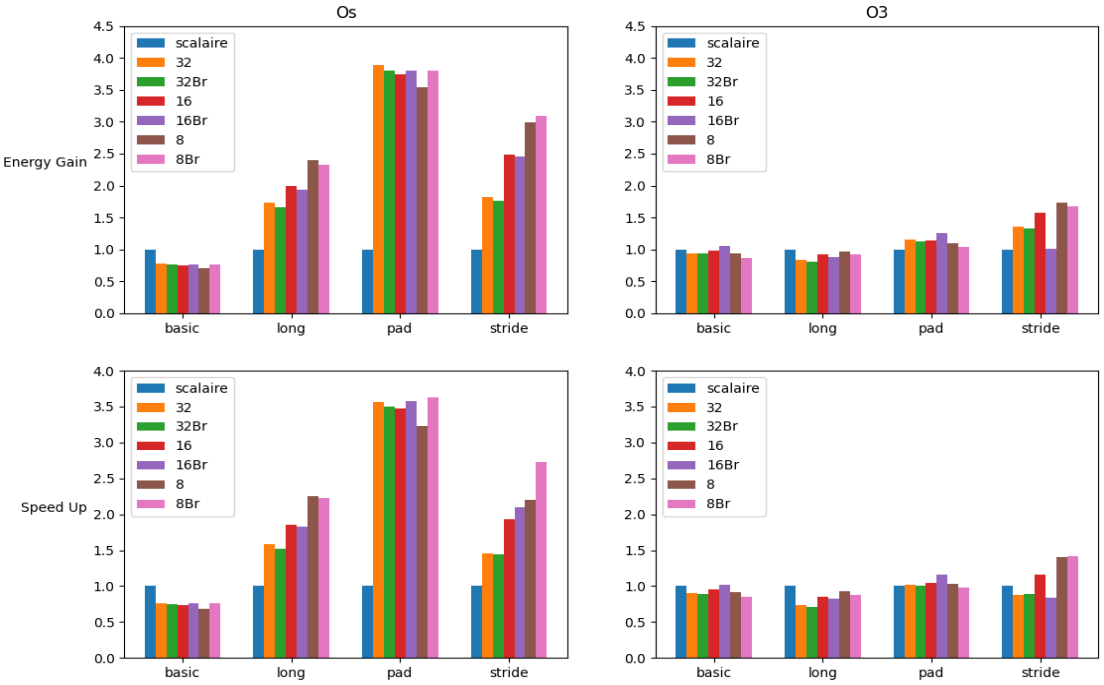


Figure 16: Gain en énergie et en vitesse de la convolution optimisé par rapport à la convolution scalaire en fonction de l'optimisation utilisée lors de la compilation GCC (Os et O3)

On peut donc voir que nous avons des résultats mitigés. Nous avons comparé les performances de la fonction optimisée compilée avec deux flags d'optimisation différents de GCC. Le premier est Os, le compilateur optimise la taille de code de sortie. Ce flag est de ce fait utilisé lorsque que nous souhaitons exécuter des codes sur des microcontrôleurs contenant peu de mémoire. C'est donc une des configurations qui est pertinente avec l'utilisation de la C-SRAM. Le second flag est O3. Avec ce flag, le compilateur va optimiser la performance de l'application.

On peut donc voir que lorsque l'on utilise le flag Os, les performances dépendent des données d'entrée de la convolution. Pour les données d'entrée simple et de petite taille, les convolutions qui utilisent la C-SRAM sont moins performantes que la référence scalaire. Cependant, lorsque les données d'entrée sont de plus grande taille et plus complexe, la convolution optimisée devient plus performante que la référence. Les gains sont parfois supérieurs à un facteur 3,5 pour certains exemples analysés. Cela peut être dû au fait que les optimisations compensent le temps utilisé pour compiler le code au moment de l'exécution lorsqu'il y a un grand nombre de calculs à effectuer.

Pour l'utilisation flag O3, nous pouvons voir que la convolution optimisée n'est pas plus performante que la convolution scalaire de référence. En effet, les

optimisations apportées par le compilateur GCC sont très importantes et plus performantes que les optimisations de la version actuelle de la convolution utilisant la C-SRAM. En effet, nous pouvons penser que les pertes de temps lors de la compilation dynamique pendant l'exécution et le temps de génération d'une instruction pour la C-SRAM ne sont actuellement pas suffisamment compensées par l'utilisation de celle-ci comparé à une version scalaire optimisée avec GCC.

Cependant, la version actuelle de la convolution peut encore être optimisée sur différents aspects, développés dans le paragraphe suivant. Nous pouvons donc émettre l'hypothèse que suite à l'évolution du compilateur et de la C-SRAM, nous pourrions obtenir une convolution optimisée plus performante que la convolution scalaire optimisé avec le flag O3.

2. Optimisation potentielle

Les résultats de la Figure 16 montrent la performance des convolutions avec des opérateurs 32, 16 et 8 bits par rapport à une version scalaire en performance et en énergie. Cependant, l'opérateur MAC utilisé dans ces exemples est décomposé en deux instructions, une multiplication et une addition. On peut donc penser que l'utilisation de l'instruction MAC, permettrait d'obtenir des gains en vitesse et énergie sur plusieurs points. On gagne, en effet, 6 cycles du processeur pour générer une institution pour la C-SRAM plus un transfert via les bus plus l'exécution d'une instruction dans la mémoire.

Une autre possibilité d'optimisation est l'utilisation du Data-locality Management Unit (DMU) pour le transfert des données. C'est un module de transfert de données qui permet de transférer les données de manière ordonnée pour la performance des calculs dans la C-SRAM, et de manière dissociée du processeur. Cela permet donc de libérer le processeur des étapes de transfert des données et le transfert pourrait se faire en parallèle des calculs dans la mémoire. Le DMU a été mis au point et expérimenté par Kevin dans le cadre de sa thèse.

Pour finir, il faut étudier plus précisément l'impact de l'utilisation des opérateurs de petites tailles, qui implique un dépassement potentiel des données pour vérifier si cette optimisation est réellement pertinente.

6. Ajout de la convolution dans TFLM

1. Ajout de la convolution.

Dans un premier temps, j'ai ajouté la convolution au Framework TensorFlow. Pour cela, j'ai ajouté un dossier « cxram » dans le noyau de l'application. Ce dossier contient l'ensemble des fonctions spécialisées pour la C-SRAM.

Ce dossier contient 3 fichiers principaux. Le premier est le fichier conv.cc qui fait appel à la convolution optimisée en fonction des paramètres qui lui sont donnés. Le second « conv_cxram.h » contient la fonction optimisée développée au préalable et adaptée pour s'intégrer au Framework. Le dernier est un fichier Hybrolang qui contient les fonctions appelées par la convolution optimisée pour effectuer les chargements de données dans la mémoire ainsi que les calculs dans la mémoire. Une fois compilé avec Hybrogen, ce fichier devient un header pour la convolution.

Ce dossier contient également un fichier README qui explique comment compiler le Framework pour un RISCV avec la C-SRAM.

Une fois que l'ensemble des fichiers a été ajouté à l'application, j'ai analysé les fichiers de compilation pour pouvoir prendre en compte l'optimisation à l'exécution du code. J'ai donc ajouté un fichier cxram.inc dans le dossier consacré aux librairies externes.

Lorsque vous souhaitez prendre en compte les optimisations, il faut ajouter le flag « CO_PROCESSOR = cxram » lors de la compilation. Cette action permet de prendre en compte le fichier de compilation. Un script remplace également tous les opérateurs génériques par des opérateurs optimisés lorsqu'ils existent.

2. Compilation croisée pour processeur RISCV

Pour pouvoir tester la fonction optimisée, dans l'environnement de TensorFlow, nous devions compiler l'ensemble de Framework avec des instructions RISCV. Pour cela, j'ai ajouté un fichier « riscv32_cxram_makefile.inc » dans le dossier. Cela permet de spécifier le cross compilateur que l'on souhaite utiliser ainsi que les flags de compilation nécessaire. Ce fichier est une extension du Makefile principale de l'application. Il y est ajouté lorsque l'un utilise le flag « TARGET = riscv_cxram ».

Trouver les bons paramètres pour le cross compilateur m'a demandé beaucoup de recherches afin d'obtenir un compilateur qui contienne toutes les librairies nécessaires avec le bon format d'instruction pour le processeur. En effet, il y a une multitude de configurations possibles lors de l'installation du cross compilateur, mais également de flag à mettre en place

J'ai finalement trouvé une installation qui contient l'ensemble des bibliothèques avec le format. Cela m'a donc permis d'exécuter l'inférence des réseaux de neurones optimisée sur le simulateur QEMU. Pour vérifier que l'optimisation ne comporte pas d'erreur, j'ai comparé le résultat de chaque convolution effectuée sur la C-SRAM avec sa version scalaire. En fin d'exécution, j'ai vérifié également que la sortie du réseau de neurones est équivalente à la sortie qui n'utilise pas d'optimisation.

IX. Déroulement du stage

Mon stage de fin d'études peut donc se décomposer en 3 grandes parties.

Une première partie de prise en main et découverte du sujet et des outils. J'ai donc, au travers de l'étude bibliographique et la manipulation d'exemples, pris en main la C-SRAM et les outils qui l'entourent. J'ai également effectué un travail de recherche et analyse de la bibliothèque C++ du Framework TensorFlow Lite et du profilage des exemples de réseau de neurones.

La seconde partie s'est articulée autour de l'analyse de la fonction de convolution, l'implémentation de la version optimisée et l'analyse des performances de la convolution optimisée.

La dernière partie de mon stage fut consacrée à l'ajout de la fonction optimisée dans la librairie TensorFlow lite, puis la compilation du Framework pour l'architecture contenant la C-SRAM et pour finir, l'évaluation des performances de l'inférence d'un réseau de neurones complet comprenant la fonction de convolution optimisée.

On obtient dons la répartition de mon travail sur la durée de mon stage représentée sous forme de diagramme de GANTT sur la Figure 17 ci-dessous.

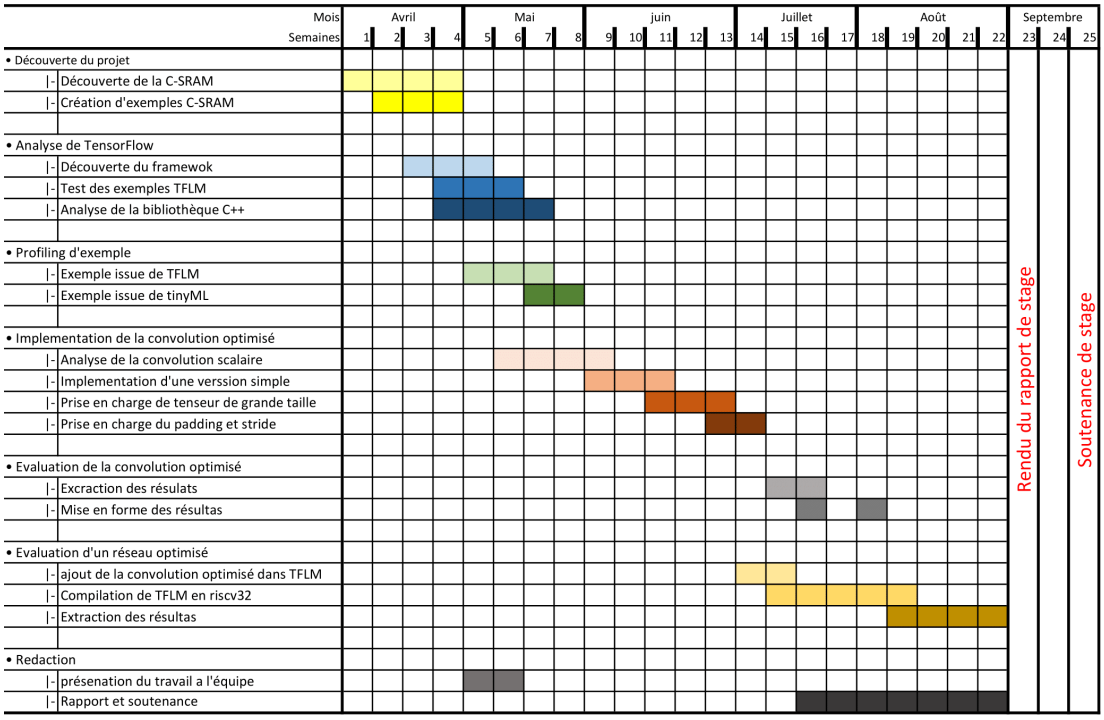


Figure 17: Organisation temporelle du stage

X. Compétences ingénieur IESE

1. Situation professionnelle métier

La situation professionnelle métier que j'ai décidé de développer est : « Optimiser le logiciel en prenant en compte les contraintes de temps d'exécution, de ressources matérielles nécessaires et de consommation. ».

En effet, cette compétence est au cœur des travaux que j'ai effectués durant mon stage. Comme développé précédemment, j'ai commencé mon stage par une grande étape de compréhension de l'architecture que j'allais utiliser, de l'association processeur RISCV et C-SRAM, la manière particulière d'organiser les données et de programmer la mémoire pour effectuer les calculs.

Durant la phase de développement qui a suivi, je me suis attelé à faire en sorte que le logiciel utilise toutes les capacités du matériel pour maximiser sa performance en vitesse et consommation énergétique.

Une dernière phase très importante d'évaluation des performances de mon travail fut de vérifier la pertinence de celui-ci, mais également comparer plusieurs solutions envisagées dans le but de pouvoir choisir de manière éclairée la plus adaptée. Par exemple, l'utilisation, ou non, de l'instruction Broadcast pour le chargement des données en mémoire.

2. Situation professionnelle transversale

Pour la situation professionnelle transversale, j'ai choisi de développer la situation : « Produire des travaux de réflexion sur une technologie émergente ».

Tout au long de mon stage, j'ai été amené à réfléchir autour des technologies émergentes. Par exemple, le DMU pour comprendre comment il fonctionne, et réfléchir sur son utilisation et son intégration dans l'application.

J'ai également été amené à réfléchir autour de l'évolution des instructions pour la C-SRAM. En effet, le flot de développement de la C-SRAM amène régulièrement la question des instructions que l'on souhaite prendre en charge. Cette évolution dépend des applications que l'on souhaite cibler. Pour la C-SRAM, les deux applications principales qui sont visées sont la sécurité et l'intelligence artificielle.

Pour la sécurité, il y a notamment la possibilité d'exécuter des algorithmes de chiffrement et de déchiffrement tel que l'AES ou des algorithmes dits « post-quantum » qui résisteraient à des attaques de l'ordinateur quantique. Mes travaux s'inscrivent dans les applications d'intelligence artificielle et notamment l'accélération et la diminution de la consommation énergétique de l'exécution de l'inférence des réseaux de neurones.

Nous avons donc proposé l'implémentation d'instructions utiles pour les applications d'intelligence artificielle, dont l'instruction MAC à double précision en premier lieu. Pour cette instruction, les entrées de la MAC sont de taille deux fois inférieures à la sortie. Par exemple, effectuer la multiplication sur des données 8 bits et accumuler le résultat sur 16 bits avec les autres multiplications, mais également la présence de multiplication de plus grande taille était nécessaire. En effet, la seule multiplication 8 bits était implémentée.

De plus, les personnes travaillant sur la sécurité ont également proposé des nouvelles instructions. Nous avons donc discuté des priorités à apporter pour leurs implémentations en fonction des ressources disponibles. Il a été décidé que dans un premier temps les instructions pour la cryptographie seraient implémentées et dans un second temps celles pour l'intelligence artificielle.

XI. Conclusion et perspectives

1. Conclusion

Tout au long de ce stage, j'ai donc pu prendre en main l'architecture développée par l'équipe qui contient la C-SRAM et le Framework TensorFlow permettant de déployer des réseaux de neurones. L'analyse de cette seconde composante a permis de détenir une fonction à optimiser, la convolution, pour accélérer et diminuer la consommation d'énergie de l'exécution de l'inférence des réseaux de neurones. J'ai donc ensuite analysé cette fonction pour comprendre en détail son fonctionnement et pouvoir développer une version optimisée de cette convolution.

Pour développer cette fonction, je me suis basé sur les deux atouts principaux de la C-SRAM, la possibilité d'effectuer des calculs en parallèle dans la mémoire sur des vecteurs de 128 bits. Et la plus importante : de garder des données de manière statique dans la mémoire et donc de diminuer les transferts de ses données. Suite à cette implémentation, j'ai évalué les performances de la fonction que j'ai optimisées et comparé ses performances avec la convolution scalaire.

Durant la fin de mon stage, je me suis attelé à ajouter la convolution optimisée au Framework TensorFlow, puis de compiler le Framework pour l'architecture qui contient la C-SRAM, en l'occurrence un processeur RISCV qui envoie les instructions à la C-SRAM. Nous pouvons donc maintenant exécuter l'inférence des réseaux de neurones en utilisant la C-SRAM comme coprocesseur, ce qui permet de diminuer la latence et la consommation d'énergie de cette exécution.

2. Perspectives

Pour poursuivre le travail de mon stage, nous pourrions dans un premier temps évaluer les performances de l'exécution de l'inférence complète d'un réseau de neurones optimisé. Suite à cela, nous pourrions faire évoluer le hardware, avec en particulier l'implémentation de nouvelles instructions qui permettraient d'optimiser le réseau tel que la MAC double précision. Une piste serait également d'évaluer la baisse de performance de l'inférence si on accepte un dépassement lors des calculs de MAC.

La dernière étape serait enfin d'exécuter l'inférence sur l'architecture réelle. Cela permettrait de vérifier que l'ensemble des étapes fonctionne, mais également de comparer les performances issues de simulations et de réelles exécutions.

3. Contributions

À l'issue de mon stage, j'ai donc apporté les contributions suivantes qui pourront resservir à l'équipe lors de la suite de mon stage :

- L'exemple PixelAccumulation que j'ai développé pour prendre en main les outils de compilation dynamique et la C-SRAM. Cet exemple a été ajouté à la version open source de Hybrogen. Cet exemple est également intégré aux tests de régression du compilateur.
- La fonction de convolution optimisée pour la C-SRAM avec 4 jeux de données pour pouvoir tester la convolution. Cette contribution est également ajoutée aux exemples open source et aux tests de régression du compilateur Hybrogen
- La méthodologie et l'automatisation de l'extraction des résultats de performance de la convolution. Cela permet donc à l'équipe de pouvoir facilement extraire de nouveaux résultats si d'autres améliorations sont effectuées et donc pouvoir extraire les gains de la mise en place d'amélioration, tels que par exemple l'utilisation de MAC double précision.
- Le Framework TensorFlow Lite pour microcontrôleur comprenant les optimisations effectuées durant le stage pour la C-SRAM.

De plus, suite à mes travaux de stage, un article de recherche est en cours de rédaction par mon tuteur de stage afin de valoriser les résultats obtenus.

XII. Références

1. Bibliographie

[1] Noel, J.-P., M. Pezzin, R. Gauchi, J.-F. Christmann, M. Kooli, H.-P. Charles, and others, 'A 35.6 TOPS/W/Mm2 3-Stage Pipelined Computational SRAM With Adjustable Form Factor for Highly Data-Centric Applications', IEEE Solid-State Circuits Letters, 3 (2020).

[2] Kevin Mambu and others, 'Instruction Set Design Methodology for In-Memory Computing through QEMU-Based System Emulator', in 2021 IEEE International Workshop on Rapid System Prototyping (RSP), 2021.

[3] Julie Dumas and others, 'Dynamic Compilation for Transprecision Applications on Heterogeneous Platform', Journal of Low Power Electronics and Applications, 11.3 (2021)

2. Sources

Figure 1 : https://www.cea.fr/
Figure 3 : https://www.allaboutcircuits.com/technical-articles/introduction-to-dram-dynamic-random-access-memory/
Figure 4 : https://www.Semiconductorclub.com/
Figure 5 : Julie Dumas and others, 'Dynamic Compilation for Transprecision Applications on Heterogeneous Platform', Journal of Low Power Electronics and Applications, 11.3 (2021)
Figure 6 : http://tpe-ia.lescigales.org/maths.php/
Figure 7 : https://www.juripredis.com/fr/blog/id-19-demystifier-le-machine-learning-partie-2-les-reseaux-de-neurones-artificiels/
Figure 8 : https://fr.wikipedia.org/wiki/R%C3%A9seau_neuronal_convolutif/
Figure 9 : [2]
Figure 10 : [3]
Figure 14 : Vienne Sze and others, 'Efficient Processing of Deep Neural Networks: A Tutorial and Survey', Proceedings of the IEEE, 105.12 (2017)

3. Lien logiciel et contributions

Hybrogen: https://github.com/CEA-LIST/HybroGen

TensorFlow lite : https://github.com/tensorflow/tflite-micro

Example pixel accumulation : https://github.com/CEA-LIST/HybroGen/blob/main/CodeExamples/CxRAM-ImagePixelAccumulation.hl

Convolution optimisé : https://github.com/CEA-LIST/HybroGen/blob/main/CodeExamples/CxRAM-Convolution.hl

XIII. Annexes

Annexe 1 Convolution Scalaire

```
// Fixed-point per-channel-quantization convolution reference
kernel.
inline void ConvPerChannel(
    const ConvParams& params, const int32_t* output_multiplier,
    const int32_t* output_shift, const RuntimeShape& input_shape,
    const int8_t* input_data, const RuntimeShape& filter_shape,
    const int8_t* filter_data, const RuntimeShape& bias_shape,
    const int32_t* bias_data, const RuntimeShape& output_shape,
    int8_t* output_data) {
  // Get parameters.
  const int32_t input_offset = params.input_offset;  // r = s(q - Z)
  const int stride_width = params.stride_width;
  const int stride_height = params.stride_height;
  const int dilation_width_factor = params.dilation_width_factor;
  const int dilation_height_factor = params.dilation_height_factor;
  const int pad_width = params.padding_values.width;
  const int pad_height = params.padding_values.height;
  const int32_t output_offset = params.output_offset;

  // Set min and max value of the output.
  const int32_t output_activation_min =
params.quantized_activation_min;
  const int32_t output_activation_max =
params.quantized_activation_max;

  // Consistency check.
  TFLITE_DCHECK_LE(output_activation_min, output_activation_max);
  TFLITE_DCHECK_EQ(input_shape.DimensionsCount(), 4);
  TFLITE_DCHECK_EQ(filter_shape.DimensionsCount(), 4);
  TFLITE_DCHECK_EQ(output_shape.DimensionsCount(), 4);
  const int batches = MatchingDim(input_shape, 0, output_shape, 0);
  const int input_depth = input_shape.Dims(3);
  const int output_depth = MatchingDim(filter_shape, 0,
output_shape, 3);
  if (bias_data) {
    TFLITE_DCHECK_EQ(bias_shape.FlatSize(), output_depth);
  }

  // Check dimensions of the tensors.
  const int input_height = input_shape.Dims(1);
  const int input_width = input_shape.Dims(2);
  const int filter_height = filter_shape.Dims(1);
  const int filter_width = filter_shape.Dims(2);
  const int filter_input_depth = filter_shape.Dims(3);
  const int groups = input_depth / filter_input_depth;
  TFLITE_DCHECK_EQ(input_depth % filter_input_depth, 0);
  const int filters_per_group = output_depth / groups;
  const int output_height = output_shape.Dims(1);
  const int output_width = output_shape.Dims(2);
  for (int batch = 0; batch < batches; ++batch) {
    for (int out_y = 0; out_y < output_height; ++out_y) {
      const int in_y_origin = (out_y * stride_height) - pad_height;
      for (int out_x = 0; out_x < output_width; ++out_x) {
        const int in_x_origin = (out_x * stride_width) - pad_width;
```

```
        for (int out_channel = 0; out_channel < output_depth;
++out_channel) {
          auto group = out_channel / filters_per_group;
          int32_t acc = 0;
          for (int filter_y = 0; filter_y < filter_height;
++filter_y) {
            const int in_y = in_y_origin + dilation_height_factor *
filter_y;
            for (int filter_x = 0; filter_x < filter_width;
++filter_x) {
              const int in_x = in_x_origin + dilation_width_factor *
filter_x;

              // Zero padding by omitting the areas outside the
image.
              const bool is_point_inside_image =
                  (in_x >= 0) && (in_x < input_width) && (in_y >= 0)
&&
                  (in_y < input_height);

              if (!is_point_inside_image) {
                continue;
              }

              for (int in_channel = 0; in_channel <
filter_input_depth;
                   ++in_channel) {
                int32_t input_val =
                    input_data[Offset(input_shape, batch, in_y,
in_x,
                                      in_channel + group *
filter_input_depth)];
                int32_t filter_val = filter_data[Offset(
                    filter_shape, out_channel, filter_y, filter_x,
in_channel)];
                acc += filter_val * (input_val + input_offset);
              }
            }
          }

          if (bias_data) {
            acc += bias_data[out_channel];
          }
          acc = MultiplyByQuantizedMultiplier(
              acc, output_multiplier[out_channel],
output_shift[out_channel]);
          acc += output_offset;
          acc = std::max(acc, output_activation_min);
          acc = std::min(acc, output_activation_max);
          output_data[Offset(output_shape, batch, out_y, out_x,
out_channel)] =
              static_cast<int8_t>(acc);
        }
      }
    }
  }
}
```

Annexe 2 Convolution Optimisé 32 bits

```
inline void ConvPerChannel(
    const ConvParams& params, const int32_t* output_multiplier,
    const int32_t* output_shift, const RuntimeShape& input_shape,
    const int8_t* input_data, const RuntimeShape& filter_shape,
    const int8_t* filter_data, const RuntimeShape& bias_shape,
    const int32_t* bias_data, const RuntimeShape& output_shape,
    int8_t* output_data) {

  // Get parameters.
  const int32_t input_offset = params.input_offset;  // r = s(q - Z)
  const int stride_width = params.stride_width;
  const int stride_height = params.stride_height;
  const int dilation_width_factor = params.dilation_width_factor;
  const int dilation_height_factor = params.dilation_height_factor;
  const int pad_width = params.padding_values.width;
  const int pad_height = params.padding_values.height;
  const int32_t output_offset = params.output_offset;
  // Set min and max value of the output.
  const int32_t output_activation_min =
params.quantized_activation_min;
  const int32_t output_activation_max =
params.quantized_activation_max;
  // Consistency check.
  TFLITE_DCHECK_LE(output_activation_min, output_activation_max);
  TFLITE_DCHECK_EQ(input_shape.DimensionsCount(), 4);
  TFLITE_DCHECK_EQ(filter_shape.DimensionsCount(), 4);
  TFLITE_DCHECK_EQ(output_shape.DimensionsCount(), 4);
  const int batches = MatchingDim(input_shape, 0, output_shape, 0);
  const int input_depth = input_shape.Dims(3);
  const int output_depth = MatchingDim(filter_shape, 0,
output_shape, 3);
  if (bias_data) { TFLITE_DCHECK_EQ(bias_shape.FlatSize(),
output_depth); }

  // Check dimensions of the tensors.
  const int input_height = input_shape.Dims(1);
  const int input_width = input_shape.Dims(2);
  const int filter_height = filter_shape.Dims(1);
  const int filter_width = filter_shape.Dims(2);
  const int filter_input_depth = filter_shape.Dims(3);
  const int groups = input_depth / filter_input_depth;
  TFLITE_DCHECK_EQ(input_depth % filter_input_depth, 0);
  const int filters_per_group = output_depth / groups;
  const int output_height = output_shape.Dims(1);
  const int output_width = output_shape.Dims(2);

  //set CxRAM parameters
  h2_insn_t *ptr;
  genMultiplyAccumulateType h2_MacCode;
  h2_cxram_line_t *filterCxram, *inputCxram, *resCxram;
  const int nbLineRes = output_depth % DATA_PER_LINE ? (output_depth
/ DATA_PER_LINE) + 1 : output_depth / DATA_PER_LINE;
  const int nbLineInput = filter_height * filter_width *
filter_input_depth;
  const int nbLineFilter = nbLineInput * nbLineRes;
  const int nbTimes = nbLineFilter % TAILLE_MAX_FILTRE ?
(nbLineFilter / TAILLE_MAX_FILTRE) + 1 : nbLineFilter /
```

```
TAILLE_MAX_FILTRE; // Nombre de passe a effectuer pour obtenir la
profondeur totale de la sortie en divisant le chargement du filtre
  const int nbLineResPerTime = nbLineRes % nbTimes ? nbLineRes /
nbTimes + 1 : nbLineRes / nbTimes;
  const int nbLineFilterPerTime = nbLineInput * nbLineResPerTime;

  h2_init_cxram();
  filterCxram = &h2_cxram_tiles[0];
//filtre
  inputCxram = &h2_cxram_tiles[nbLineFilter + 1];
//input (+1 pour l'offset)
  resCxram = &h2_cxram_tiles[nbLineFilter + nbLineInput + 1];
//output (+1 pour l'offset)
  ptr = h2_malloc(1024);
  h2_MacCode = genMultiplyAccumulate(ptr);
  printf("Code generated\n");
//boucles de calcul en CxRAM
  for (int time = 0; time < nbTimes; time++)
      {
            const int out_channel_begin = time * nbLineResPerTime *
DATA_PER_LINE;
            const int out_channel_end = (out_channel_begin +
nbLineResPerTime * DATA_PER_LINE) > output_depth ? output_depth :
out_channel_begin + nbLineResPerTime * DATA_PER_LINE;
            const int nbLineResThisTime = (out_channel_end -
out_channel_begin) / DATA_PER_LINE;
    loadFilter(filter_shape, filter_data, filterCxram, input_offset,
out_channel_begin, out_channel_end);
    for (int batch = 0; batch < batches; batch++){
      for (int out_y = 0; out_y < output_height; out_y++){
        const int in_y = (out_y * stride_height) - pad_height;
        for (int out_x = 0; out_x < output_width; out_x++){
          const int in_x = (out_x * stride_width) - pad_width;
          loadInput(input_shape, filter_shape, input_data, batch,
in_x, in_y, params, inputCxram);
          h2_MacCode(filterCxram, inputCxram, resCxram, nbLineInput,
nbLineRes);
          printf("MAC cxram OK\n");
          for (int out_channel = out_channel_begin; out_channel <
out_channel_end; out_channel++){
            int32_t acc = resCxram[0].i16[out_channel-
out_channel_begin];
            if (bias_data) {
              acc += bias_data[out_channel];
            }
            acc = MultiplyByQuantizedMultiplier(
                acc, output_multiplier[out_channel],
output_shift[out_channel]);
            acc += output_offset;
            acc = std::max(acc, output_activation_min);
            acc = std::min(acc, output_activation_max);
            output_data[Offset(output_shape, batch, out_y, out_x,
out_channel)] =
                static_cast<int8_t>(acc);
          printf("Quantization OK\n");
          }
        }
      }
    }
```

```
// -*- c -*-
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tensorflow/lite/kernels/internal/common.h"
#include "tensorflow/lite/kernels/internal/types.h"

#define DATA_PER_LINE 4  // nombre de données par ligne
#define TAILLE_MAX_FILTRE 255

typedef void (*genMultiplyAccumulateType)(h2_cxram_line_t*,
h2_cxram_line_t*,h2_cxram_line_t*, int inDepth, int outDepth);

// chargement du filtre dans la CSRAM
inline void loadFilter(const tflite::RuntimeShape& filter_shape,
const int8_t* data,
                h2_cxram_line_t* filterCxram, int32_t offset,
                int out_channel_begin, int out_channel_end) {
  const int input_depth = filter_shape.Dims(3);
  const int width = filter_shape.Dims(2);
  const int height = filter_shape.Dims(1);
  int index = 0;
  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      for (int in_channel = 0; in_channel < input_depth;
in_channel++) {
        for (int out_channel = out_channel_begin;
             out_channel < out_channel_end; out_channel++) {
          int32_t filter_val =
              data[Offset(filter_shape, out_channel, y, x,
in_channel)];
          filterCxram[0].i32[index++] = filter_val;
        }
        if (out_channel_end % DATA_PER_LINE) {
          index += (DATA_PER_LINE - (out_channel_end %
                                     DATA_PER_LINE));  // alignement
des filtres
        }
      }
    }
  }
  // ajout de l'offset a la fin
  for (int i = 0; i < DATA_PER_LINE; i++) {
    filterCxram[0].i32[index++] = offset;
  }
}

// charement des input dans la CSRAM
inline void loadInput(const tflite::RuntimeShape& input_shape,
                const tflite::RuntimeShape& filter_shape, const
int8_t* data, const int batch, const int x_origin, const int
y_origin, const tflite::ConvParams& params, h2_cxram_line_t*
inputCxram) {
  const int filter_height = filter_shape.Dims(1);
  const int filter_width = filter_shape.Dims(2);
  const int depth = input_shape.Dims(3);
  const int width = input_shape.Dims(2);
  const int height = input_shape.Dims(1);
  const int dilation_width = params.dilation_width_factor;
```

```
  const int dilation_height = params.dilation_height_factor;
  const int32_t offset = params.input_offset;
  int index = 0;
  for (int filter_y = 0; filter_y < filter_height; ++filter_y) {
    const int y = y_origin + dilation_height * filter_y;
    for (int filter_x = 0; filter_x < filter_width; ++filter_x) {
      const int x = x_origin + dilation_width * filter_x;
      for (int channel = 0; channel < depth; channel++) {
        const int32_t val =
            ((x >= 0) && (x < width) && (y >= 0) && (y < height))
                ? data[Offset(input_shape, batch, y, x, channel)]
                : -offset;
        for (int i = 0; i < DATA_PER_LINE; i++) {
          inputCxram[0].i32[index++] = val;
        }
      }
    }
  }
}

inline void printCxram(int len, int nb_ligne, char* mesg,
                       h2_cxram_line_t* vectorLine) {
  for (int i = 0; i < nb_ligne; i++) {
    printf("%s :", mesg);
    for (int j = 0; j < len; j++) {
      printf(" %05d", vectorLine[i].i32[j]);
    }
    printf("\n");
  }
}

genMultiplyAccumulateType genMultiplyAccumulate(h2_insn_t *ptr)
{
#[
      int 32 1 MultiplyAccumulate(int[] 32 4 filter, int[] 32 4
input, int[] 32 4 res, int 32 1 inDepth, int 32 1 resDepth)
      {
            int 32 1 i;
            int 32 1 j;
            int 32 1 k;
            for (i = 0; i < resDepth; i = i + 1)
            {
                  res[i] = 0;
                  for (j = 0; j < inDepth; j = j + 1)
                  {
                        k = j * resDepth;
                        k = k + i;
                        res[i + 1] = input[j] + filter[inDepth *
resDepth]; // offset
                        res[i + 1] = res[i + 1] * filter[k];
                                          // multiply
                        res[i] = res[i] + res[i + 1];// accumulate
                  }
            }
      }
]#
      return (genMultiplyAccumulateType) ptr;
}
```

DOS DU RAPPORT

Etudiant : BRICOUT Thaddée

Année d'étude dans la spécialité : IESE5

Entreprise : CEA Grenoble

17 rue des Martyrs,

38054, Grenoble, France.

Responsable administratif : Florent LEPIN

Téléphone :

Courriel: florent.lepin@cea.fr

Tuteur de stage : Henri-Pierre CHARLES

Téléphone :

Courriel: henri-pierre.charles@cea.fr

Enseignant-référent : Fréderic ROUSSEAU

Téléphone :

Courriel: Frederic.Rousseau@univ-grenoble-alpes.fr

Titre : Evaluation de l'inférence d'un réseau de neurones sur une architecture C-SRAM (Computing SRAM)

Résumé :

Ce stage s'inscrit dans une équipe de développement d'un module de calcul en mémoire C-SRAM. Le développement de ce module nécessite de l'évaluer en fonction de son usage pour différents domaines. Le but du stage est donc d'évaluer les performances de l'inférence d'un réseau de neurones avec l'utilisation de cette carte.

Pour cela, j'ai dû dans un premier temps prendre en main cette architecture particulière composée d'une C-SRAM couplée avec un processeur RISCV et l'environnement Hybrogen. J'ai ensuite analysé le Framework de machine Learning TensorFlow afin de voir comment il serait possible d'exécuter une partie de son code sur la C-SRAM dans le but d'améliorer ses performances.

Suite à ces deux travaux, j'ai identifié puis optimisé une fonction de convolution avec l'utilisation de la C-SRAM. Cela m'a permis de comparer les performances de la fonction scalaire avec la fonction optimisée.

Pour finir, j'ai ajouté la fonction optimisée dans le Framework TensorFlow afin de pouvoir évaluer les performances globales de l'inférence du réseau avec et sans l'utilisation de la C-SRAM.