Performance of a Neural Network Accelerator Architecture and its Optimization Using a Pipeline-Based Approach

Ali Oudrhiri

Thèse présentée pour l'obtention du grade de

DOCTEUR de SORBONNE UNIVERSITÉ

École Doctorale Informatique, Télécommunications et Électronique

réalisée entre

STMicroelectronics, Crolles
Laboratoire d'Informatique de Paris 6

Performance of a Neural Network Accelerator Architecture and its Optimization Using a Pipeline-Based Approach

---

Ali OUDRHIRI

Frédéric PETROT, Prof., UGA, CNRS, Grenoble INP, TIMA, Grenoble	*Président du jury*
Angeliki KRITIKAKOU, Assoc.Prof, Inria, Univ Rennes, CNRS, IRISA, Rennes	*Rapporteur*
Philippe COUSSY, Prof., Lab-STICC, Univ. de Bretagne-Sud, Lorient	*Rapporteur*
Roselyne CHOTIN, Assoc.Prof, Sorbonne Univ., CNRS, LIP6, Paris	*Examinateur*
Maxime PELCAT, Assoc.Prof, IETR, INSA Rennes	*Examinateur*
Pascal URARD, Directeur innovation, STMicroelectronics, Crolles	*Invité*
Alix MUNIER KORDON, Prof., Sorbonne Univ., CNRS, LIP6, Paris	*Directeur de thèse*

ABSTRACT

In recent years, neural networks have gained widespread popularity for their versatility and effectiveness in solving a wide range of complex tasks. Their ability to learn and make predictions from large data sets has revolutionized various fields. However, as neural networks continue to find applications in an ever-expanding array of domains, their significant computational requirements become a pressing challenge. This computational demand is particularly problematic when deploying neural networks in resource-constrained embedded devices, especially within the context of edge computing for inference tasks.

Nowadays, neural network accelerator chips emerge as the optimal choice for supporting neural networks at the edge. These chips offer remarkable efficiency with their compact size, low power consumption, and reduced latency. Moreover, the fact that they are integrated on the same chip environment also enhances security by minimizing external data communication. In the frame of edge computing, diverse requirements have emerged, necessitating trade-offs in various performance aspects. This has led to the development of highly configurable accelerator architectures, allowing them to adapt to distinct performance demands.

In this context, the focus lies on Gemini, a configurable inference neural network accelerator designed with imposed architecture and implemented using High-Level Synthesis techniques. The considerations for its design and implementation were driven by the need for parallelization configurability and performance optimization.

Once this accelerator was designed, demonstrating the power of its configurability became essential, helping users select the most suitable architecture for their neural networks. To achieve this objective, this thesis contributed to the development of a performance prediction strategy operating at a high-level of abstraction, which considers the chosen architecture and neural network configuration. This tool assists clients in making decisions regarding the appropriate architecture for their specific neural network applications.

During the research, we noticed that using one accelerator presents several limits and that increasing parallelism had limitations on performances. Consequently, we

adopted a new strategy for optimizing neural network acceleration. This time, we took a high-level approach that did not require fine-grained accelerator optimizations. We organized multiple Gemini instances into a pipeline and allocated layers to different accelerators to maximize performance. We proposed solutions for two scenarios: a user scenario where the pipeline structure is predefined with a fixed number of accelerators, accelerator configurations, and RAM sizes. We proposed solutions to map the layers on the different accelerators to optimize the execution performance. We did the same for a designer scenario, where the pipeline structure is not fixed, this time it is allowed to choose the number and configuration of the accelerators to optimize the execution and also hardware performances. This pipeline strategy has proven to be effective for the Gemini accelerator.

Although this thesis originated from a specific industrial need, certain solutions developed during the research can be applied or adapted to other neural network accelerators. Notably, the performance prediction strategy and high-level optimization of NN processing through pipelining multiple instances offer valuable insights for broader applications.

Keywords: *Neural Network Accelerators*, *ASIC*, *Output Stationary*, *Estimation*, *Optimization*, *Throughput Enhancement*, *Latency*, *Area*, *Power*, *Pipeline Configuration*.

Résumé

Introduction

Depuis l'histoire de l'humanité, la résolution de problèmes a toujours été une capacité naturelle de l'homme, façonnant sa compréhension du monde à travers la pensée rationnelle et l'art de la résolution de problèmes. Cette capacité a conduit au développement d'approches algorithmiques, puis à l'utilisation d'ordinateurs pour améliorer nos capacités à résoudre des problèmes complexes. Cependant, le cerveau humain n'est pas strictement basé sur des instructions rigides. En effet, l'être humain est capable d'apprendre par l'observation, l'expérience et l'entrainement, ce qui lui permet d'exécuter des tâches en se fondant sur ces apprentissages. Cette approche flexible pour résoudre des problèmes a été une grande source d'inspiration pour le domaine de l'intelligence artificielle, en particulier pour le développement des réseaux de neurones.

Au sein de l'IA, les réseaux de neurones se sont distingués en résolvant de manière autonome des problèmes complexes, notamment dans des applications telles que les assistants personnels virtuels, les systèmes de recommandation et les véhicules autonomes. Le regain d'intérêt pour les NN est en partie dû aux avancées significatives en matière de matériel, en particulier les accélérateurs sur puce (ASIC).

La compétition pour le développement d'accélérateurs spécifiques à l'inférence de réseaux de neurones est en cours, en particulier dans le domaine des accélérations en périphérie, un environnement contraignant en termes de surface et d'énergie. L'optimisation des architectures matérielles revêt une importance cruciale pour répondre aux besoins d'applications diverses. Cette concurrence a stimulé le développement d'architectures configurables, capables de s'adapter à plusieurs applications. La prédiction précise des performances de ces accélérateurs pour diverses configurations est essentielle pour leur sélection. De plus, l'utilisation de multiples instances de ces accélérateurs afin d'optimiser leur efficacité et leur configurabilité est une avenue prometteuse, notamment grâce à l'utilisation de pipelines d'accélérateurs.

Cette thèse est le fruit d'une collaboration CIFRE entre STMicroelectronics et le

laboratoire LIP6 de Sorbonne Université. En résumé, cette thèse apporte des contributions majeures dans quatre domaines principaux :

- Le développement d'un accélérateur d'inférence de réseaux de neurones configurables appelé Gemini.
- La mise en place d'un cadre de prédiction des performances de Gemini en fonction du réseau de neurones et de la configuration matérielle.
- L'utilisation d'un shéma fixe de pipelines d'accélérateurs pour optimiser l'exécution d'un réseau de neurone, où l'affectation des tâches aux différents accélérateurs est optimisée.
- Utilisation de pipelines d'accélérateurs non fixes, permettant d'adapter la structure du pipeline pour une optimisation plus fine de l'exécution des réseaux de neurones et du matériel.

Les deux derniers axes de recherche sont illustrés à travers l'exemple de Gemini.

La thèse couvre les fondements des réseaux de neurones, l'architecture matérielle de Gemini, la conception de Gemini, la configurabilité, les pipelines d'accélération fixes et non fixes, et se termine par une conclusion mettant en avant les contributions de la recherche et les futures perspectives.

Généralité sur les réseaux de neurones utilisés

Le premier objectif de cette thèse consiste à concevoir un accélérateur matériel spécifiquement dédié à l'inférence de réseaux de neurones. Ces réseaux, inspirés du fonctionnement du cerveau humain et de sa capacité d'apprentissage, opèrent grâce à des ajustements dynamiques des *poids* synaptiques, mimant ainsi la capacité d'apprentissage humaine. La prédiction dans ce contexte s'appuie sur l'utilisation de ces ajustements pour effectuer une inférence précise.

L'histoire des réseaux de neurones remonte aux années 1940, avec les travaux novateurs de Donald Hebb sur le renforcement des voies neuronales par l'utilisation, un concept fondamental dans le processus d'apprentissage humain. Au fil des décennies,

ces réseaux ont évolué en passant par différentes phases de développement, notamment avec l'introduction du réseau de neurones convolutif et d'autres avancées notables. Cependant, la résurgence actuelle des réseaux de neurones est attribuable à trois facteurs déterminants : le développement de matériel spécifique pour supporter les algorithmes, la disponibilité de données partagées pour l'entraînement de ces réseaux et la croissance d'outils en accès libre. Ces facteurs combinés ont favorisé l'émergence d'une multitude de réseaux, chacun présentant des structures diversifiées composées de couches variées et orientés vers une grande variété d'applications, chacune exigeant des spécifications spécifiques.

Dans le cadre de cette thèse, notre attention se porte sur l'inférence de réseaux de neurones feed-forward, caractérisés par une structure en cascade, où chaque couche suit la précédente, et est séparée par une fonction d'activation. Dans cette étude, nous avons choisi d'utiliser la fonction ReLu comme fonction d'activation. Nous nous concentrons en particulier sur les réseaux comportant des couches de convolution, convolutions séparables, pooling et couches entièrement connectées. En ce qui concerne les applications, notre intérêt se porte principalement sur les réseaux de neurones qui possèdent un nombre de *poids* modéré. Cette orientation s'explique par notre objectif de les utiliser dans des contextes de calcul en périphérie, où les contraintes liées à la taille et à la consommation d'énergie sont particulièrement prédominantes. Nous avons identifié trois réseaux spécifiques qui seront au cœur de nos études tout au long de cette thèse. Le premier, MobileNet x0.25 [61], est composé de 27 couches, principalement constituées de convolutions séparables. Ensuite, nous avons VGG-like inspiré du modèle VGG-16 [115] et comprend 11 couches combinant des convolutions, des max-pools, et des couches entièrement connectées (relativement grandes). Enfin, PNet, avec ses 7 couches [130], partage la même structure que les précédents. Ces réseaux ont des tailles de *poids* de 850 000, 526 000 et 7 900, respectivement, et des dimensions d'images d'entrée (*ifmaps*) de 224x224x3, 128x128x1 et 32x32x1, respectivement.

La figure 1 résume les notations utilisées tout au long de la thèse pour décrire les entrées, les sorties et les paramètres des différentes couches des réseaux de neurones : un réseau de neurones (NN) prend en entrée un lot de k images ou activations d'entrée (*ifmap*) en 3D. Chaque ifmap est constituée de C canaux, avec W pixels de largeur et H pixels de hauteur dans une image 2D. Les sorties du NN sont un lot de k activa-

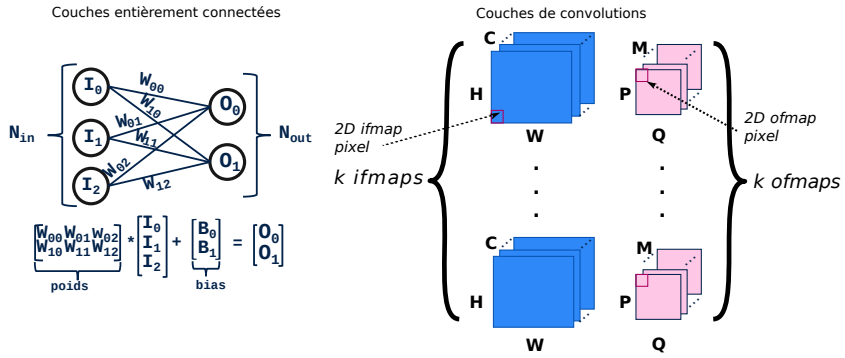

Fig. 1 *Notations adoptées pour les paramètres des couches des réseaux de neurones*

tions de sortie (ofmaps) en 3D, ayant les dimensions $P \times Q \times M$ pour la hauteur, la largeur et les canaux respectivement. La valeur de M correspond au nombre de filtres pour les couches de convolution, convolution séparables et maxpool. Pour les couches entièrement connectées, les *ifmaps* et *ofmaps* en 3D sont aplaties en une dimension pour donner les neurones d'entrée (N_{in}) et de sortie (N_{out}).

Support matériel des réseaux de neurones

Au sein du projet Gemini, nous étions contraints par une architecture matérielle prédéfinie. Celle-ci était basée sur une mémoire SRAM intégrée, qui stocke les informations liées au réseau de neurones, associée à une unité de traitement de neurones (NPU) responsable de l'exécution des calculs nécessaires pour les sorties de chaque couche. Les opérations clés consistent principalement en des multiplications et accumulations (MAC). Le NPU se compose d'unités de processeurs (PE) qui fonctionnent en parallèle. Chaque unité de processeur est équipée de la logique nécessaire pour effectuer des opérations de type MAC (ou autres) et dispose de registres de stockage.

Cette architecture s'inscrit dans le cadre du paradigme de calcul en mémoire proche. En ce qui concerne le flux de données, nous étions contraints d'utiliser un schéma où la sortie est stationnaire [118]. Cette stratégie de flux de données repose sur le fait que les opérations de multiplication et d'accumulation sont stationnaires au sein des registres des unités de processeurs. Chaque processeur calcule ainsi un pixel de sortie. Cette approche est particulièrement intéressante, car elle permet d'exécuter des calculs simultanément, améliorant la latence, tout en réutilisant les données pour minimiser les opérations d'accès et de transfert de données, ce qui a un impact direct sur la consom-

mation énergétique des puces.

La figure 2 illustre cette architecture, mettant en évidence la structure de l'accélérateur comprenant le NPU, la SRAM, ainsi que les PEs (où ALU désigne le bloc réalisant les MACs ou autres opérations de calculs). Le flux de données à sortie stationnaire est également représenté, montrant que la somme des sorties reste constamment dans les registres des PEs, tandis que les données d'entrée (*ifmaps* et *poids*) sont transférées aux PEs à chaque cycle.

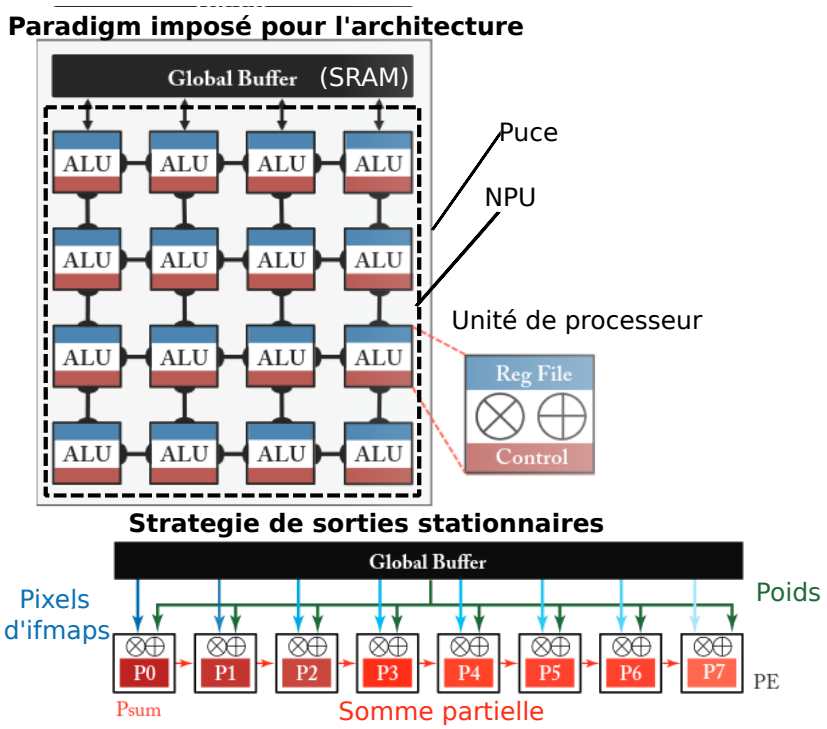

Fig. 2 *Architecture imposée pour un accélérateur ayant un flux de données à sortie stationnaire*

Au cours de cette étude, nous avons examiné les indicateurs clés de performance (KPI) essentiels pour comparer les accélérateurs de réseaux de neurones. Parmi eux, la latence (nombre de cycles d'exécution) et le débit (nombre d'inférences par seconde) sont cruciaux. Nous avons également pris en compte la surface de la puce, la consommation (puissance ou énergie), et des KPI combinant plusieurs aspects, tels que l'efficacité énergétique mesurée en $TOPS/W$ (opérations en téra par seconde par watt).

Gemini, un accélérant configurable pour l'inférence de réseaux de neurones

Dans le cadre du projet Gemini, nous avons dû concevoir un accélérateur de réseaux de neurones basé sur l'architecture présentée précédemment, en utilisant une stratégie de flux de données à sortie stationnaire. Cet accélérant devait en outre satisfaire plusieurs spécifications garantissant le succès du projet. Tout d'abord, Gemini doit être configurable, capable de changer d'architecture pour s'adapter aux différentes contraintes de surface, de consommation et de latence. De plus, Gemini doit traiter des données quantifiées et avoir des sorties au format compatible avec TensorFlow [64]. Gemini doit également répondre à certaines spécifications relatives aux KPIs. Enfin, Gemini doit être validé de manière exhaustive, et la conception doit être maintenable. Ces différentes spécifications imposent des contraintes au niveau de la conception et de l'implémentation.

Les entrées et les sorties de l'accélérateur sont celles du réseau de neurones présenté précédemment. Cependant, au niveau du matériel, il a fallu choisir leur représentation. Pour cela, nous avons opté pour la quantification TensorFlow en 8 bits pour les *poids* et les *fmaps*.

En ce qui concerne la conception, nous avons suivi les directives imposées, en utilisant une architecture composée d'une RAM pour les *fmaps* appelé **FMAP** RAM et une RAM pour les *poids* appelée **WEIGHTS** RAM. Le NPU contient un tableau d'unités de traitement (PEs) organisées en 2D, avec des paramètres configurables, notamment $WPAR$ pour la parallélisation en largeur des *ofmaps* et $MPAR$ pour la parallélisation des filtres. Ces paramètres structurels sont cruciaux pour la conception globale et la planification des opérations, en particulier pour optimiser les convolutions. La Figure 4.1 illustre ces notations avec $(WPAR, MPAR) = (2, 4)$.

En ce qui concerne le NPU, il est composé de trois composants. Tout d'abord, les unités de traitement (PE) de Gemini: ces PEs sont organisées en parallèle, et comme décrit précédemment, la somme partielle est stationnaire, tandis que les *poids* et les *fmaps* leur sont diffusés à chaque cycle. Dans le cas des couches entièrement connectées, ces PEs calculent simultanément $N = WPAR \times MPAR$ pixels de sortie. Pour les couches de convolution, les PEs sont organisés en un tableau bidimensionnel,

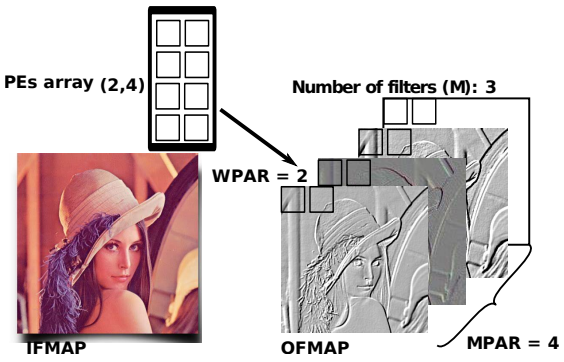

Fig. 3 *Unités de traitements pour* $(WPAR, MPAR) = (2, 4)$

ce qui permet de calculer $WPAR$ pixels *d'ofmap* pour $MPAR$ *ofmaps* en parallèle. L'architecture de chaque PE comprend deux étapes pipelinées: la première effectue des opérations primaires telles que les MAC pour les calculs de couches, tandis que la deuxième étape se concentre sur la quantification des pixels de sortie.

Le deuxième composant est les mixeursd'entrée :ils réorganisent les données (*ifmaps et poids* pour les adapter aux besoins des PEs. Le dernier composant est l'étage d'écriture. Il permet la réorganisation des données en sortie des PEs et leur écriture en mémoire.

En ce qui concerne les RAMs, la RAM **WEIGHTS** contient les **poids** du réseau de neurones ainsi que certains paramètres de la couche, tels que le type de couche. Pour un réseau de ℓ couches, les *poids* et les paramètres des couches sont répartis couche par couche. Pour la RAM **FMAP**, elle contient initialement les *ifmaps*, puis pendant l'exécution, elle contient les sorties des couches intermédiaires qui sont de nouveau réécrites dans cette RAM. Enfin, à la toute fin de l'exécution, elle contient les *ofmaps*. Il est important de préciser que l'ordonnancement est conçu de manière à ce qu'il n'y ait jamais de lecture et d'écriture simultanées dans cette SRAM. Cela est possible car pendant la lecture, on lit des données utilisées pendant plusieurs cycles, ce qui évite de lire les données à chaque cycle, libérant ainsi des cycles auxquels l'écriture est possible. Enfin, il est important de préciser que pour les deux RAMs, la géométrie des RAMs ainsi que l'organisation des données dépendent des paramètres architecturaux $WPAR$ et $MPAR$.

Concernant l'implémentation de Gemini, il convient de noter que le NPU est conçu en utilisant la synthèse à haut niveau (HLS). Ce choix est justifié par le fait qu'elle utilise un langage de description de haut niveau comme le C++, ce qui facilite la configurabilité, la maintenabilité et la validation exhaustive (étant donné que les simulations sont

plus rapides qu'en RTL). De plus, la HLS permet d'accomplir efficacement certaines tâches telles que le déroulement de boucles et le pipeline. Toutefois, il est important de préciser que la description de l'architecture en HLS requiert une certaine adaptation du code pour obtenir des résultats optimisés. Seules les RAMs n'ont pas pu être décrites en HLS, elles ont été instanciées en RTL en utilisant les RAMd SPREGHD de STMicroelectronics.

La conception et l'implémentation de Gemini ont abouti à un tape-out en P18, qui a servi de premiers démonstrateurs. De plus, nous avons pu effectuer des simulations sur la netlist placée et routée, démontrant que Gemini est compétitif par rapport à d'autres accélérateurs, comme le montre le tableau 1.

Accelerator	PEs	Freq & Bits	Area	TOPs/W	GOPs/mm²
Envision (28nm) [92]	512	200 MHz & 8	1.87 mm²	3.80	-
ShiDiaNao (65nm) [40]	64	1 GHz & 16	0.66 mm²	-	293
Eyeriss(65nm) [25]	384	200 MHz & 8	12.25	0.246	-
UNPU (65nm) [79]	-	200 MHz & 8	16 mm²	4.30	43
QNAP (28nm) [91]	144	470 MHz & 8	1.9mm²	11.3	745
COMPAC(65nm) [107]	(128)	25 MHz & 8	1.74mm²	1.044	-
SCNN (65nm) [100]	64	1 GHz & 16	7.9mm²	-	-
Orlando (28nm) [38]	-	1.75 GHz & 8 and 16	34mm²	2.9	-
Gemini (18nm)	128	350 MHz & 8	0.655mm²	1.9/3.1	2560

Table 1 *Comparaison de KPI pour différent accelerateurs*

Évaluation des performances de Gemini

Comme présenté ci-dessous, Gemini possède une architecture configurable avec $(WPAR, MPAR)$, deux paramètres architecturaux qui dimensionnent l'ensemble du circuit et agissent donc sur les KPIs. Grâce à ces deux paramètres, Gemini peut être adapté pour différents marchés nécessitant différents compromis en termes de KPI. Pour cela, nous avons décidé de concevoir un évaluateur de performance à haut niveau permettant de prédire, pour un NN donné, la surface de la puce, la latence de l'exécution d'un NN et la consommation en fonction de $(WPAR, MPAR)$.

La méthodologie pour construire cet estimateur repose sur les principes suivants :

- Comme l'exécution d'un NN sur Gemini est prédictive, nous évaluons la latence Lat grâce à des formules analytiques. Il a été établi que

$Lat = \sum_{l=1}^{L} \left(\left\lceil \frac{\alpha_l}{MPAR} \right\rceil \times \left\lceil \frac{\beta_l}{WPAR} \right\rceil \gamma_l \right) + \sum_{l=1}^{K} \left\lceil \frac{\delta_l}{N} \right\rceil \epsilon_l$. Avec L le nombre de couches de convolution, K le nombre de couches entièrement connectées, et α_l, β_l, γ_l, δ_l, ϵ_l sont des constantes dépendant du type de couche l.

- La consommation statique et la surface ont été évaluées en prenant en compte la complexité des composants de Gemini, qui sont les unités de processeurs, les mixeurs d'entrée et l'étage d'écriture. La consommation statique et la surface sont donc modélisées par :
  $G = c_0 + c_1 \times N + c_2 \times N\lceil \log_2 WPAR \rceil + c_3 \times WPAR$; avec c_i, $i \in [0, 4]$, des constantes à déterminer.

- Pour la consommation dynamique, nous avons convenu de modéliser la consommation de chaque couche en fonction des différents paramètres qui la constituent. La consommation de l'ensemble du NN sera donnée par la moyenne des consommations des couches pondérées par leurs latences. Le principe pour prédire la consommation de chaque couche est obtenu avec la même équation que pour la consommation statique et la surface, sauf que les constantes sont désormais des fonctions dépendant du NN à exécuter.

Afin de déterminer ces constantes, nous avons recueilli des données en simulant 214 configurations différentes de $(WPAR, MPAR)$ et environ 100 NN à une couche. La détermination des constantes s'est faite par régression linéaire. L'estimateur a été validé pour tous les types de réseaux supportés et son efficacité a été illustrée sur le réseau VGG-like.

Grâce à cet estimateur, un utilisateur de Gemini peut déterminer quelle architecture choisir pour son réseau (en calculant des fronts de Pareto, par exemple).

Accélération de réseaux de Neurones en utilisant un pipeline fixé

Au cours de nos différentes évaluations de KPI, nous avons constaté qu'utiliser un seul NPU présentait des inconvénients. En effet, l'augmentation de la parallélisation ne résulte pas toujours en une réduction de latence. De plus, dimensionner un accélérateur par rapport à un NN fait que certains PEs ne sont utiles que pour les grandes couches,

et l'accélérateur se retrouve sous-utilisé pour les autres couches. Enfin, cette approche d'utiliser un seul NPU fait que les images sont traitées une par une.

Après une réévaluation de la structure du réseau de neurones, nous avons constaté qu'il fonctionnait tel un pipeline. Par conséquent, nous avons proposé un système de pipeline composé de n accélérateurs $G_i, i \in [0, n-1]$, séparés par des RAM R_i. Chaque NPU dispose d'un nombre fixe de PE, noté N_i, et les RAM ont une taille prédéfinie, notée s_i. Les données d'entrée (*ifmaps*) et de sortie (*ofmaps*) sont stockées dans les RAM **IFMAP RAM** et **OFMAP RAM**.

L'idée est que chaque NPU traite qu'une partie des couches de NN pour une image puis donne le relais au NPU suivant pour d'autres couches. Les *poids* sont répartis en différentes RAMs en fonction des couches supportées par chaque NPU. Le schéma 6.4 illustre ce principe.

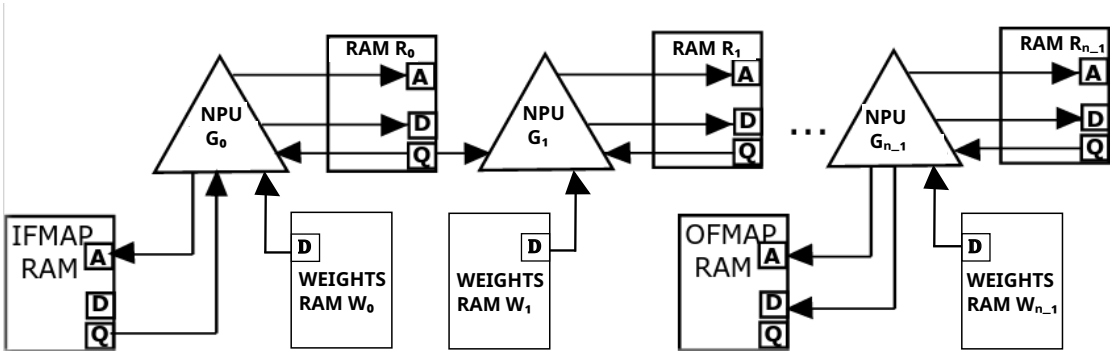

Fig. 4 *Description du pipeline de n NPUs*

L'objectif dans cette étude est de déterminer l'allocation de chaque couche sur le système, c'est-à-dire de déterminer quelle couche est traitée par quel NPU du pipeline. Ce mapping doit prendre en compte les contraintes liées aux tailles des RAMs. Ce problème s'apparente au problème d'assemblage SALBP [18], qui est prouvé comme étant NP difficile [10]. Cependant, dans notre cas, où nous avons des simplifications, nous pouvons le résoudre de façon polynomiale en nous inspirant de [57].

Nous avons donc modélisé le problème en programmation linéaire en nombre entier, pour lequel nous avons proposé des algorithmes polynomiaux permettant de résoudre trois sous-problèmes :

- Trouver le mapping permettant d'optimiser la latence Lat_2 du système si l'on considère que l'application repose sur des inférences ponctuelles. Ce qui signifie que les images sont traitées séquentiellement comme si l'on avait un seul NPU.

- Trouver le mapping permettant d'optimiser le débit du traitement. C'est équivalent à optimiser la latence Lat_1 pour une application de flux où les images peuvent être traitées simultanément.
- Enfin, comme les deux approches précédentes sont antagonistes, nous avons vu qu'il est possible d'optimiser la latence Lat_2 pour un débit donné.

  Nous avons illustré ces algorithmes sur un pipeline de plusieurs Gemini où nous avons montré qu'en fonction de l'objectif, le mapping peut changer.

Accélération de réseaux de Neurones en utilisant un pipeline non-fixé

Nous avons proposé d'étendre le pipeline vu précédemment au cas où la structure matérielle du pipeline ne serait pas fixée. Le nombre de NPU, ainsi que leur nombre de PEs et leurs tailles, ne sont plus figés. Avec cette liberté supplémentaire, il est désormais possible d'optimiser des KPI liés au matériel, souvent en conflit avec des critères de vitesse, tels que le débit.

Nous avons donc proposé d'optimiser, pour un débit donné, plusieurs KPIs, tels que la réduction du nombre total de processeurs, la surface, la puissance, l'énergie et une combinaison linéaire convexe de ces différents KPIs.

Pour résoudre ce problème, nous l'avons modélisé en utilisant un modèle linéaire en nombre entier. Ensuite, nous avons proposé un schéma de programmation dynamique fournissant une solution en complexité polynomiale.

Nous avons testé ces méthodes sur Gemini en comparant les solutions du pipeline aux solutions que nous aurions pu obtenir en utilisant un seul NPU avec un nombre élevé de processeurs. Le pipeline s'avère être très efficace pour réduire le nombre de PEs ou la surface afin d'atteindre un débit donné. Cependant, le plus impactant réside dans la possibilité d'atteindre des débits impossibles à atteindre en utilisant un seul NPU, notamment grâce au traitement simultané des images. De plus, cela permet de fonctionner à une fréquence plus basse, minimisant ainsi la puissance nécessaire pour un traitement à un débit donné.

Conclusion, contributions et perspectives

Cette thèse a apporté des contributions significatives à la fois sur le plan industriel et académique. Du côté industriel, au cours de cette thèse, j'ai participé à la conception et à l'implémentation de Gemini, un accélérateur matériel dédié à l'inférence de réseaux de neurones feed-forward comportant un certain nombre de couches standard. Cet accélérateur cible un marché de calcul en périphérie et répond aux contraintes spécifiques de ce domaine. De plus, Gemini a été conçu avec une grande flexibilité pour s'adapter aux besoins variés de différents clients. Il était donc naturel de développer un outil permettant aux clients de choisir la configuration de Gemini la plus adaptée à leur application.

Sur le plan académique, certaines des méthodes et outils développés au cours de cette thèse peuvent être réutilisés dans des contextes plus larges. Par exemple, l'outil de prédiction des KPIs peut être adapté pour d'autres accélérateurs en ajustant la méthodologie. De plus, le schéma de pipeline optimisant différentes KPIs pour l'exécution de réseaux de neurones est générique et peut être appliqué à divers accélérateurs. Ces travaux ont déjà donné lieu à une publication [12] et à un article soumis à une conférence internationale.

Plusieurs perspectives s'ouvrent pour améliorer davantage le support matériel de l'exécution des réseaux de neurones. Au cours de cette thèse, l'architecture de Gemini était prédéfinie, mais il est possible d'explorer le paradigme du calcul en mémoire, offrant une meilleure réutilisation des données et un potentiel de parallélisme accru. L'outil de prédiction développé peut également être adapté pour prendre en charge plusieurs réseaux de neurones, au lieu d'un seul. De même, le schéma de pipeline pourrait être conçu pour être adaptable à plusieurs réseaux de neurones plutôt qu'à un seul.

Ces perspectives s'inscrivent parfaitement dans l'idée d'améliorer à la fois l'optimisation locale (architecture) et globale (pipeline haut niveau), ce qui s'avère être la meilleure approche pour obtenir une optimisation complète du système.

ACKNOWLEDGEMENT

Je tiens tout d'abord à exprimer ma gratitude envers les membres du jury pour avoir évalué mes travaux et pour les discussions enrichissantes qui ont émergé lors de la défense. Je suis honoré que le sujet ait suscité votre intérêt. Mes remerciements s'adressent en premier lieu à M. Coussy et Mme. Kritikakou pour avoir accepté d'être mes rapporteurs. Je vous suis reconnaissant pour vos remarques et appréciations. Je souhaite également remercier M. Petrot et M. Pelcat pour avoir été examinateurs. Je tiens particulièrement à remercier Roselyne, non seulement pour son rôle d'examinatrice, mais aussi pour tout ce qu'elle a fait pour moi afin que cette thèse puisse aboutir. Sans l'opportunité qu'elle m'a offerte, je n'aurais pas pu finaliser administrativement ma thèse. Je te suis également reconnaissant pour le soutien que tu m'as apporté, ainsi que pour ta gentillesse et ton sens de l'aide, non seulement à mon égard, mais également envers tous ceux qui ont bénéficié de ton dévouement au laboratoire au quotidien. Ta présence au sein du laboratoire est d'une valeur inestimable et jamais suffisamment soulignée. Je tiens à exprimer ma profonde gratitude envers ma directrice de thèse, Alix. J'ai rencontré des difficultés lors de la rédaction de ce paragraphe, car il m'était difficile de trouver les mots justes pour exprimer à quel point je te suis reconnaissant pour avoir sauvé ma thèse. Sans ton intervention pour me motiver à poursuivre la thèse, à me redonner espoir et à retrouver ma bonne humeur, je ne serais pas parvenu à maintenir le cap durant cette période. Tes nombreuses interventions pour résoudre les problèmes administratifs ont été cruciales, et bien des fois, je n'aurais pas pu soutenir ou même rester en France sans ton intervention. Je tiens également à te remercier pour nos échanges fréquents, tant sur le plan scientifique où j'ai été passionné par ton expertise en recherche opérationnelle, que sur le plan personnel, où tu m'as permis d'avancer sereinement dans mes travaux au LIP6. Tes conseils et nos discussions sur d'autres sujets passionnants ont été d'une grande valeur pour moi.

Je remercie STMicroelectronics pour avoir cofinancé ma thèse. Je remercie Pascal pour avoir accepté de faire partie du jury. Mes remerciements vont également à Roberto pour son accompagnement et ses précieux conseils, ceux que j'ai déjà

suivis et ceux que j'espère pouvoir suivre à l'avenir. Je tiens également à remercier Bruno, Vincent, José, Mathieu, Carlo et tous les autres ingénieurs que j'ai eu l'opportunité de croiser à ST.

Je tiens à remercier mes amis grenoblois avec qui j'ai passé de bons moments et qui ont fait que mon déménagement à Paris n'était pas évident. En commençant par Soufiane et Thomas, qui ont été à mes côtés depuis le début jusqu'à la fin, et qui m'ont chaleureusement accueilli à chaque passage à Grenoble. Je souhaite également remercier mes amis rencontrés à ST, les Narvalos Clément, Emilien, Nathan et Fidel, avec qui j'ai partagé d'énormes fous rires, ainsi que de nombreuses discussions théoriques et d'optimisation sur divers sujets (le fromage blanc revenait souvent quand même ...). Je suis reconnaissant d'avoir passé tant de moments agréables avec vous en dehors du travail, et je vous remercie d'avoir réussi, par votre sympathie, à me faire rester lorsque j'envisageais de partir. Je tiens également à exprimer ma reconnaissance envers mes autres amis rencontrés pendant mon séjour à Grenoble, qui m'ont ruiné avec les nombreux allers-retours : Jésus, Chloé, Cécile, Lilia, Emma, Cédric, Nader, Anaïs, Florian, Stéphane, Marouane, Thibault, Marie, Paul, Simon, Eva, Gaëtan, Yura, Loïc, Quentin, Pauline, Martin et Félix.

Je souhaite exprimer ma gratitude envers mes amis parisiens qui m'ont accueilli, en commençant par ceux que j'ai rencontrés au laboratoire. Un grand merci à Maxime pour m'avoir immédiatement intégré et accueilli au sein du laboratoire. Je souhaite également remercier les autres avec qui j'ai partagé de nombreux moments de bonne humeur: Baptiste, Garance, Nathan, Clara, Mathuran, Theophilos, Habib, Rieul, Ilyas, Noé, Jonathan, Xin Yue, Adrien, Marie-Minerve, Gergana et Ouassim. Je suis reconnaissant d'avoir retrouvé les fantômes du passé, Anas.T et Abdou, mes meilleurs amis, que j'ai la chance de revoir très souvent maintenant. Mes remerciements vont également envers mon ami Ketsana, qui m'a beaucoup aidé et soutenu pendant mon séjour à Paris. Je remercie également ses amis et sa famille qui ont toujours été adorables avec moi. Je tiens également à exprimer ma gratitude envers mes autres amis chers, Anas B, Simo et Jad, que je n'ai pu voir que brièvement ces derniers temps, mais qui ont su remplir mon cœur à chaque rencontre.

Je souhaite exprimer ma gratitude envers mes parents et mon petit frère, qui sans eux, rien de tout cela n'aurait été possible. J'espère qu'ils sont fiers de moi et que cette thèse justifie ne serait-ce que peu le manque que l'on a pu ressentir. J'ai eu la chance d'avoir un petit frère aussi encourageant et aimant, qui se préoccupait de moi et de mes problèmes, et qui prenait tout son temps pour passer de bons moments avec moi lorsque je rentrais au Maroc. Mon père, qui a consenti à d'énormes sacrifices dès mon plus jeune âge et m'a transmis sa passion pour les sciences et les mathématiques, ce qui a guidé mon parcours. Je remercie également ma mère pour ses sacrifices, son soutien et sa présence lors de ma soutenance, malgré toutes les difficultés. Je présente mes excuses pour les épreuves qu'ils ont dû traverser et je leur suis reconnaissant pour leur soutien dans les moments difficiles.

Finalement, je tiens à exprimer ma gratitude envers mon grand-père, qui m'a toujours soutenu et qui a toujours fait l'impossible pour être présent dans les moments importants de ma vie. Sa place est incontestablement ici, parmi ceux que je remercie aujourd'hui.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

CHAPTER 1

Introduction

Throughout human history, individuals have always been natural problem solvers, shaping their understanding of the world through rational thinking and the art of problem-solving. Over time, this logical progression led to the development of algorithmic approaches, and subsequently, the use of computers to enhance our problem-solving capabilities. These machines played a crucial role in speeding up and improving our abilities, making it easier for us to handle complex challenges. As a result, we accomplished what was once considered impossible, breaking through the limits of what we thought we could achieve. Problems that seemed too difficult to solve due to their complexity became more manageable. From simulating weather patterns to modeling detailed molecular structures, these computer systems gave us the ability to take on tasks we could not have imagined before.

However, at one point, we realized that the human brain does not operate strictly based on rigid instructions. In fact, we could learn through observation, experience, and training, surpassing the boundaries of classical algorithms. This revelation paved the way for Artificial Intelligence (AI), a discipline that aimed to replicate the flexible and adaptable nature of human thinking. Within the field of Artificial Intelligence, what truly excels are neural networks (NN). These systems, inspired by the human brain, have demonstrated exceptional abilities in autonomously handling complex problems, such as those encountered in various AI applications. Some of today's widely renowned applications include virtual personal assistants like Siri and Alexa, recommendation systems like those used by streaming platforms, and even autonomous vehicles, which rely heavily on NNs to make critical decisions.

While the concept of neural networks (NNs) is not new, their recent resurgence can be partly attributed to significant hardware advancements [75]. The computational requirements of NNs are considerable, and a few decades ago, the required processing power simply was not available. Central Processing Units (CPUs), Field-Programmable Gate Arrays (FPGAs), and especially Graphics Processing Units (GPUs) played a pivotal role in reviving the potential of NNs by addressing these computational needs [76]. However, for NNs, Application-Specific Integrated Circuits (ASIC) accelerators have emerged as the most promising hardware candidates. They offer the potential to further reduce latency while maintaining a compact footprint and low power consumption.

These characteristics make ASIC accelerators particularly appealing for edge computing devices and the broader embedded systems market.

The competition to develop accelerators, specifically tailored for neural network inference, is underway. While training can be conducted on the cloud, the demand for efficient inference in this rapidly expanding market has led to the exploration of various architectures. Extensive hardware architecture optimizations at fine-grained levels, focusing on parallelization and data management, have given rise to a plethora of competitive accelerators. Moreover, as these accelerators serve a multitude of applications, the necessity of configuring their designs to adapt to markets with diverse used NNs and trade-off requirements has become important. Therefore, accurately predicting the performance of an accelerator across various configurations is crucial to selecting the most suitable one for each application.

With the availability of all these high-performance accelerators, a natural question arises: can we further optimize their use even more through high-level strategies? one approach involves using multiple instances of these accelerators to push performance boundaries even further and enhance configurability. For this task, the accelerator pipeline provides an excellent opportunity, notably due to its structure aligning with that of a neural network. Perhaps employing multiple accelerators with lower degrees of parallelism could yield greater efficacy than attempting to augment parallelization within a singular accelerator.

1.1 Context and Contributions

This thesis is conducted within a collaboration involving STMicroelectronics and the LIP6 laboratory at Sorbonne Université (CIFRE contract) supervised by Alix Munier Kordon. I spent the initial two years of my research journey exclusively at STMicroelectronics, where I was welcomed into the Innovation Team.

For nearly 19 months, I worked in a team dedicated to the Gemini project. The team consisted of one manager Pascal Urard, a senior technical manager Roberto Guizzetti, two other PhD students (Nathan Bain and Emilien Taly), an apprentice (Fidel Rodriguez Monteiro), and several interns. The project's objective was to design a configurable neural network inference accelerator (also called Gemini) for an internal client. The client specified the supported neural network type—specifically, feed-forward networks—along with the hardware architecture, and strict requirements for the accelerator, including constraints related to latency, chip area, power, and validation. Additionally, we were required to use High-Level Synthesis for configurability, prototyping, and validation facilities.

In the initial two months, we focused on creating the first version of an architecture

named Gemini-1, which did not have the optimal parallelization discussed in this thesis. Over the next 17 months, we worked on improving it, eventually arriving at the architecture described in the thesis. While the main emphasis of the Gemini project was on designing this Intellectual Property (IP), we had the added responsibility of delivering the complete package to the client. This included the source code that generated netlists and the validation environment. We also took on additional tasks, such as Tape-outs, using advanced ST technology. This allowed us to test the technology and, in exchange, gave us access to silicon for measuring the IP's performance. We performed these activities for both Gemini-1 and the more advanced architecture, referred to as Gemini-2, which is detailed in this thesis.

After completing the design and satisfying the initial client's needs, the accelerator became available for potential use by other clients. This presented the challenge of configuring the accelerator effectively to meet diverse clients' requirements. To address this, I collaborated with the laboratory, in conjunction with STMicroelectronics. Together, we devised a strategy to efficiently estimate performance such as area, power, and latency—of an accelerator at a high-level, considering specific neural network and architecture combinations. These performance estimation models were developed using industrial tools available at STMicroelectronics.
These performances estimations rely on just two architectural parameters and neural network parameters. This enabled us to select the most suitable architecture for each client's application. This developed tool predicts performances in CMOS C40, facilitating the selection of the optimal configuration based on performance requirements. This work also resulted in a publication set for the SBAC-PAD 2023 conference [98]. The entire task, including writing the paper, spanned approximately one year.

Furthermore, while analyzing the performance and its variations based on configurations, we identified several barriers limiting fine-grained optimizations of accelerator architectures. By delving deeper into the neural network structure, we realized the potential for pipelining multiple neural network accelerators to further optimize processing. We initially explored a scenario where the pipeline architecture was fixed, as if we were end-users of a chip with multiple fixed pipeline accelerators, seeking to determine which neural network layer to execute on each accelerator for each incoming neural network, optimizing latency, throughput, or finding trade-offs among them. Subsequently, we expanded this problem from a user's perspective to that of a designer, who had the freedom to choose the accelerators' number and their architectures. In this context, we aimed to optimize not only execution speed but also hardware performance. As these criteria often conflicted, we decided to optimize several performance metrics at a fixed throughput. Specifically, for a given neural network and a fixed throughput requirement, we aimed to determine the number of accelerators, their architectures, and

the allocation of layers to these accelerators to optimize multiple performance metrics. For both pipeline problems, we provided a tool to solve them efficiently. Note that the fundamental building block of this tool is the performance estimator mentioned in the previous paragraph.
The problem related to non-fixed hardware led to a submission to an international conference.

The work on the pipeline was conducted at LIP6, where I physically joined the team in March 2023. Upon the conclusion of the CIFRE contract in May 2023, I found myself without sufficient material to draft the manuscript due to the incomplete status of the pipeline study. In response, LIP6 provided me with a part-time 6-month contract to support my thesis. During this period, I was able to finalize the pipeline study and complete the manuscript and conference papers writing.

In summary, this thesis has made significant contributions in four main areas:

Development of a feed-forward configurable neural network inference accelerator: this thesis has actively contributed to the implementation of an efficient and configurable neural network accelerator using High-Level Synthesis. This included the creation of a validation framework. This IP is currently in use at STMicroelectronics.

Performance prediction framework: the research work resulted in the design of a performance prediction framework that facilitates the estimation of latency, area, and power at a high-level of abstraction for various neural networks and hardware configurations of the Gemini accelerator. This framework assists in selecting the most suitable configuration for specific applications. It is currently utilized within the innovation team, particularly for the CMOS C40 technology. Additionally, this research served as the foundation for a publication at the SBAC-PAD 2023 conference [98].

Fixed pipelined neural network accelerator framework: the thesis has made a significant contribution by developing a framework to enable the utilization of accelerators in a fixed pipeline configuration. In this configuration, various parameters such as the number of accelerators, their specific arrangement, the size of RAMs, and their architecture are fixed. The framework finds the best neural network layers mapping on the accelerators to optimize either the throughput or the latency, or a trade-off between them.

Non-fixed pipelined neural network accelerator framework: in contrast to the previous scenario, this framework addresses a dynamic design challenge where the pipeline configuration is not fixed. The primary objective here is to determine the optimal hardware parameters, including the number of accelerators, their configurations, order, and RAM sizes as well as the neural network layers mapping to this pipeline. This optimization process targets a range of performance metrics, encompassing latency, the total number of processing elements, power consumption, area utilization, energy, or a

combination of these factors, all while adhering to predefined throughput constraints. Additionally, the aim is to assess whether this flexible pipeline approach outperforms a single accelerator with a high number of processing elements.
While this strategy has shown promising results for a Gemini accelerator pipeline, it has not yet been implemented on a chip. However, the overall pipeline strategy applied to any accelerator and its effectiveness on Gemini has resulted in a submission to an international conference and the presentation of a poster at GDR SOC2 2023.

1.2 Dissertation organization

This dissertation is structured into eight chapters:

In Chapter 2, we provide a foundational understanding of neural networks. We begin with a brief historical overview of their development and then delve into the core features and components of neural networks, including various types of networks, common layers, and activation functions. We also highlight key applications and benchmarked neural networks, emphasizing those relevant to the Gemini project.

Chapter 3 focuses on the hardware foundations of the Gemini IP architecture. We start with an overview of hardware accelerator architectures, followed by a detailed exploration of dataflow strategies. Key Performance Indicators (KPIs) used in the domain are discussed, and we elucidate how quantization affects these KPIs.

Chapter 4 details the design and implementation of Gemini to meet customer requirements. We explain data representation within Gemini, provide an architectural overview, including the neural processing unit (NPU) and RAMs, and briefly discuss layer scheduling. We also highlight the impact of design choices leveraging High-Level Synthesis tools. It concludes with a comparison of Gemini's KPIs with state-of-the-art accelerators.

Chapter 5 focuses on the critical aspect of configurability within the Gemini accelerator architecture. It aims to build a high-level of abstraction model that predicts Gemini KPIs (including latency, area, and power) based on hardware architecture and supported neural networks. It starts by describing the environment used for collecting simulation data used for the model. Furthermore, the chapter explains how the KPI model is constructed and utilized, employing Pareto fronts to facilitate hardware configuration selection.

In Chapter 6, we tackle the problem of fixed pipelined neural network accelerators. We explore the potential of using multiple fixed instances of accelerators in a pipeline structure to enhance execution KPIs (latency and throughput). This chapter presents the pipeline environment, mapping layers onto accelerators, and related literature. We then introduce linear models and polynomial solutions to different sub-problems, illustrating

the methodology through a pipelined scenario with Gemini instances.

Chapter 7 transitions from fixed hardware to a scenario of non-fixed pipelined neural networks accelerators. We explore the environment, related works, and new optimization objectives, including hardware performance. This chapter offers modeling and resolution approaches and presents solutions tailored to the Gemini accelerator. Additionally, we discuss potential extensions of this paradigm.

Finally, in Chapter 8, we conclude this manuscript by summarizing the key findings and contributions of the research presented throughout the dissertation. We provide insights into the implications of our work and suggest avenues for future research in the field.

CHAPTER 2

Introduction to Neural Networks

The initial objective of my thesis was to contribute to the design of Gemini, a hardware accelerator for neural network inference.

Deep Neural Networks, a subset of Artificial Intelligence (AI), enable machines to learn without explicit programming, revolutionizing the field. Brain-inspired computation, drawing inspiration from the human brain, plays a crucial role in shaping these algorithms. Neurons, the fundamental units of the brain, are interconnected through dendrites and axons, receiving and transmitting signals, known as activations, across synapses, of which there are approximately 10^{14} to 10^{15} in the human brain. Importantly, synapses possess the capacity to modulate signals by adjusting *weights* (w_i). It is through these dynamic synaptic *weight* modifications that the brain is believed to learn. This property of dynamically adjusting *weights* within a relatively stable neural structure provides a promising foundation for the development of machine learning algorithms. The basic mathematical representation of a neuron is illustrated in Figure 2.1. With y the output of the neuron, w_i the *weights*, x_i the inputs of the neurons, b the bias

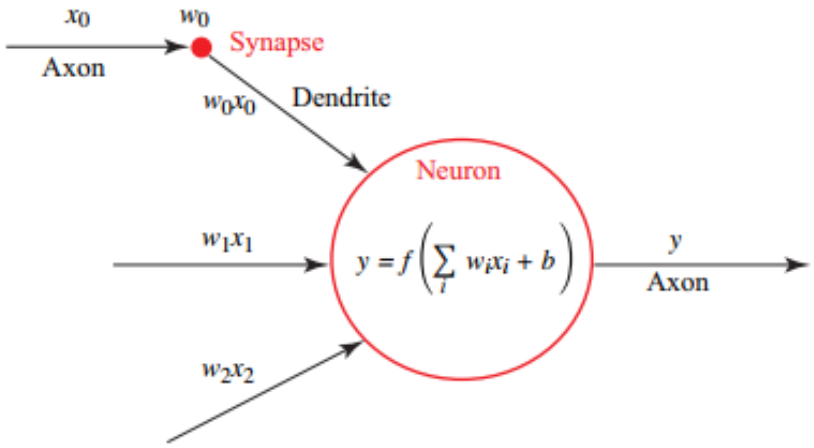

Fig. 2.1 *Neurons connection in the brain and its mathematical model. Figure inspired by [118]*

and f the nonlinear activation function detailed later. This mathematical representation is the one that must be implemented efficiently in hardware.

In the field of brain-inspired computing, there is a specialized area known as spiking computing. This area draws inspiration from the way the brain communicates, where

signals resemble spike-like pulses. What makes spiking computing unique is that it recognizes that the conveyed information is not solely determined by the size of these spikes. Instead, it depends on factors like when the pulse arrives and how the neuron processes it. This processing is not just based on a single value but also considers the width of the pulse and the timing of different pulses. An example of this research is the IBM TrueNorth project [90]. Our focus in on classical artificial NN as imposed by the project (presented in Section 1.1).

On the other hand, machine learning, in general, is primarily divided into supervised and unsupervised learning. Supervised learning deals with labeled data, where the algorithm learns from a dataset for which both the inputs and their corresponding outputs are known. The objective is to construct a model that can generalize observations from the known data to make predictions on unseen data. Examples of supervised learning tasks include classification (e.g., image recognition) and regression. In unsupervised learning, the data is unlabeled. Here, the goal is not to make predictions, but rather to explore the data's structure and identify correlations among its elements. A notable application of unsupervised learning is clustering, where common characteristics within the dataset are discovered. Finally, there is the semi-supervised that combines elements of both supervised and unsupervised learning by using a mixture of labeled and unlabeled data. This approach aligns more closely with real-world scenarios, where labeling data involves experts participation. Typically, a preliminary unsupervised learning step is employed to clean the data before applying supervised learning methods.

In the vast majority of cases, neural networks are primarily utilized for supervised learning, where a data for training phase is necessary.

In this chapter, we will provide a comprehensive understanding of the neural network that serves as the foundation for the Gemini accelerator. Our exploration begins with a brief historical context of neural networks in Section 2.1. Subsequently, in Section 2.2, we present an overview of key aspects of neural networks, discussing their fundamental features and associated challenges. This section also serves as an opportunity to establish the necessary vocabulary and notation. Topics covered here encompass the structure of neural networks, including their inputs and outputs, layer configurations, and details regarding training and inference processes. Then, Section 2.3 delves into the practical applications of neural networks, providing insights into various use cases, benchmarked neural networks, and NNs used within the Gemini project. Finally, we provide a concise conclusion in Section 2.4.

2.1 Neural Networks History

The journey of Artificial Neural Network research started as far back as the 1940s. It was Donald Hebb who laid the foundation by emphasizing that neural pathways strengthen with use, a fundamental concept in human learning [20]. However, it was not until 1965 that Ivakhnenko and Lapa proposed an early working learning algorithm with a multi-layer neuron structure [63]. Throughout the 1980s, neural network research showed promise, but traditional von Neumann computer architecture [113] continued to dominate the computing landscape. A significant turning point arrived in 1989 when LeCun et al. [77, 73] introduced a Convolutional Neural Network (CNN) using the back-propagation for digit recognition. This pioneering network structure became the prototype for modern neural networks. The early 2010s witnessed a surge in applications based on NNs, marked by notable achievements like Microsoft's speech recognition system in 2011 [37] and the introduction of AlexNet, a NN for image recognition in 2012 [71]. These advancements participate in the growing impact and potential of NNs utilization in various domains. More recently, deep learning techniques have found application in increasingly challenging and futuristic domains, achieving performance levels previously unattainable with conventional approaches, and in some cases, even surpassing human performance (in image recognition [46]). Several key factors have contributed to the rapid progress of Deep Neural Networks (DNNs) in recent years [118, 75]:

- Abundance of training data: the availability of vast amounts of training data has played a pivotal role. Companies like Facebook, Walmart, and YouTube handle immense volumes of data, providing the substantial datasets needed to effectively train these algorithms.
- Advancements in computing power: progress in semiconductor technology and computer architecture has significantly increased computing power. This advancement enables the efficient computation of the complex weighted sums required for DNNs, both during training and inference (detailed in Subsection 2.2.5).
- Development of open-source frameworks: the early successes in NN applications spurred the development of open-source frameworks. These frameworks have made it more accessible for researchers and practitioners to work with NNs. These collective efforts, coupled with evolving algorithmic techniques, have greatly improved accuracy and expanded the range of domains in which NNs can be applied.

2.2 Neural Networks Overview

In this section, we will discuss all the essential features of a neural network that must be supported by the Gemini accelerator.

We will start with an explanation of neural network inputs and outputs, along with their respective notations, in Section 2.2.1. Subsequently, in Subsection 2.2.2, we will delve into the various layer connections within a neural network, specifying the NN types defined by their connectivity. Among these layers, we will provide descriptions of those utilized in the neural networks supported by Gemini in Subsection 2.2.3. Moving forward, we will discuss activation functions, which are crucial for separating layers within a neural network structure, in Subsection 2.2.4. Lastly, in Subsection 2.2.5, we will cover the challenges posed by the training and inference phases of neural networks, particularly in terms of accuracy.

2.2.1 Inputs and Outputs

Before explaining NNs architecture, we will describe their inputs and outputs:
The inputs of a NN are $3D$ *input feature maps* (*ifmap*). A NN can receive a batch of k *ifmaps* to process. The figure 2.2 shows that an *ifmap* is composed of C channels, each channel has $W \times H$ pixels with W the width and H the height of the $2D$ image.

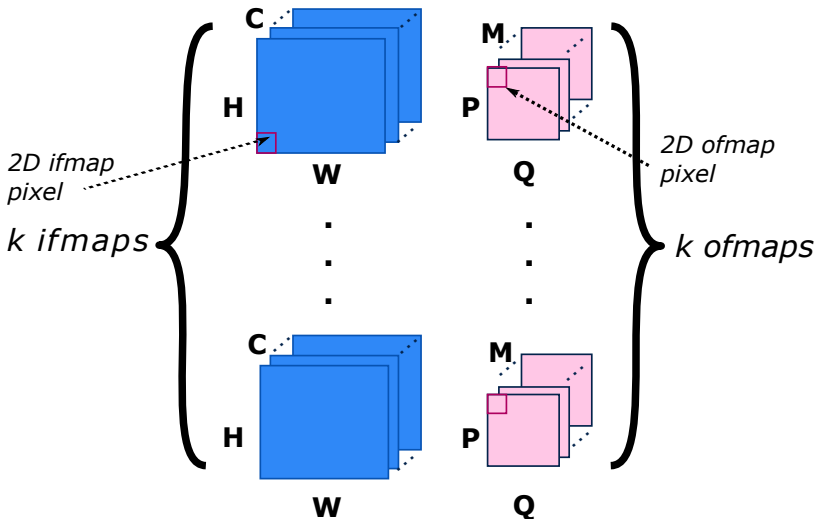

Fig. 2.2 *Neural Network's ifmaps and ofmaps. Inspired by [118]*

Output feature maps (*ofmap*) are the output of the NN model, they have M channels, P height and Q width. So each channel has $P \times Q$ pixels.

Similarly, these neural network inputs and outputs are the ones employed throughout every layer comprising the network.

Figure 2.3 shows the structure of a NN with its corresponding inputs and outputs.

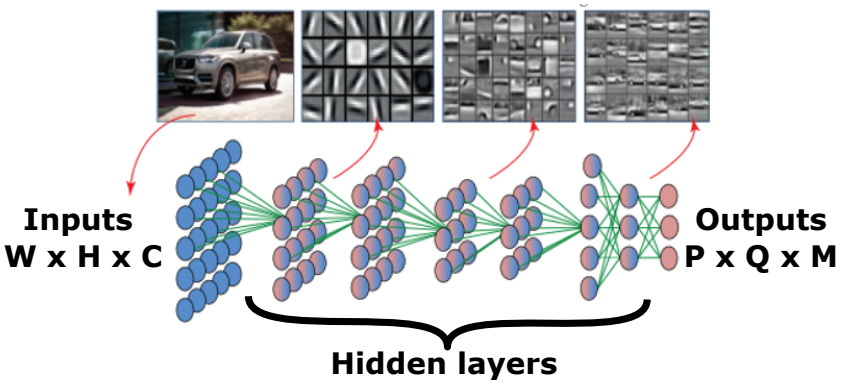

Fig. 2.3 *Structure of a neural network*

2.2.2 Neural Networks Layers Connection

Neural networks are composed of successive layers, potentially separated by an activation function. The first and last layers, referred to as the input and output layers, respectively, while the intermediate layers are known as hidden layers. A layer in an NN is characterized by two key elements [118]:

- Connections between inputs and outputs: these connections determine the nature of the layer. There are fully connected layers where all input elements are connected to all output elements, as illustrated on the left side of Figure 2.4 between the input layer and the hidden one. Conversely, there are sparsely-connected layers where not all inputs are connected to all outputs, as depicted on the right side of the same figure, where the hidden layer is not connected to all outputs. This type of connection is associated with a receptive field, where outputs are sensitive only to specific inputs. For example, an output can be specific to an input and its nearby neighbors. Sparsely-connected layers encompass various layer types, such as convolutions, pooling, depthwise layers, and more.
- Connection *weights*: connection *weights* represent the unique values associated with connections between inputs and outputs within a layer. These *weights* can be either zero or non-zero and do not alter the fundamental nature of the layer.

It is essential to note that the absence of connections in sparsely-connected layers is not represented by a *zero-weight* value. Instead, the structure itself determines which inputs affect the outputs. For convolutions, this is achieved through the use of filters, as we will detail later. On the other hand, the nature of connections between layers characterizes the type of NN. There are primarily two types:

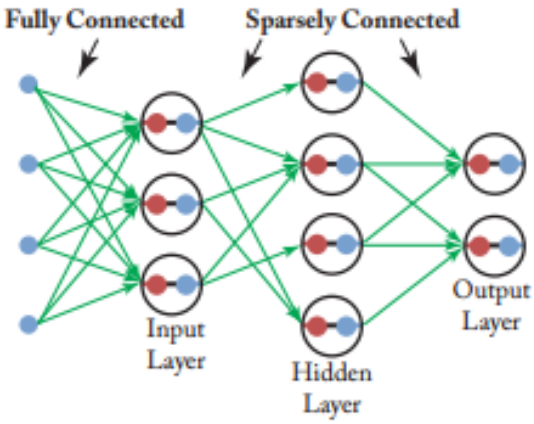

Fig. 2.4 *Fully connected layer VS sparsely-connected layer (from [118])*

- Feed-forward NNs: the output of one layer is connected to the input of the following layer, and there are no backward connections from the output to the input (no loops). This implies that feed-forward NNs lack memory; once the output of a layer is fed to the next one, it can be discarded as it is no longer needed.
- Recurrent NNs (RNNs): in contrast, RNNs allow an output of a layer to be used as input, creating a loop within the network. This means that the output of a layer depends not only on the current input but also on its previous output. This mechanism introduces a form of memory within the NN. RNNs are particularly suited for handling sequential data because they can incorporate information from previous inputs. One well-known type of RNN is the Long Short-Term Memory (LSTM) NN, commonly used for processing sequential data such as text or speech modeling and processing [116]. Figure 2.5 illustrates the difference between feed-forward and recurrent NNs, with a loop in the recurrent NN indicating the use of previous output in the calculation of subsequent outputs.

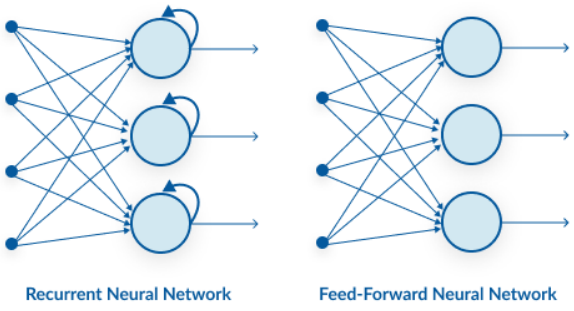

Fig. 2.5 *Feed-forward and recurrent NN*

Throughout this thesis, our focus will primarily be on feed-forward NNs, for several reasons. Feed-forward NNs are not only the most commonly used type for our project

(as specified in Section 1.1), but they also enjoy widespread popularity across various applications, particularly in image processing.

Furthermore, the resurgence of feed-forward NNs in recent years can be attributed to the development of transformers, which provide a viable alternative to the use of RNNs for processing sequential data, as demonstrated by Vaswani et al. in their work on attention mechanisms [122]. Transformers employ encoders and decoders that are implemented using feed-forward NNs paradigm.

However, feed-forward NNs definition has evolved to encompass other types of layers that preserve data throughout the network. This extension includes NNs with residual blocks, where the output of an NN is produced and used for the subsequent layer without overwriting it; instead, it is summed with the outputs of other layers. This approach gained popularity with networks like ResNet [54]. Figure 2.6 illustrates this concept where data x of a layer has to be further summed to the output $F(x)$. Additionally, there are NNs that concatenate layer outputs instead of adding them, as exemplified by [81]. Although our study may not encompass these specific types of feed

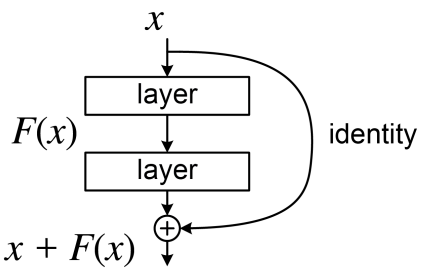

Fig. 2.6 *Residual blocks connexion*

forward NNs, their implementation does not require a fundamentally different strategy from what we will explore. The necessary features can be easily incorporated to our different studies.

2.2.3 Neural Network Layers Supported

As highlighted in Section 1.1, Gemini is specifically designed for feed-forward neural networks, as detailed in Subsection 2.2.2. It offers support for a range of layer types, encompassing convolutional-like layers (comprising convolutional layers, depthwise layers, and pooling layers), as well as fully connected layers. These various layer types will be further detailed in this subsection.

2.2.3.1 Convolutional-like Layers

These layers are characterized by sparse connections (detailed in Subsection 2.2.2). They process $3D$ images, referred to as input feature maps (*ifmaps*), with dimensions including height H, width W, and the number of channels C. The primary operation, apart from pooling as discussed in the following paragraph, involves Multiplication and ACcumulation (MAC) (it is the fundamental operation of a neuron as seen in Figure 2.1).

The convolution layer's *weights* are structured as $3D$ filters with dimensions comprising height R, width S, and the number of input channels C, which matches the *ifmaps*' channel number. Additionally, a $1D$ scalar known as *bias* is added to the result. To compute a single output feature map *ofmap* pixel, each filter convolves with the corresponding *ifmap* region. Specifically, each *ifmap* pixel is multiplied by the corresponding filter *weight*. In Figure 2.7, the red section illustrates the computation of first the top-left *ofmap* pixel, denoted as O_0. This corresponds to the result of the convolution between the filter and the top-left *ifmap* region: $O_0 = \sum_{i=0}^{S\times R\times C-1} W_i \times I_i + B_0$. Here, W_i represents the filter *weights*, and I_i represents the *ifmap* pixels. Other pixels from the same *ofmap* are generated by sliding the filter to cover all *ifmap* pixels. In our study, following conventions inspired by [118], the filter traversal begins from the top-left and proceeds to the bottom-right of the *ifmap*. It should be noted that the output of

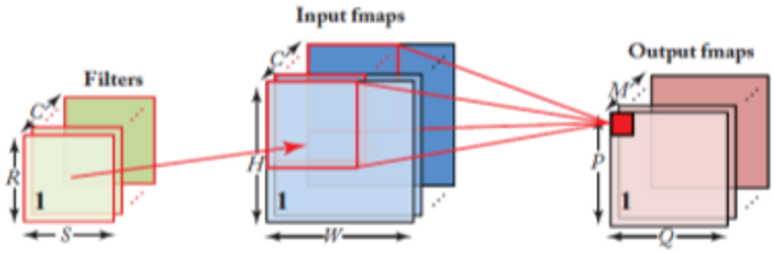

Fig. 2.7 *Convolutional layer computation*

convolving an $3D$ *ifmap* with a filter results in a single-channel $2D$ *ofmap*. To obtain an $3D$ output feature map with M channels, a set of M filters is required. Consequently, the number of channels in the *ofmap* corresponds to the number of filters applied.

The size of the resulting $2D$ *ofmap* depends on several key parameters:

- Stride: this parameter determines the step size at which the filter is shifted to compute the next portion of the *ofmap*. It is important to note that using a *stride* greater than one reduces the channel size of the *ifmap* in comparison to the original *ifmap*. We differentiate between vertical stride ($stride_v$) and horizontal stride ($stride_h$), depending on whether we are sliding vertically (bottom) or horizontally (rightward).

- Padding: there are two types of *padding*:
  - Valid padding: in this mode, only the valid pixels of the *ifmap* are considered, meaning the filter window does not extend beyond the boundaries of the *ifmap*. This results in a reduction of the channel size of the *ofmap* compared to its original size.
  - Same padding: the goal here is to maintain the same $2D$ size as the original *ifmap* after convolution. To achieve this, the filter may extend beyond the *ifmap* boundaries. In this case, the value assumed by the *ifmap* pixels outside the boundaries is typically set to zero.

  For both types of padding, we distinguish between vertical *padding* pad_v when crossing the boundaries of the *ifmap* in the top and bottom directions, and horizontal *padding* pad_h when the filter extends beyond the *ifmap* boundaries on the left and right. For both pad_h and pad_v, they are set to 0 for *valid padding* and 1 for *same padding*.

Similarly, other parameters, such as dilation (well explained in [53]), could be considered, but they are not within the scope of our study.

In summary, after a convolution operation, we obtain an *ofmap* consisting of M channels, with the size of each channel determined by the formula: $\frac{W-(S-1)\times pad_h}{stride_h} \times \frac{H-(R-1)\times pad_v}{stride_v}$.

Pooling layers are also crucial in feed-forward NNs. They share similarities with convolutions, particularly in their use of filters with similar parameters such as *stride* and *padding*. These layers reduce the spacial resolution of a feature map (downsampling). Several pooling operations can be operated (average, maximum, minimum); in our study, the maxpool is the most used. In contrast to convolution, where the operation involves the multiplication of the filter *weights* with the *ifmap* pixels, pooling layers do not employ *weights*. The pooling operation, such as maximum pooling, is computed among the various pixels covered by the filter. For instance, in Figure 2.8, the *ofmap* pixel 20 is computed by taking the maximum value among the *ifmap* pixels covered by the red filter: $\max(12, 20, 8, 12)$. To calculate the remaining *ofmap* pixels, the filter is moved through the *ifmap*. Another distinct characteristic of pooling layers

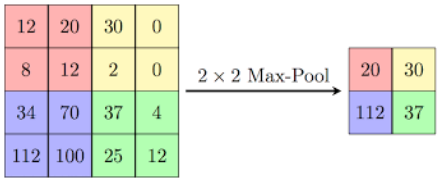

Fig. 2.8 *Maxpool example*

is that the filter has a single dimension, and the number of filters M must match the

number of *ifmap* channels: pooling preserves the same number of channels. The size of the *ifmap* channel is also contingent on the *padding* and *stride* and can be determined by the following formula: $\frac{W-(S-1)pad_h}{stride_h}\frac{(H-(R-1)pad_v}{stride_v}$.
Another convolution-like layer that has gained widespread use is **the depthwise separable convolution**. This layer type gained popularity following the introduction of the MobileNet architecture by Chollet [28]. The primary objective was to replace standard convolutions, which involve multiple parameters, with depthwise separable convolutions. The depthwise separable convolution is divided into two distinct operations: the **depthwise** operation and the **pointwise operation**:

- Depthwise: the depthwise layer is similar to convolution in that it also involves *weights* grouped into filters. However, there is a distinction in the calculation process. In this layer, each *ifmap* channel is processed separately, with one filter dedicated to each channel (where each filter has a single channel). Consequently, each filter is slid across a single *ifmap* channel. The result of the depthwise operation is then an *ofmap* with the same number of channels as the original *ifmap*. The size of the channels follows the same equation as in convolution, with the only difference being that the number of filters is set to C, and each filter has just one channel.
- Pointwise: this is a traditional convolutional layer employing filters with dimensions of $1 \times 1 \times C$. To generate an *ofmap* with M channels, M filters are required.

Figure 2.9 shows an example of depthwise separable convolution, the *ifmap* channels are processed separately and the *ofmap* of the depthwise layer has also 3 channels. During the pontwise, filters with $1 \times 1 \times 3$ dimensions are used. The utilization of

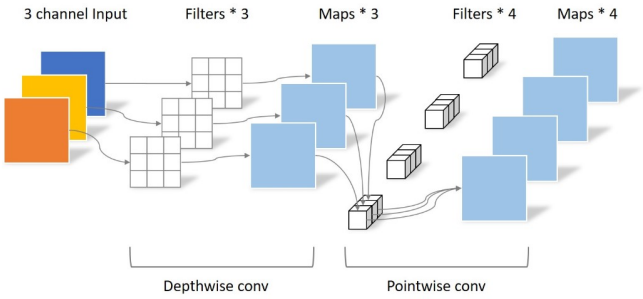

Fig. 2.9 *Example of depthwise separable convolution*

this layer enables NNs to possess fewer parameters compared to those employing full convolutions. Moreover, when properly trained, networks with depthwise separable convolutions can achieve high accuracy, even with a reduced parameter count [14].

2.2.3.2 Fully connected Layers

This is the most standard layer type. The $3D$ *ifmap* is flattened into N_{in} $1D$ *input neurons*, and the output is also represented as a $1D$ flattened *ofmap* referred to as *output neurons*, with a total of N_{out} output neurons.

Each *output neuron* is connected to all *input neurons* through synapse *weights*. Consequently, every *output neuron* is the sum of the *weight* values of the *input neurons*, to which a *bias* (there are N_{out} biases) is added. The number of *weights* in a fully connected layer is therefore $N_{in} \times N_{out}$.

This layer is commonly viewed as a matrix multiplication between the *weights* and the *input neurons*, followed by the addition of *biases*. Figure 2.10 illustrates a fully connected layer and its representation in a matrix product for $N_{in} = 3$ and $N_{out} = 2$. The computation of the output pixel for each layer is summarized in Table 2.1. Note

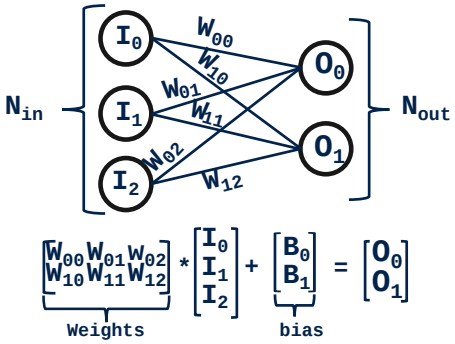

Fig. 2.10 *Example of fully connected layer*

Table 2.1 *Table summarizing outputs pixels computation for different NN layers*

Layer type	*Ofmap* pixel equation
Convolution	$O_{s,r,m} = \sum_{i=0}^{S-1} \sum_{j=0}^{R-1} \sum_{c=0}^{C-1} W_{i,j,c} \times I_{s+i,j+i,c} + B_m$
Maxpool	$O_{s,r,m} = \max_{i \in [0,R_1], j \in [0,S-1]} I_{s+i,r+j,c}$
Depthwise	$O_{s,r,c} = \sum_{i=0}^{S-1} \sum_{j=0}^{R-1} W_{i,j,c} \times I_{s+i,r+j,c} + B_c$
Fully connected	$N_i = \sum_{j=0}^{S \times R \times C-1} I_j \times W_j + B_i$

that for convolution, we have omitted the *stride* and *padding* for the sake of readability.

For an *ifmap* or *ofmap* pixel $O_{m,s,r}$, r represents the width, s the height, and m the channel index. In the case of fully connected layers, for input or output neurons N_i, i is the index of the neuron, considering a flattening order of channels, rows, and then columns.

2.2.4 Non-linear Activation

A NN model consists of several consecutive layers, with a non-linear activation function applied between each pair of layers. Specifically, inserting two layers without any non-linear activation functions between them is equivalent to having just one single layer. Adding a non-linear activation function effectively separates these two layers.

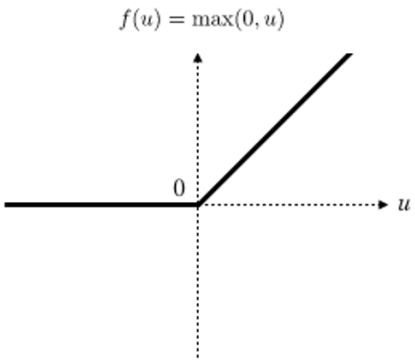

Fig. 2.11 *ReLu function*

Traditionally, activation functions such as sigmoid and hyperbolic tangents were commonly used. However, in hardware implementations, the rectified linear unit (ReLU) is preferred due to its simplicity. In Figure 2.11, a ReLU is represented, where negative values are set to zero, and positive values follow the identity function. There are other alternatives that can be used as well, such as the ”swish” [104] or ”exponential ReLU” activation functions. Finally, note that activation is not always required after pooling layers. In this thesis, we will utilize the ReLU activation function.

2.2.5 Training VS Inference

As mentioned in the chapter's introduction, the Gemini project is focused on supervised learning, which necessitates both training and inference phases. As a reminder, the NN model consists of several layers characterized by *weights*. These *weights*, as highlighted in Section 2.1, mimic the human ability to learn. This learning process involves adjusting the values of *weights* (as well as *biases*), which is accomplished during the **training** phase. Once this phase is completed, the NN is fixed and can be used for predictions, a step referred to as **inference**. Here is an overview of these two phases:

Training: the goal of training is to find the *weights* that maximize the probability of the NN making accurate predictions. In the case of supervised learning, this is achieved by maximizing the probability of correctly predicting the outputs while minimizing the probability of incorrect predictions. In supervised learning, training is conducted on a dataset where the correct output for each input is known. The difference between the ideal correct probabilities and the probabilities computed by the NN based on its current *weights* is referred to as the loss (L). There are several ways to define the loss, with one of the most common being mean squared error: $L = \frac{1}{m}\sum_{i=0}^{m-1}(y_i - \widehat{y_i})^2$, where y_i represents the prediction of the NN, $\widehat{y_i}$ is the ideal value provided by the dataset, and m is the number of predicted classes. The *weights* are updated to minimize the loss. This update of *weights* for the purpose of minimization is achieved through optimizers. A survey by Abdulkadirov et al. [5] covers several optimization methods, but one of the most well-known and widely used is gradient descent. For each iteration t in the range $[0, T-1]$, the *weight* $w_{i,j}$ (where i is the layer index and j is the index within the same layer) is updated as follows: $w_{i,j}^{t+1} = w_{i,j}^{t} + \frac{\partial L}{\partial w_{i,j}}$. This equation is computed in two steps: first, forward propagation (similar to the inference process described later) is performed to calculate the predictions and loss, and then, a second step of backpropagation calculates the gradients and propagates the loss backward through the network's layers, tuning the *weights* of each layer accordingly [50].

An important consideration in training is the selection of the framework and dataset. To facilitate the development of NNs and promote the exchange of pre-trained models, numerous deep learning frameworks have emerged from various origins. These open-source libraries encompass a range of software tools designed for NNs. For instance, Caffe [65], originating from UC Berkeley in 2014, provides support for programming languages such as *C, C++*, *Python*, and *MATLAB*. On the other hand, Google introduced TensorFlow [3] in 2015, which is compatible with both *C++* and *Python*. TensorFlow also offers versatility by accommodating multiple CPUs and GPUs. It also stands out for its flexibility of adapting features. Finally, there is also Torch, which was developed by Facebook and NYU and supports *C, C++*, and *Lua*. PyTorch [101] is its successor and is built in *Python*. In the context of our thesis, TensorFlow was imposed as the framework of choice since it is used by our customers for designing their NN. Several datasets were used for training networks in the context of this thesis. The most commonly used ones include CIFAR [70], MNIST [74], and CELEbA [83].

Inference: in this step, the *weights* are already fixed, and the inference is performed by computing the NN layers in a forward manner.

In supervised learning, the initial dataset is typically divided into two subsets: one for learning and another for inference, which is used to assess the model's quality. The quality of inference, which can also be evaluated at the end of training, is of-

ten measured in terms of *accuracy*. *Accuracy* quantifies how well a neural network performs in making correct predictions or classifications and is calculated as follows: $Accuracy = \frac{\text{(Number of Correct Predictions)}}{\text{(Total Number of Predictions}}$
In this calculation, the predicted class is typically determined by the highest probability. A higher accuracy value indicates better performance. Additionally, other metrics, such as the *F1-score*, can be employed (although not used in this thesis). The F1-score offers a more comprehensive assessment, as it considers both *precision* and *recall*: $Precision = \frac{\text{(True Positives)}}{\text{(True Positives + False Positives)}}$; $Recall = \frac{\text{(True Positives)}}{\text{(True Positives + False Negatives)}}$. $F1 - Score = 2 * \frac{(Precision \times Recall)}{(Precision + Recall)}$.
The F1-score is particularly useful when dealing with imbalanced datasets.

For this thesis, only the inference phase is considered, as the Gemini IP project (described in Section 1.1) solely focuses on inference. This approach is logical because the project targets edge computing for fast, low-power, and small-area applications. Therefore, there is no need to implement the training process on such devices; training is typically performed offline and involves fixing the *weights* for the inference phase executed at the edge.

2.3 Neural Network Applications

In our modern world, Neural Networks are widely employed in various areas, and their effectiveness is unquestionable. Every day, we witness the emergence of fresh applications, highlighting their continuous expansion. However, the complexity required for different NNs varies depending on their specific use cases. In this section, we will start in Subsection 2.3.1 by presenting the wide-ranging applications of Neural Networks. Then, in Subsection 2.3.2 we will present some bench-marked NNs with a focus on the ones used for the Gemini project.

2.3.1 Diverse Use-cases

Numerous fields benefit from the extensive utilization of Neural Networks, with some of the most renowned ones including:

- Computer vision: it is a field dedicated to enabling computers to efficiently process and understand visual data, including images and videos. Its ultimate goal is to replicate human-like visual perception. In this context, neural networks have emerged as invaluable tools for advancing computer vision applications by processing videos [29] or images [88, 26] for several applications: object detection [48], recognition, estimation, positioning, event detection [114], scene reconstruction, image restoration [94], editing, video enhancement, and statistical learning

[6]. This field is even more appealing when considering that 70% of the data comprises videos [30].

- Speech and language: this field encompasses several applications, including translation, audio generation [131], natural language processing [59], and chatbots [66]. Its popularity has been on the rise, especially with the advent of tools like ChatGPT or Google Bard that could answer questions and provide text-based answers [34].
- Biomedical and medicine: neural networks have proven their effectiveness in medical diagnosis. This is primarily due to their capacity to process and analyze extensive patient data, outperforming the diagnostic capabilities of individual doctors over a human lifetime. Actually, they have demonstrated remarkable performance in the diagnosis of cancers [112]. In the biomedical sector, NNs have made substantial contributions to various domains, with a noteworthy breakthrough observed in the field of DNA sequencing [129].

The Gemini project, as described in Section 1.1, was originally conceived to support computer vision NNs but, in reality, can accommodate any feed-forward neural network and, consequently, any application.

2.3.2 Benchmarked and Utilized Neural Network

In this section, we will focus on famous NNs especially used for computer vision. Some NNs gained popularity through the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [106]. In this competition, the objective was object detection and classification, leading to the emergence of several widely used NNs: AlexNet [71] in 2012, GoogLeNet [120], VGG [115] in 2014 and, ResNet [55] in 2015. These NNs rely on using numerous layers with millions of parameters to maximize accuracy, and their performance has driven their widespread usage.

Another field that influences the design of specific NN architectures is edge computing. NNs are designed with a limited number of operations to cater to the edge market, which includes embedded devices running on batteries or requiring on-chip RAM, limiting the number of operations and parameter storage. Examples of such edge-friendly architectures include MobileNet [61] and SqueezeNet [62]. Furthermore, pruning techniques, as presented by He and Xiao [56] in a survey, enable the reduction of parameters in a NN. With these techniques, the number of NN parameters can be reduced to be supported by embedded devices.

Table 2.2 provides an overview of some benchmark NNs, including those used in the context of this thesis.

In this project, the neural networks used are primarily designed for edge computing

Neural Network	Number of operations
LENET-5	60,000
GoogLeNet	6,800,000
AlexNet	60,000,000
VGG-16	138,000,000
SqueezeNet	1,250,000
Tiny YOLO	6,000,000
YOLO v3	62,375,348
MobileNet x0.25	850,000
PNet	7,900
VGG-like	526,000

Table 2.2 *Benchmarked and used neural networks, along with their parameters.*

applications, with a specific focus on embedded computer vision tasks. This introduces a constraint on the choice of neural networks, as they must be suitable for efficient computation on edge devices. Given the limited area available on such devices, it is not feasible to employ gigantic SRAM that could store large NNs with millions of *weights*. Therefore, the application is constrained to NNs of moderate size that can be accommodated within the hardware limitations.

Some of the NNs used for validation and study in Gemini project (Section 1.1) include: MobileNet x0.25 [61] with 27 layers predominantly comprising depthwise separable convolutions, VGG-like (shown in Figure 2.12), which is inspired by VGG-16 [115] and consists of 11 layers combining convolution, maxpools, and fully connected layers (that are relatively large), and P-Net with 7 layers [130], containing the same layers as the previous ones. These networks have *weights* sizes of 850,000, 526,000, and 7,900, respectively, and input feature map (*fmaps*) sizes of 224x224x3, 128x128x1, and 32x32x1, respectively.

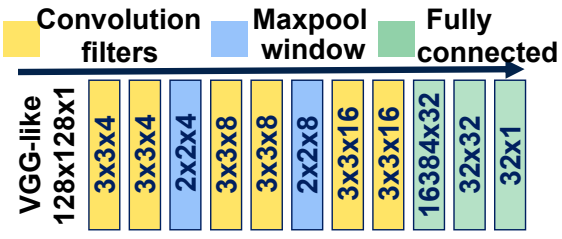

Fig. 2.12 *VGG-like network structure*

2.4 Conclusion

In this chapter, we have provided an overview of all the neural networks features used in this thesis. These networks have evolved significantly since their modest introduction

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Fig. 2.13 *MobileNet network structure*

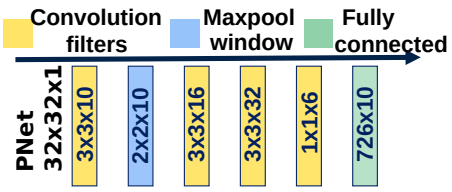

Fig. 2.14 *PNet network structure*

in the 1940's, now featuring complex multi-layer architectures that demand extensive computations.

Within the context of the Gemini project, our focus lies on feed-forward neural networks that support various layers, including convolution, depthwise, pooling, and fully connected layers. The ReLU activation function is applied between these layers. Note that this project exclusively handles inference, while training is conducted offline. Although neural networks find diverse applications, our study concentrates on edge computing with moderate-sized neural networks. The primary models studied include MobileNet, VGG-like, and P-Net.

Now that we have identified the features of the neural networks to processed, the next step is selecting the appropriate hardware support to efficiently implement these networks. This hardware must optimize several key performance indicators, which we will delve into in Chapter 3.

CHAPTER 3

Neural Networks Hardware Accelerators

The evolution of Neural Network (NN) inference acceleration has been marked by a decisive shift away from traditional Central Processing Units (CPU) and Graphics Processing Units (GPU) technologies employed in server environments towards specialized hardware solutions known as Application-Specific Integrated Circuits (ASICs) finely tuned to operate at the edge

The development of edge computing accelerators was driven by the need to minimize both area and power consumption. In comparison to solutions based on GPUs and CPUs, these accelerators offer compact, energy-efficient alternatives. Additionally, having the accelerator integrated on the same chip as sensors and actuators results in power savings by eliminating the need to transfer data externally. This integration allows for immediate processing following data acquisition, reducing latency, enhancing safety, and addressing security risks associated with data transfer.

In this chapter, we are solely focused on integrated circuit accelerators, particularly ASICs, even though similar considerations apply to Field Programmable Gate Arrays (FPGAs). The optimization of GPUs or CPUs is beyond the scope of this study.

The selection of the Gemini architecture, which was mandated for our project, was made specifically to enable support for multiple feed-forward Neural Networks. It is important to note that designing an efficient accelerator for multiple NNs is a considerably more complex task than developing hardware tailored to a specific NN. In the latter case, the order of operations is predetermined, leading to a simpler and fundamentally different implementation compared to hardware that must accommodate various NNs whose parameters (layers and layers parameters) are not known in advance. Additionally, our target is configurable architecture, which allows for tuning the level of parallelization to adapt the hardware to specific applications. This flexibility is essential in ensuring optimal performance across a range of use cases.

Additionally, the choice of the RAM paradigm for Gemini was indirectly influenced by certain constraints. Given our target for embedded devices with limitations on area and power, we made the decision to exclusively utilize on-chip SRAMs, avoiding the use of DRAM due to its costly communication and larger footprint. This choice of relying solely on on-chip RAMs necessitates that the RAM stores the entire Neural Network without a progressive filling approach of the RAMs. Additionally, it is im-

portant to note that In-Memory Computing implementation was not feasible at the time due to the unavailability of the corresponding RAM technology. Hence, our focus is directed towards architectures where on-chip RAMs, along with the Neural Processing Unit (NPU), are utilized, following the Near Memory Computing (NMC) approach. In this paradigm, computations performed by the NPU are carried out directly on the data supplied by the RAMs located on the same chip.

Crucially, this accelerator was developed within the constraints of an industrial project, driven by the company's desire to obtain silicon and have it ready for testing during the thesis. This prioritized rapid implementation and low-level optimization over exploring various architectural possibilities, which explains why the architecture was mandated.

In this chapter, our focus is on an architecture that adhere to these considerations. It should support various NNs, offers configurability, and adopts a near-memory computing approach (with NPUs and RAMs). To structure our exploration, this chapter is divided into several sections: in Section 3.1, we will begin by discussing the fundamental aspects of NN accelerators. Following that, in Section 3.2, we will explore the different categories of data-flows typically selected for NN accelerators. Our examination continues in Section 3.3, where we will discuss the key performance indicators commonly used to evaluate and compare accelerator performance. Section 3.4 will provide a detailed look at the critical topic of data quantization within accelerators and its role in achieving improved performance. Finally, in Section 3.5, we will conclude this chapter by summarizing the key insights and implications discussed throughout.

3.1 Principles of Neural Network Accelerator Architectures

As mentioned in the preceding paragraph, the accelerators studied comprise both NPUs and SRAMs. While optimization efforts are primarily focused on NPUs, adjustments in SRAM organization are necessary. The NPU serves as the engine responsible for performing the computations required to generate the *ofmaps*. A basic overview of the accelerator architectures is illustrated in Figure 3.1.

Let us provide a top-down description: we have both off-chip DRAM and on-chip components. Typically, DRAM stores the entire neural network. On the chip, there is a global buffer, which serves as a second hierarchical memory level with limited storage. In our example, this could be an SRAM. Since we are not considering DRAM in this context, the global buffer needs to store the entire neural network. Next, we have the NPU itself, which includes all Processing Elements (PEs) and the interconnections which encompass connections between the global buffer and PEs, and among the PEs themselves. In the literature [118], these interconnections are referred to as the

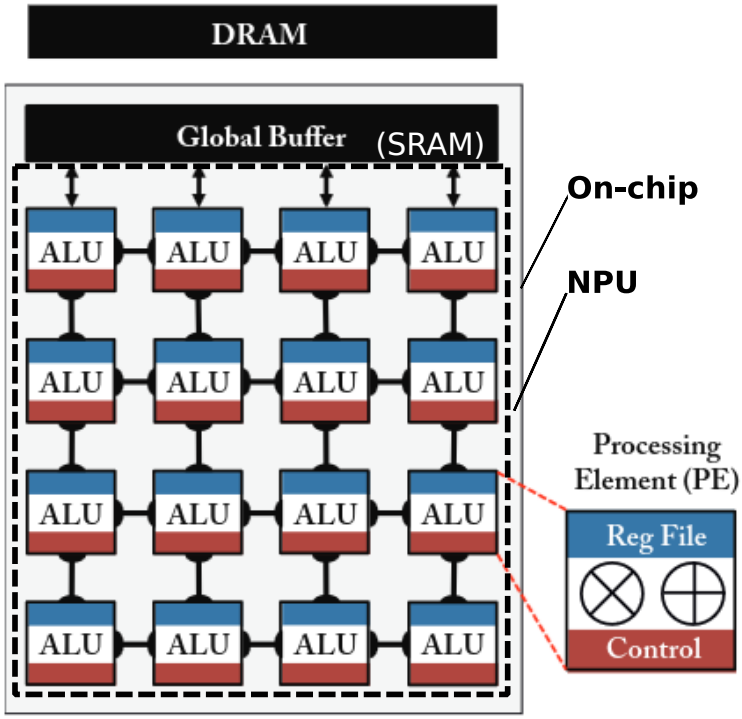

Fig. 3.1 *Generic architecture for an NN accelerator. Figure adapted from [118]*

Network On Chip (NOC). The Processing Elements are the blocks responsible for executing operations within the layers, such as MACs for convolutions, depthwise and fully connected layers, and maximum operations for maxpools, for example. They contain the necessary operators (multipliers, adders), control logic (to determine which operations to perform), and a register file for storing locally processed data, constituting a third hierarchical memory level.

3.2 Neural Network Accelerators Data-flows

The data-flow within the accelerator is optimized to improve performance, taking into account two crucial aspects. Firstly, the optimization efforts are centered on power reduction, driven by the fact that data movement governs dynamic power consumption, as highlighted by Sze et al. [118] and Horowitz [60]. To address this issue, the architectures maximize data reuse, thereby minimizing data movement. Secondly, latency is optimized by fully exploiting parallelization opportunities within the layers to achieve optimal computation. The potential parallelization within NNs are detailed in [118, 4].

Three traditional classes of data-flow architectures enable parallelization and data reuse: the input, weight, and output stationary paradigms. It is important to acknowledge the existence of alternative paradigms that combine principles from these three classes, as row stationary seen in Eyeriss [25], as well as paradigms that do not rely on data reuse [128]. These three paradigms are illustrated in Figure 3.2. Here is a brieve

explanation of them:

In the **input stationary data-flow,** the register file of the Processing Elements (PEs) stores *fmap* pixels, as illustrated in Figure 3.2 (c). For convolution-like layers, each *input feature map* (*ifmap*) pixel is reused for all MACs (or maximum for maxpools) that use it before flushing it: actually, each *ifmap* pixel is used for all M filters and for each filter it can be used for a maximum of $S \times R$ when sliding the filter window on the *ifmap*. This maximum utilization is feasible only when the *padding* is *same*. In this paradigm, the filter *weights* are broadcasted every cycle, and the accumulation of partial sums contributing to *onput feature map* (*ofmap*) pixel calculation is distributed across different PEs. This introduces complexity in managing the sharing of partial sum flow among the PEs, which varies depending on the sizes of the filters. For fully connected layers, an input neuron can be reused for all N_{out} output neurons. Finally, the latency is increased as several MACs are operated simultaneously. The input stationary data-flow has been applied in the SCNN accelerator [100].

In the **weight stationary data-flow,** each PE stores a *weight* in its local register, while *ifmaps* pixels are broadcasted every cycle, as seen in Figure 3.2 (a). Each *weight* can potentially be used for all $2D$ *ofmap* pixels ($W \times H$, when *padding* is *same*). Similar to the previous paradigm, partial sums flow across multiple PEs to produce the final *ofmap* pixel. For fully connected layers, an output neuron can be reused for all N_{in} input neurons. Latency is enhanced as each PE performs a MAC operation. Several works have explored this data-flow approach [24, 49, 67, 135].

The **Output stationary data-flow** is the one imposed by the Gemini project. It is the most regular data-flow. In this paradigm, each PE calculates its own *ofmap* pixel, and the partial sum of each *ofmap* pixel remains stationary inside the PE and is not shared among PEs, as shown in Figure 3.2 (b). Unlike the two previous paradigms, there is no need for control mechanisms to determine which PEs should share information. The execution is predictable for all PEs working simultaneously. Additionally, there is no need for additional operators to combine the results of different PEs. Parallelization is achieved for all simultaneously calculable outputs. For fully connected layers, a maximum of N_{out} outputs can be processed concurrently. In convolution-like layers, it can be applied to $P \times Q$ pixels of the same $2D$ *ofmap* and each pixel of the M filters to process. Some architectures implement simultaneous computing of $P \times Q$ for the same *2D ofmap* [40], while others process only one pixel of each *ofmap* channel [103]. Some architectures combine both approaches, such as [93] or Gemini (presented in Chapter 4).

Furthermore, in output stationary data-flows, in addition to the reuse of partial sums, it is also possible to reuse *ifmaps* pixels and *weights* with an efficient scheduling. In this approach, the data is not stored in the register files of PEs. Instead, data read from the

global buffer during one cycle can be utilized in subsequent cycles without requiring another read, as long as the data remains unchanged (for SRAMs, this means reading from the same address twice). Since the data remains stationary in the global buffer, it can be reused across different cycles. This data reuse is less regular compared to the two previous data-flows, but can still offer benefits in terms of reducing power consumption.

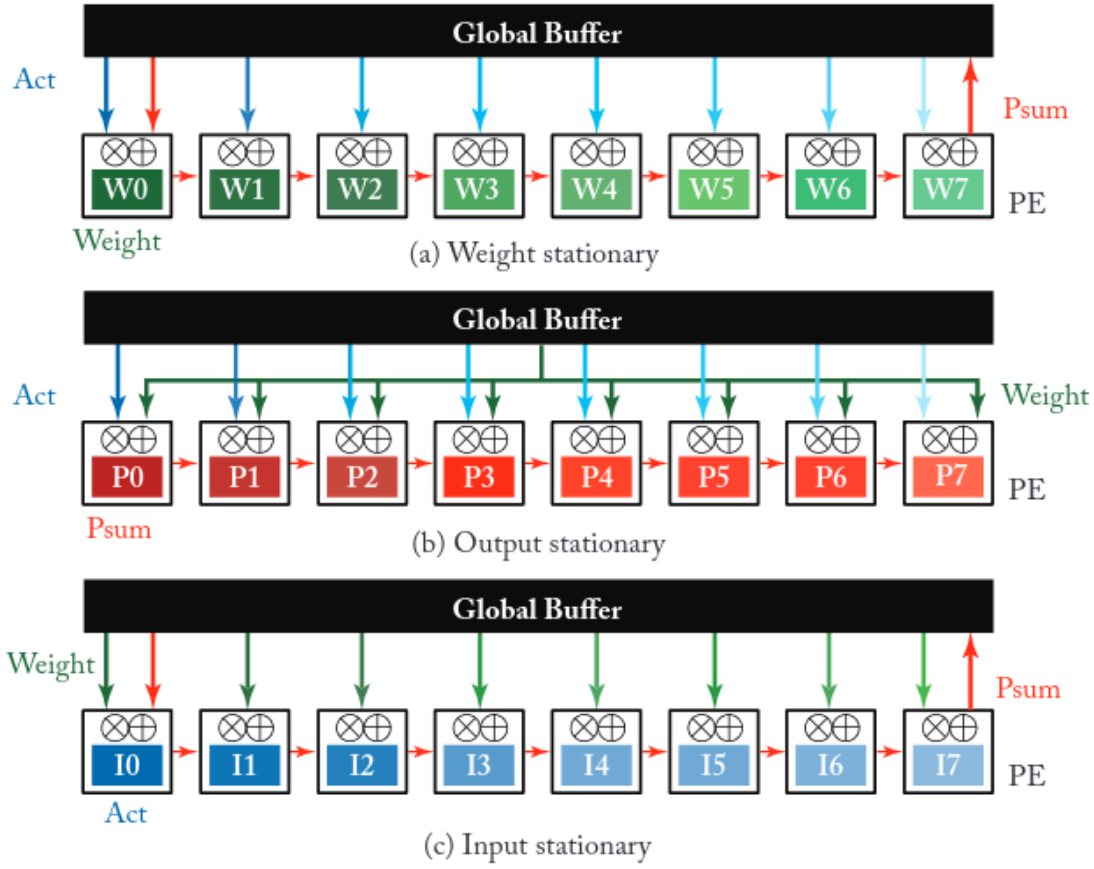

Fig. 3.2 *Most common NN accelerators data-flows. From [118]*

In conclusion, the output stationary architecture was indeed imposed for Gemini, primarily due to its advantages in terms of data reuse and parallelization , as well as its regularity. However, this regularity also presents its primary drawback. Since all outputs are calculated uniformly, it becomes challenging to effectively leverage data sparsity and prevent redundant calculations for specific PEs and specific data instances. This challenge becomes particularly pronounced when the data being processed contains a significant number of zero values in both *fmaps* and *weights* (when quantized) [118]. This limitation arises because all PEs simultaneously perform identical operations on all data.

3.3 Key Performance Indicators of the Design

The objective of the Gemini team is to design an accelerator that implement efficiently neural networks. In order to evaluate Gemini performance and compare it with other state-of-the-art accelerators (done in Subsection 4.4.2), several Key Performance Indicators (KPI) must be taken into account.

In this section, we start by highlighting the difference between latency and throughput in Subsection 3.3.1. Following that, Subsection 3.3.2 provides an explanation of chip area metrics. In Subsection 3.3.3, we delve into the characterization of power and energy aspects, and lastly, in Subsection 3.3.4, we spotlight additional KPIs employed to characterize neural network accelerators.

3.3.1 Latency and Throughput

Latency, denoted as Lat, represents the time taken by a system, measured in clock cycles, to produce an output from a given input. In the context of NN inference, latency corresponds to the duration required to calculate the inference of an input feature map (*ifmap*) by a NN model. It can also be expressed in seconds by dividing the number of clock cycles by the frequency of processing.

On the other hand, throughput, denoted as T, refers to the volume of data processed within a specified time frame, typically expressed as the number of frame inferences per second. In our context, each frame corresponds to a single inference. It is important to note that latency and throughput are distinct concepts. In some scenarios, they may be related, such as when a new input is loaded only after the previous one has been fully processed.

The definition of throughput is dependent on the specific NN being processed. It can be derived as shown by [118]: $T = \frac{operations}{second} \times \frac{1}{\frac{operations}{second}}$. The first term, denoted as $T' = \frac{\text{operations}}{\text{second}}$, is independent of the particular NN and is often used to compare accelerators without the need to run the same NN. Several papers prefer to use T' as the definition of throughput to eliminate the dependence on the specific NN being processed. However, calculating T' can be complex. While calculating T only requires counting the number of cycles needed for an inference (and then compute the number of inferences per second), T' necessitates counting operations. Unfortunately, the definition of an operation can vary from one paper to another. Some count only the operations in the processing elements, while others include those used in scheduling (such finite state machines operations or the mixers used for data transfer). Additionally, processing elements (PEs) are not always fully utilized during the whole processing, so cycles when they are unused need to be considered or not, depending on conventions. The definition that takes these considerations into account is given by [118]:

$T' = \frac{operations}{second}$ = operations/second for 1 PE $\times$ number of PEs $\times$ utilisation of PEs.

To prevent any confusion and ensure meaningful comparisons, we will use the first definition of throughput, denoted as T, and use it for comparisons whenever possible. This approach will help eliminate discrepancies arising from different definitions and provide a consistent basis for evaluation.

3.3.2 Chip Area

The area of an NN accelerator is a crucial consideration. It is generally encompassing storage (RAMs) and the NPU. It is typically expressed in mm². RAM sizing depends on the NN and processed *ifmaps*, while NPU area relies on its hardware architecture. Comparing accelerator area becomes complex when some include RAMs while others do not. Such comparisons lack meaning, as complexity can shift from NPUs to RAMs. For instance, data replication in RAMs inflates their size, whereas efficient scheduling and data reuse require extra NPU hardware resources. Moreover, technology differences further complicate area comparisons. Although gate count comparisons abstract away technology specifics, it is essential to acknowledge that technology also influences scheduling and resource allocation, impacting this parameter.

3.3.3 Power and Energy Consumption

Power and energy consumption are critical factors for NN accelerators, especially in battery-powered devices. Power is the sum of leakage and dynamic power:

- Leakage power: also known as static power, it is the power consumed by a digital circuit when it is not actively performing computation. This power is due to subthreshold leakage current in transistors. It flows even when transistors are in the off state. It is exponential with temperature for several technologies [137]. In NN accelerators, minimizing leakage power is essential for energy efficiency when the accelerator is in an IDLE state.
- Dynamic Power: this is the power consumed by a digital circuit when actively performing computation or transitioning between different logic states. It primarily results from the charging and discharging of capacitances in the circuit as data is processed and signals change. Dynamic power is measured during NN inference and depends on the specific NN being used. It can be quantified for each clock cycle, but it is typically measured as an average over the entire execution, which is the most commonly used approach.

As it was highlighted for the area, the power is dependent on the technology. Additionally, different papers may or may not consider the power of RAMs in their evaluations.

The energy E in joules for an accelerator is also crucial. In this study, we focus on the energy consumed during one inference, representing the energy dissipated when an *ifmap* is entirely processed to produce the corresponding *ofmap*. For a specific task, energy can be computed directly from the power P using the formula: $E = \int_{Lat} P(t)dt$ where Lat is the latency of the task. If power is constant over time, this simplifies to: $E = Lat \times P$. It is the case when considering the average power.

3.3.4 Other KPIs

Additional Key Performance Indicators (KPIs) are emerging, such as energy efficiency metrics that combine throughput and power. These metrics are often expressed in units like $\frac{\text{Ops/s}}{\text{W}}$ or $\frac{\text{Ops}}{\text{Watt} \times mm^2}$, where Ops represents the number of operations, W is for watts, and mm^2 represents the area unit. These custom KPIs serve several purposes:

- Independence from NN: by dividing throughput by power, these metrics abstract away the specific neural network being used, allowing for comparisons across different networks.
- Trade-off evaluation: these metrics help capture the trade-off between different aspects of performance. For instance, increasing throughput often involves increasing parallelization, which can result in higher area and power consumption. Observing only the throughput will mask the cost in hardware. By taking into account a combination of all KPIs, the trade-off is effectively evaluated.
- Highlighting specific accelerators: some custom KPIs may emphasize specific strengths of an accelerator compared to competition, so they are preferred for publications.

Additionally, there are KPIs that aim to abstract from the impact of technology by normalizing the values with respect to the technology node. These metrics provide a more technology-agnostic perspective for evaluation.

3.4 Quantization

Quantization is the process of reducing the precision of data (including *weights* and *fmaps*) used in calculations, by representing them as a restricted set of discrete values in fixed-point notation. While this technique enhances hardware performance, it does so at the expense of reduced precision. It involves the conversion of floating-point data into fixed-point representations within a specified bit-width and value range. Quantization plays a crucial role in optimizing neural network accelerators for edge computing, offering several key benefits:

- Reduced storage and data transfer: quantization decreases the amount of data that needs to be stored in RAMs and transferred between the NPU and RAMs. This reduction in data size can lead to the use of smaller RAMs, resulting in decreased area and leakage power. Furthermore, since data movement is a major contributor to dynamic power consumption, quantization saves a significant amount of power by minimizing the data that needs to be moved [118, 60].
- Simplified operations: fixed-point operations with lower bit widths are less complex and require less power and area compared to full floating-point operations. For example, consider the energy consumption and area requirements of different multipliers: a 32-bit floating-point multiplier consumes around $3.7pJ$, a 32-bit fixed-point multiplier consumes approximately $3.2pJ$, and an 8-bit fixed-point multiplier only requires about $0.2pJ$. In terms of area, they occupy approximately $7700\mu m^2$, $3485\mu m^2$, and $282\mu m^2$, respectively. These figures highlight the benefits of using low bits fixed-point representation for reducing energy and area requirements for operators [118, 60].
- Maintained accuracy: interestingly, despite the reduction in arithmetic precision, quantization often does not lead to a substantial loss in accuracy [47]. This makes it an attractive technique for edge computing scenarios where power and area efficiency are crucial, and a minor reduction in accuracy is acceptable

The objective then of quantization is to reduce precision while maintaining high accuracy.

Quantization involves converting floating-point data into a range of possible integers, and it is characterized by the number of bits used for both *weights*, denoted as *weightbits*, and feature maps, referred to as *fmapbits*. Various types of quantization methods exist [12], all aiming to decrease the number of values (number of quantization bits) while simultaneously minimizing quantization errors, which represent the average difference between the original full-precision data and the quantized reduced-precision representation.

3.5 Conclusion

In this chapter, we introduce hardware accelerators designed to support the neural networks discussed in Chapter 2. This description is framed within the context of the Gemini project, where a Near-Memory Computing architecture was selected. The chapter begins by outlining the principles of such hardware accelerator architectures. Subsequently, we explore the data-flow strategies used to enhance parallelization, facilitating speed improvements and data reuse, which, result in power savings. It is important

to note that the architecture mandated for the Gemini accelerator aligns with output-stationary data-flows.

Following this, we present various key performance indicators (KPIs) commonly employed in this field to characterize accelerator performance and facilitate comparisons. We also delve into the details of data representation within accelerators, with a focus on quantization, which offers further performance optimization.

Once the foundational aspects of hardware accelerators are covered, we can delve deeper into design choices and challenges in realizing output-stationary data-flows for Gemini. Furthermore, having a well-conceived design is not sufficient; an efficient implementation is equally crucial. Chapter 4 will provide an explanation of the design and implementation of Gemini. Subsequently, we can evaluate the performance of the Gemini accelerator using the KPIs described earlier.

CHAPTER 4

Gemini Design and Implementation

Gemini neural network accelerator is designed as an industrial Near Memory Computing (NMC) solution for Neural Network (NN) inference tasks (all NN *weights* are pre-computed). It was initiated as part of an internal project at STMicroelectronics, with the primary goal of creating a configurable NN accelerator for the edge computing market. As elucidated in Chapter 3, the architecture was predefined for this design. It adheres to a near memory computing paradigm, featuring a Neural Processing Unit (NPU) and RAMs integrated within the same chip. Furthermore, it has to operate on an output stationary data-flow model. The success of the project relies on meeting a set of customer-specified specifications, which must be considered throughout the design and implementation phases:

- Gemini must generate accurate outputs in TensorFlow-compatible [3] format and handle 8-bit data for both *feature maps* (*fmaps*) and *weights*, following TensorFlow's quantization. However, exhaustive validating Gemini's correctness is a challenging task due to its support for multiple neural networks and diverse architectural configurations.
- Despite its primary focus on edge computing, particularly for embedded devices, Gemini must remain configurable to meet customer-specific requirements. It can serve different customers with diverse needs in terms of speed, area, and power, achieved through simple adjustments in hardware architectural parameters.
- Performance optimization is imperative, addressing key performance indicators in this domain, including power consumption, area, and processing speed, all while maintaining high inference accuracy.
- Gemini must support multiple types of NN layers, including convolutions, depthwise layers, fully connected layers, and pooling layers.
- Finally, ensuring the design's readability and maintainability is crucial. Given that Gemini serves as a prototype solution intended for internal customers, it is imperative that the design be thoroughly comprehensible. Additionally, the rapidly evolving nature of the NN domain necessitates the ability to incorporate new features to remain in step with advancements.

This chapter will delve into how these constraints influence design and implementation choices.

Gemini's purpose is to support the processing of a batch of k ifmaps (in $3D$) by a NN to produce a batch of k ofmaps (in $3D$). The inputs of Gemini can be divided into two categories: the NN description and the ifmaps.

The NN description refers to all information relative to the NN. For each layer of the NN, it encompasses the *weights* and the layers parameters:

- The *weights* used during processing: in the case of convolutional and depthwise layers, they correspond to the filter *weights* and *biases*. For fully connected layers, they correspond to the synaptic *weights* and *biases*.
- Parameters: for each layer, some information must be given to the hardware to process the *ifmap* by the NN. For example, we have the *ifmap* dimensions, the NN parameters (as filter sizes for convolutions or output neurons number for fully connected ones) and parameters specific to each layer (as *padding* type or *stride* value for convolutions).

The ***ifmaps***: Gemini receives a batch of k *ifmaps* to be processed sequentially by the NN. The *ifmaps* received directly by Gemini are the ones to be processed by the first layer. The *ifmaps* can be different according to the NN layer: for convolutional, dephtwise and pooling layers the *ifmap* is an image of C channels, each channel has a width W and a height H. For fully connected layers, the *ifmaps* are flattened into $1D$ arrays of input neurons (N_{in}).

The outputs of Gemini are the ***ofmaps***: Gemini produces a batch of k *ofmaps* (that are the outputs of the last layer of the NN). For convolutional layers (as well as maxpool and depthwise ones), each *ofmap* is an image of M channels, each channel is an $2D$ image that has a width Q and a height P. For fully connected layers, each ofmap is flattened into an $1D$ array of output neurons (N_{out}).

This chapter outlines the design and implementation challenges encountered in the Gemini project. We start in Section 4.1 by discussing the data quantization used in the Gemini IP, then Section 4.2 provides an in-depth examination of the Gemini accelerator's architecture, starting with an exploration of the structural parameters defining its design. Subsequently, we present the NPU architecture, describing its constituent blocks. Further elaboration is offered on the RAM organization, concluding with a broad overview of layer scheduling and its interplay with the NPU architecture and RAM organization. It is important to note that certain aspects, such as layer scheduling, remain not detailed, as they were predetermined by the project requirements and fell outside the scope of my research. Section 4.3 explains the implementation challenges of Gemini. It starts by explaining how High-Level Synthesis (HLS) was useful for the Gemini implementation, then we explain the different challenges encountered

to implement the NPU for different hierarchical levels using this method, additionally an explanation of the RTL wrapper allowing the use of real RAM is given. Finally, in Section 4.4, we highlight the project's achievements, detailing the tape-outs conducted and comparing Gemini with other benchmark accelerators.

4.1 Gemini Data Representation and Quantization

The decision was made to quantize the data within Gemini in order to optimize power and area, even though it involves a slight sacrifice in accuracy, as described in Subsection 3.4. For the quantization method, we were required to use TensorFlow quantization [64] as part of the project.This method involves converting initial floating-point data into integers within the range of 0 to 255. This quantization process is uniform and follows the equation: $q = (f - z)s$, where q represents the quantized integer value, f is the initial floating-point value, z is the zero point, and s is the scaling factor.

To ensure that the quantized values fall within the $[0, 255]$ range, we use a multiplication by the scaling factor s. The scaling factor is calculated as the inverse of the difference between the maximum and minimum float values. The zero point corresponds to the quantized value of zero in the floating-point domain. Since all quantized integers are positive, it is necessary to determine the corresponding value of zero in the quantized domain.

So, the inputs, specifically *fmaps* and *weights*, processed by the accelerators are quantized. However, following all the required multiplications and accumulations for layer computations, the *ofmaps* pixels surpass the quantization range. To resolve this issue, we perform a rescaling of the output by multiplying it with the scaling factor. This rescaling process requires the presence of a dedicated hardware block within PEs, as elaborated in Subsection 4.2.2. It is important to highlight that multiplying the *ofmaps* pixels by the known scaling factor before processing opens up the possibility of efficient hardware implementation. For instance, we opt to represent the scaling factor as a multiple of a power of two: $s = k \times 2^{-scalingbits}$, where k is an integer, and $scalingbits$ is the number of bits for the scaling factor. This approach enables us to implement multiplication by a power of two at a significantly lower cost than a general multiplication in the hardware. It is done using shifts.

The customer required the use of 8 bits for both *fmaps* and *weights* to achieve great performance while maintaining accuracy suitable for their application.

4.2 Gemini Configurable Architecture

In this chapter, we begin by introducing in Subsection 4.2.1 the two structural parameters configuring Gemini. Following that, we describe the accelerator's architecture in Subsection 4.2.2, starting with an overview of the processing elements and progressing to the entire accelerator blocks. Additionally, we provide details in Subsection 4.2.3 on the RAM organization and explain the scheduling process for convolution-like layers and fully connected layers in Subsection 4.2.5.

4.2.1 Presentation of Gemini Structural Parameters

Gemini is composed of a Neural Processing Unit (NPU) and two single port SRAM modules: the features maps (*fmaps*) RAM and the *weights* RAM. The NPU contains the block in charge of calculations, called the processing elements (PEs) array. There are N parallel PEs organized in $2D$ $(WPAR, MPAR)$ with $N = WPAR \times MPAR$. These two structural parameters are configurable before the logic synthesis. They size all the designs from the PEs array to top-level RAMs, and they fix the scheduling of the operations. They were introduced to optimize the convolutions processing. $WPAR$ stands for width parallelization of the output feature map (*ofmap*) and $MPAR$ is the filter parallelization (since the number of filters is usually called M in literature [25]). Fig.4.1 illustrates the notations with $(WPAR, MPAR) = (2, 4)$.

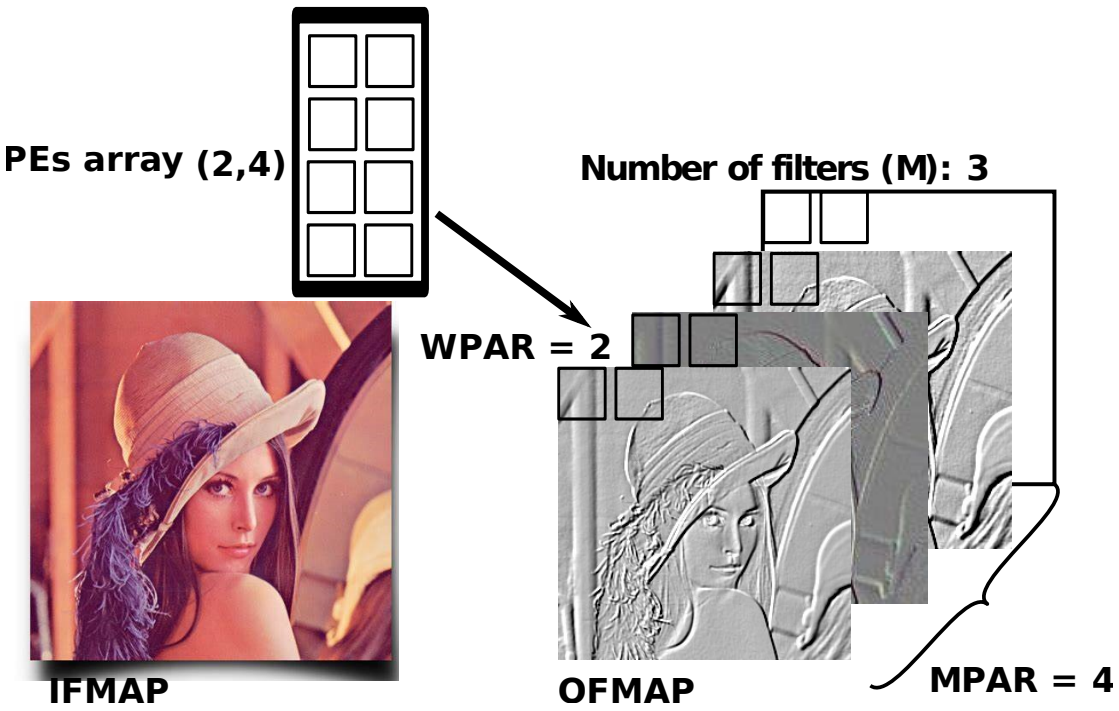

Fig. 4.1 *PEs array organization for* $(WPAR, MPAR) = (2, 4)$

4.2.2 NPU Architecture

The NPU architecture is composed of the PEs array, the input mixers (*ifmaps* mixer and *weights* mixer) and the storing stage. Fig.4.2 shows the NPU hardware blocks in the top-level architecture.

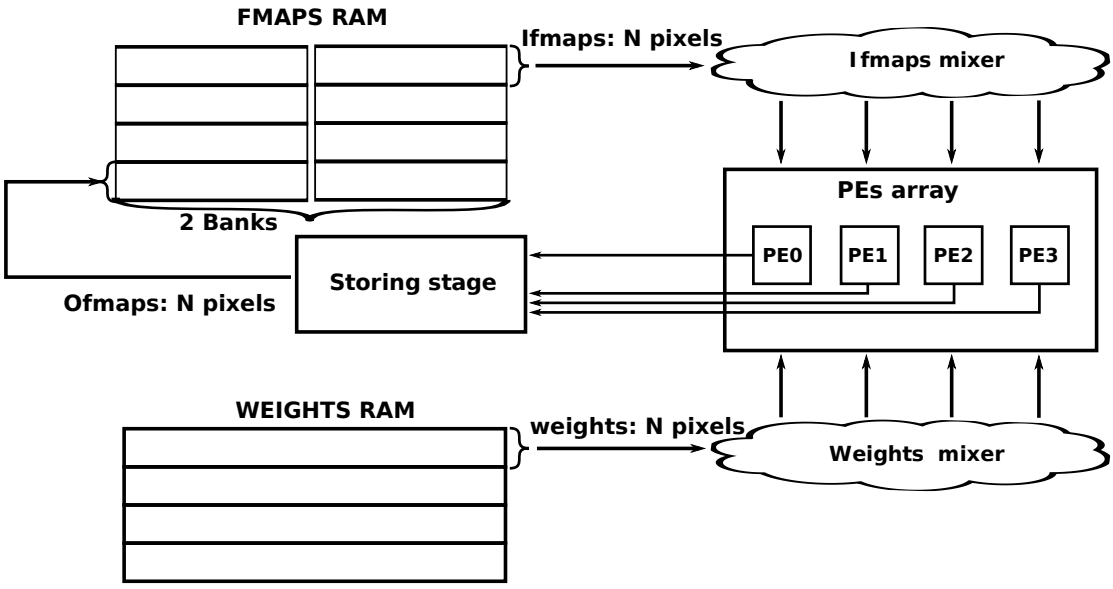

Fig. 4.2 *NPU hardware blocks*

- **The processing elements array** is composed of N PEs. PEs are organized following an output stationary data-flow, each processor then computes an *ofmap* pixel. The partial sums contributing to the output pixels are stored in the accumulator (register) of each PE when new *weights* and *fmaps* pixels are broadcasted to the PEs every cycle. This paradigm is described in [119] and used by several accelerators, such as [40] or [93].

  Output calculations are optimized for parallel processing with N PEs. In the case of fully connected layers, these N PEs simultaneously compute N output pixels from the total of N_{out} output neurons. For convolutional layers, the N PEs are organized in a two-dimensional array. They calculate $WPAR$ pixels from the $2D$ *ofmap* channel ($P \times Q$) simultaneously. This simultaneous computation is applied to $MPAR$ *ofmaps* from the total of M $2D$ *ofmap* channels that need to be processed. Consequently, this configuration allows for the concurrent calculation of N *ofmap* pixels during convolutions. For instance, in Figure 4.1, 8 pixels are computed concurrently: 2 pixels per filter ($WPAR$), and this is done for 4 filters ($MPAR$).

  The decision to parallelize the PE array in a $2D$ fashion (rather than $1D$) is driven by two key factors. Firstly, it is a practical choice. NNs often process *fmaps* that start large with a low channels number at the first layers and then progres-

sively shrink while the number of channels increases. Employing two levels of parallelization ensures that PEs are utilized effectively across a wider range of scenarios. Without this, some PEs would remain unused when dealing with small *fmaps*. Secondly, several operators do not scale directly with $MPAR$. Instead, they scale with $WPAR$ and are shared among the $MPAR$ processing units. This save area and power.

In Figure 4.1, we illustrate an example where 2 pixels are computed simultaneously for 4 $2D$ *ofmaps*. It is important to note that in this example, as $MPAR$ exceeds M, the last two $WPAR$ processing units are not utilized.

Regarding the architecture of a single processing element (PE), each PE consists of two pipelined stages, as illustrated in Figure 4.3:

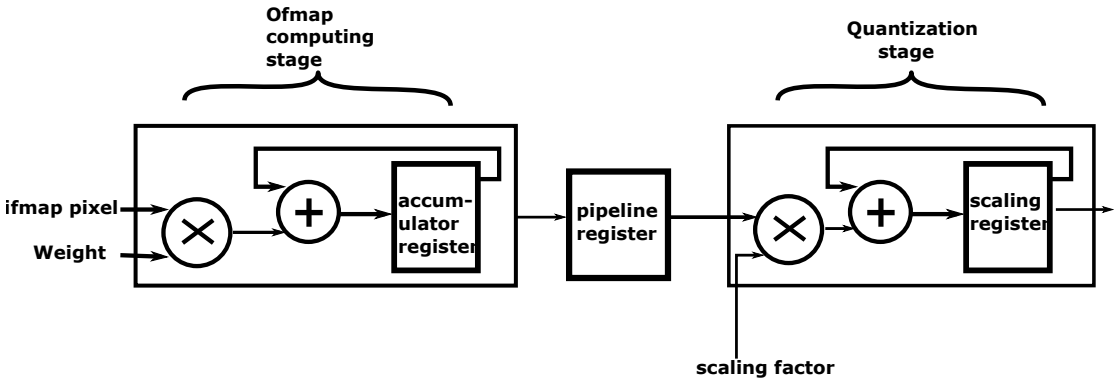

Fig. 4.3 *Architecture of one Processing Element*

- The first stage, located on the left side of Figure 4.3, encompasses the logic responsible for performing the primary operation for layer computations. This involves MACs (Multiply-ACcumulate) between *fmaps* pixels and *weights* for convolutions and fully connected layers, as well as the maximum operation for maxpools. The accumulator register accumulates partial sums (or maximums for maxpool) across the required cycles.
- The second one is the quantization stage. It is triggered when *ofmaps* pixels are calculated (such as after completing all MAC operations in a convolution layer). Its purpose is to put the *ofmaps* pixels in the desired range of quantization by multiplying them by a scaling factor and taking only *fmapbits* bits from the most significant bits. Further details were given in Subsection 3.4. The quantization stage always takes 5 clock cycles and necessitates a scaling register to preserve the output throughout these 5 calculation cycles. This block also performs the application of the ReLu function following the scaling.

A pipeline register is positioned between the two stages to enable communication and simultaneous operation in a pipeline fashion.

- **Input mixers**: in general, mixers are combinational blocks that receive disordered data as input and produce sorted data as output. It is crucial for the PEs array to perform regular operations in every cycle, while the complexity of data management is handled by the mixers. These mixers ensure that in each cycle, every PE receives the correct *fmap* pixel and the appropriate *weight* for its operation. Typically, they are implemented as shifters mapped to multiplexers (in RTL). As observed in Figure 4.2, inside the NPU, these mixers are positioned at the interface with the RAMs, responsible for transferring data to the PE array. There are two input mixers within the NPU:
  - The *ifmaps* mixer is connected to the read buffer of the FMAPS RAM. Its function is to sort the N pixels being read to ensure that, during each cycle, the N processing elements (PEs) in the array receive the correct *fmap* pixel for computation.
  - The *weights* mixer is connected to the read buffer of the WEIGHTS RAM. It functions similarly to the *ifmaps* mixer, sorting weights and *biases*. During each cycle, it delivers the necessary *weights* and *biases* to the PEs array based on the layer being computed.

  The details of the mixer functions are not given, as they were imposed and not studied during my thesis.
- Finally, **the storing stage** is located at the output of PEs array. It eliminates some useless PEs computations that should not be written in the *fmaps* RAM. For convolutions, the PEs array calculates *ofmaps* pixels corresponding to horizontal *padding* and *strides* even if they are not necessary for the *ofmap* (they are eliminated by the storing stage). A design choice allowing a few useless operations done by PEs was made to simplify the mixers and to optimize the power and area.

  The storing stage also writes the quantized outputs into the FMAPS RAM in the correct order upon receiving a signal, preventing conflicts with reading during the same cycle. This function is mapped to a mixer. This stage is pipelined with the PEs.

4.2.3 RAMs Organization

In this paragraph, we provide a detailed explanation of how the RAMs are organized within Gemini. In fact, data organization within the RAMs must follow a specific format to enable Gemini's processing. This data reorganization is performed prior to Gemini's execution and is managed by an external element. For the *fmaps*, it can be executed directly by the sensor itself (for example, if an FPGA is provided with the sensor), or

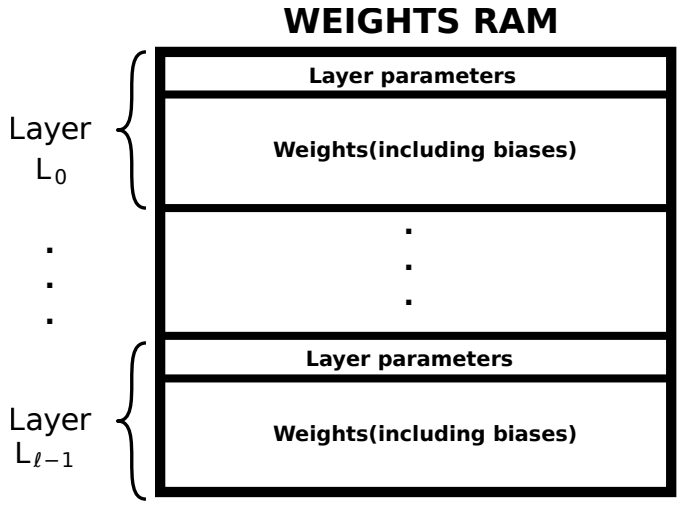

Fig. 4.4 WEIGHTS *RAM organization for a ℓ layers NN* $\{L_0, L_1, L_{\ell-1}\}$

we can employ a microcontroller to handle this operation before initiating the NN execution on Gemini.

Furthermore, it is important to note that Gemini's capabilities are limited to NNs that can entirely fit within its RAM. Consequently, at the start of the execution, all the data required for processing must already be present in the RAM. There is no gradual filling of RAM through mechanisms like DMA (Direct Memory Access) [2].
Gemini utilizes RAMs developed by STMicroelectronics, specifically of the SPREGHD type, operating in CMOSC40. These RAMs are single-port, meaning reads and writes cannot occur simultaneously within the same cycle.
The organization of WEIGHTS and FMAPS are explained in the following paragraphs:

4.2.3.1 WEIGHTS RAM Organization

The WEIGHTS RAM stores the NN's layer *weights* (including *biases*) and layer parameters in a compact format. Figure 4.4 illustrates the RAM's organization for a ℓ-layer NN. The data for each layer is arranged sequentially in the RAM.
For each layer, the first lines are dedicated to the layer parameters. These parameters are specific to the layer. For convolutional layers, these parameters include M, W, H, P, Q, R, and S, and for fully connected layers, they include N_{in} and N_{out}. Additionally, these parameters may encompass constants used during NPU execution, which are stored in memory for reading instead of allocating specific hardware within the NPU to calculate them. It is done for costly operations as constants obtained by divisions.
After storing the layer parameters, the RAM holds the layer *weights*. For convolutional layers, this corresponds to the $M \times S \times R \times C$ filter *weights* along with M *biases*. In the case of fully connected layers, we have $N_{in} \times N_{out}$ synapses in addition to N_{out} *biases*.

This process continues for each subsequent layer until all layer parameters and *weights* are placed.

A WEIGHTS RAM word consists of $N \times$ weightbits bits. The number of words required depends on the size of the NN and should be sufficient to accommodate all the parameters and *weights* of the network.

Concerning the *weights* placement inside the memory, it is done in a judicious way to facilitate the NPU execution and data reuse. The approach differs between convolutional and fully connected layers.

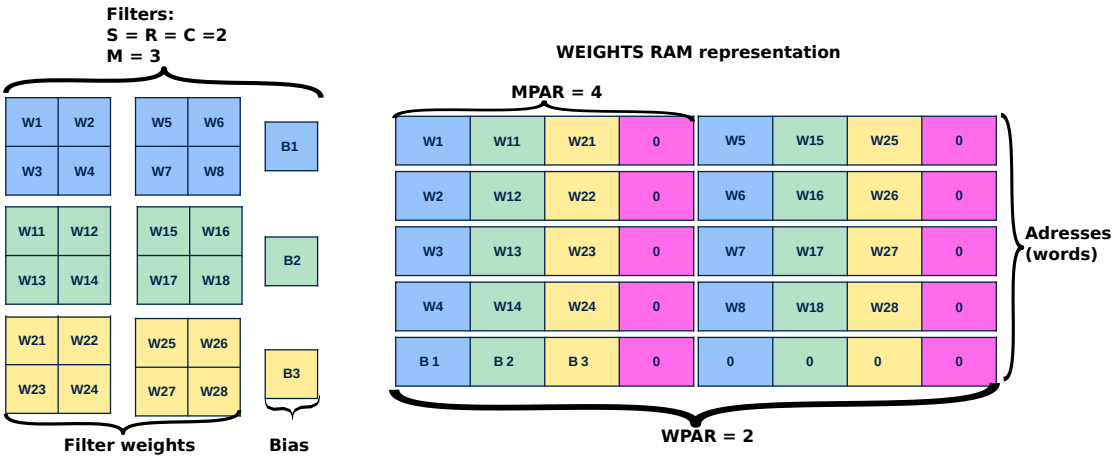

Fig. 4.5 *Example of convolution* weights *placement in the RAM*

For **convolutional layers**, filter *weights* are organized as follows: they are grouped into sets of $MPAR$ pixels and interleaved. This means that the *weights* of the first $MPAR$ filters are placed in an interleaved manner before moving on to the remaining filter *weights*. If $MPAR$ is greater than M, the set is completed with zeros. Additionally, each RAM line contains $WPAR$ pixels from each of the $MPAR$ sets of filter *weights*. In Figure 4.5, we illustrate an example with a hardware configuration of $(2, 4)$ and $M = 3$ filters of size $S \times R \times C$ respectively equal to $2 \times 2 \times 2$ *weights*. Each RAM line in the figure contains $WPAR = 2$ pixels from each of the $MPAR$ filters, and the *weights* of the three filters are interleaved. The last pixel in each $MPAR$ set is set to zero because $MPAR > M$.

Regarding the pixel order placement within a filter, we begin by organizing the pixels of each channel, followed by columns, rows, and finally, the *bias* starting from the top-left filter *weight*. In the example from Figure 4.5, for the blue filter, the order is as follows: $W_1, W_5, W_2, W_6, W_3, W_7, W_4, W_8, B_1$. This arrangement is chosen to facilitate the simultaneous processing of $MPAR$ filters.

Depthwise layers are adopting the same memory placements for *weights*.

For **fully connected layers**, the *weights* or synapses are organized into groups of $N = WPAR \times MPAR$ sets, corresponding to each RAM line. This means that we place all N values for N_{out} before moving on to the next set. If N exceeds N_{out}, the set

is filled with zeros. In Figure 4.6, we provide an example for a hardware configuration of $(2, 4)$, resulting in $N = 8$, with $N_{out} = 2$. Here, only two pixels from the N set (corresponding to a RAM line) have non-zero values.
Regarding the placement order, we arrange the *weights* for each output neuron row by row, starting from the top of the matrix. In Figure 4.6, for the first N_{out}, the order is $W_{00}, W_{01}, W_{02}, B_0$. This arrangement is chosen to efficiently process N output neurons simultaneously.

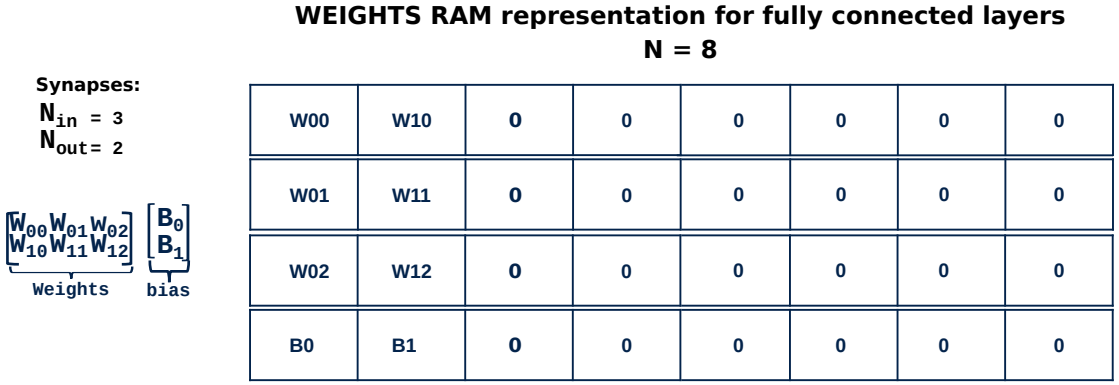

Fig. 4.6 *Example of fully connected* weights *placement in the RAM*

4.2.4 FMAPS RAM Organization

The FMAPS RAM contains the *fmaps* pixels: before the beginning of the executions, this RAM contains *ifmaps* pixels; during the executions, intermediary *fmaps* are also stored inside this RAM overwriting non-meaningful data, and finally, at the end of the execution, it contains the final *ofmaps*.

The FMAPS RAM, in reality, consists of $WPAR$ memory banks, that are $WPAR$ separate RAMs. One FMAPS RAM corresponds then to the concatenation of one word of each $WPAR$ RAM word. Each bank's word consists of $MPAR \times$ fmapbits bits. The number of words is determined to ensure that the largest *fmap* in the NN fits in the RAM without overwriting essential pixels from the preceding *fmap*, which are necessary to complete its processing.

Similar to the *weights*, the placement of *fmaps* is optimized for parallel execution on $WPAR$ and $MPAR$. The *fmap* pixels are grouped into sets of interleaved $MPAR$ channels. This means that $MPAR$ channels from the total C channels are placed before the other sets. If C is less than $MPAR$, the set is completed with zeros. Regarding the order of $2D$ pixels (within the same channel), we start by arranging them in columns and then in rows, beginning with the top-left pixel.
In Figure 4.7, we provide an example for a hardware configuration of $(2, 4)$ with a *fmap* size of $W \times H \times C$, where $W \times H \times C$ is equal to $2 \times 4 \times 2$ pixels. In this illustration, each of the $WPAR = 2$ banks holds $MPAR = 4$ pixels. The last two pixels in each

bank are set to 0 because $C = 2 < 4$. The arrangement shows that for each channel, pixels are stored first by columns and then by rows. It is required that the entire NN fits

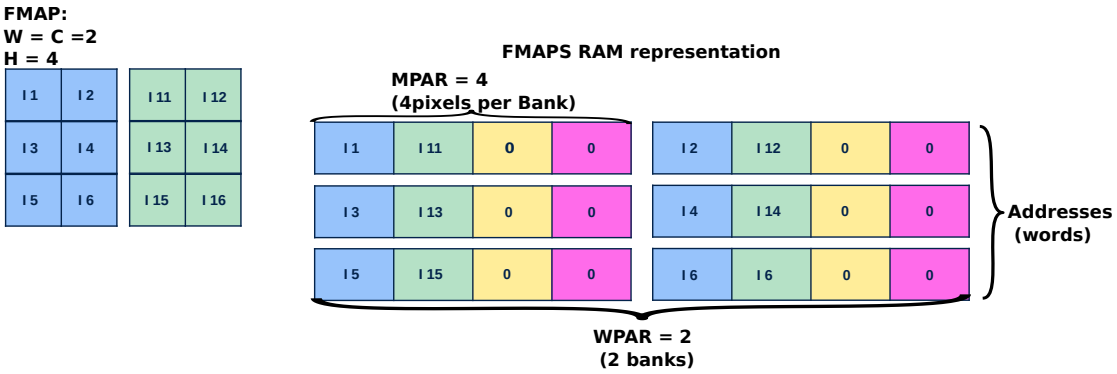

Fig. 4.7 *Example of fmaps pixels placement in the RAM*

into the on-chip RAMs, including *weights* and intermediate *ifmaps*. Note that the pixels of the ifmaps are shuffled in such a way to be processed efficiently by the NPU.

In this section, we introduced the organizational principles of Gemini RAMs. In certain specific cases, additional complex optimizations were implemented to reduce the RAMs' footprint, particularly by eliminating unnecessary zeros. However, I was not directly involved in these optimizations.

4.2.5 Layers Execution Scheduling

This subsection explains at a high-level abstraction the sequence of operations required for Gemini to produce an output.

The NPU processes the NN layer by layer. It begins by reading the parameters from the WEIGHTS RAM for the current layer. After this, multiple execution cycles are carried out depending on the specific layer's requirements. Finally, there is a transition cycle to prepare for processing the next layer, which also starts with parameters reading. The execution cycles follow distinct scheduling patterns for convolutions and fully connected layers. This scheduling is facilitated by the RAM organization introduced in Subsection 4.2.3:

For **convolution-like layers**, as discussed in Subsection 4.2.2, $MPAR$ $2D$ *ofmap* channels out of a total of M are processed simultaneously. Within each channel, pixel calculations are organized into sets of $WPAR$. This allows for simultaneous processing of a total of N *ofmap* pixels. Once all the channel pixels ($P \times Q$) are computed (requiring $\lceil \frac{P \times Q}{WPAR} \rceil$ sets of $WPAR$), $MPAR$ *ofmap* channels are ready for quantization. The NPU then process the following $MPAR$ channels until all M *ofmap* channels are generated.

To compute the N *ofmap* pixels by the ofmap computing stage of the PE array (Figure

4.3), a word is fetched from the WEIGHTS RAM read buffer during each cycle. Subsequently, the *weights* mixer selects one *weight* for each $MPAR$ filter and broadcasts it to the PEs, where $WPAR$ PEs receive the same *weight*. Within the PEs' computation stage, these *weights* are multiplied with N *fmaps* chosen by the *ifmap* mixer, following the fetch of one word from the FMAPS RAM read buffer (with one word per bank). The results are accumulated within the PE register. Within a single filter, the execution order follows that of the RAMs (refer to Subsection 4.2.3): columns take precedence over rows, starting from the top-left edge of the filter.
It is important to highlight that this scheduling optimize the data reuse. The word fetched from the RAMs serves multiple cycles and does not necessitate more than one read. For example, a word line from the WEIGHTS RAM is employed for $WPAR$ cycles, since the mixer selects only one *weight* per cycle. The same principle applies to the *fmaps*, where the same word is employed for different cycles, with the mixer being the sole component changing the data fed to the PEs.
Once all filter MAC operations are completed, the N *ofmap* pixels undergo quantization and ReLu application (Figure 2.11 in Subsection 2.2.4). All these steps operated by the PEs are the same for all *padding* and *stride* values (see Subsection 2.2.3), the non-used pixels are eliminated in the following step.
After quantization, the *ofmap* pixels undergo processing in the storage stage, which filters out non-useful data (resulting from *padding* and *strides*) and then writes them into the FMAPS RAM. As a reminder, the computation stage, quantization stage, and storing stage are pipelined and operate on different pixels concurrently.
Depthwise layers follow the same scheduling as convolutions, while pooling layers follow a similar scheduling pattern, with the *weight* multiplication being replaced by the pooling operation.

For **fully connected layers**, N among the N_{out} output neurons are processed concurrently. During each cycle, one word is fetched from the FMAPS RAM and then one input neuron from the set of N_{in} is broadcasted to the PEs by the *ifmap* mixer. It is then multiplied with N *weights* selected by the *weights* mixer (after accessing the read buffer of the WEIGHTS RAM) and accumulated within the PE accumulator each cycle. Subsequently, similar to convolution, the N output pixels are quantized and directed to the storing stage, which writes them into the FMAPS RAM.
Once again, this scheduling facilitates data reuse, particularly for the FMAPS RAM, where the word read is utilized over N cycles, with only the mixer changing the single input neuron provided to the PE array.

Finally, it is important to highlight that even though data fetching and writing into *FMAPS* occur during the same cycles, it does not lead to conflicts when using single-port RAM. As mentioned in Subsection 4.2.2, the storing stage can only write when it

receives a signal, indicating the absence of a read during the same cycle. This absence of read operation is common because, as explained below, there are several cycles where data is fetched from the RAM's read buffer without resulting in an actual read, as the data was already present in the buffer following a previous read in preceding cycles. It is only reused.

4.3 Implementation

Some aspects of the Gemini architecture were determined during my thesis, but my primary focus for the past two years, along with the other PhD students, has been the implementation and addressing all the challenges it presents.

Actually, once the architectural design is finalized, the implementation becomes a critical task, entailing various challenges. Figure 4.8 provides an overview of the entire process. The initial design phase involves creating a synthesizable $C++$ code representation of the NPU architecture. At this step, RAMs are mapped to naive models. During this phase, extensive simulations are conducted to validate the correctness of the outputs. To achieve this, we compare the results with those obtained from the *Tensorflow* library. Subsequently, this code is utilized by a High-Level Synthesis tool *SIEMENS CATAPULT* to generate RTL code for the NPU. This RTL code is then integrated with real (SPREGHD models) RAM instantiation to produce the RTL code for the Gemini accelerator. At this stage, RTL simulations are performed using *CADENCE XCELIUM* to verify the correctness of the generated RTL. Next, the RTL undergoes synthesis using the logic synthesis tool *SYNOPSYS DCSHELL*, and the gate netlist is simulated using *CADENCE XCELIUM*. The final circuit is obtained after placement and routing, carried out by *CADENCE INNOVUS*, resulting in the final GDS layout. Back-annotated simulations, taking into account real cell and interconnect delays, are then conducted to verify the proper functioning of Gemini.

Subsection 4.3.1 introduces the HLS and its importance in Gemini design, the Subsection details the NPU's implementation using HLS across various abstraction levels, spanning from the NPU itself to the abstract blocks and within the PEs array. Section 4.3.3 then outlines the integration of the NPU into the Gemini top-level structure to generate the ultimate Gemini RTL description.

4.3.1 HLS overview

HLS is a commonly employed method to generate a synthetizable RTL design [87, 32, 105] (in a hardware description language such as $VHDL$ or $Verilog$) from a high-level programming language such as $C/C++$ or $SystemC$. The fundamental objective of HLS is to automate the conversion of algorithmic abstractions into hardware concrete

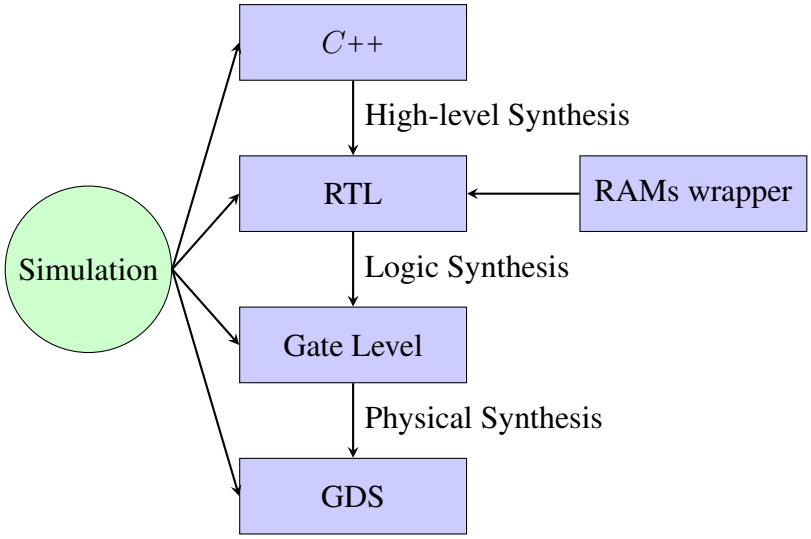

Fig. 4.8 *Gemini Design steps*

configurations, thereby streamlining the design process and reducing time-to-market for complex digital systems. By using HLS, designers can explore multiple architectural alternatives, optimize for various performance metrics (e.g., power consumption, throughput, area utilization), and explore trade-offs between them. Lahti et al. [72] in Figure 4.9 illustrates a meta-analysis on 46 different papers comparing HLS and classical RTL design. It shows that HLS enables a considerable time saving for each step of the chip making process. It also reduces the necessary design expertise, as various steps are automated and managed by the HLS tool. For instance, with *SIEMENS CATAPULT*, loop pipelining or unrolling [58] can be effortlessly accomplished using commands. HLS is also well-suited for configurable designs due to its high-level language, which simplifies automation and is technology-agnostic, making it easier for re-targeting. Additionally, HLS provides useful tools for easy debugging; for instance, it allows for RTL simulations directly from test-benches written for the high-level description. Furthermore, HLS provides all the estimations typically available when RTL is generated, including timing, area, and power analysis. Finally, the choice of using $C/C++$ for the design offers significant advantages in terms of readability. It enables object-oriented coding practices that abstract complexity and improve comprehensibility, for instance. Moreover, it greatly facilitates code maintenance, allowing for the easy addition of new features without extensive modifications to existing code. Furthermore, C/C++ is compatible with documentation tools like Doxygen, which simplifies the process of creating documentation within the code, as explained in [99].

The primary stages involved in generating RTL from a high-level language like $C/C++$ (or any other language) include:

- Code Compilation: this initial phase involves compiling the code.
- Allocation: in this step, libraries containing primitive components are chosen and

mapped to operations. This stage also defines the design hierarchy and incorporates features such as loop unrolling or pipelining.

- Scheduling: the scheduling phase determines when each operation is executed within the cycle. It plays a critical role in handling dependencies among operations.
- Binding: the binding step is responsible for optimizing resource sharing and making necessary adjustments.

These stages collectively lead to the creation of RTL designs from high-level descriptions. Nevertheless, while HLS offers numerous advantages, it involves giving up some control over the design. Throughout various stages, such as allocation and scheduling, the tool makes several decisions. Therefore, to generate RTL with the desired architecture, it requires writing the code with careful consideration and taking into account the tool's decision-making process.

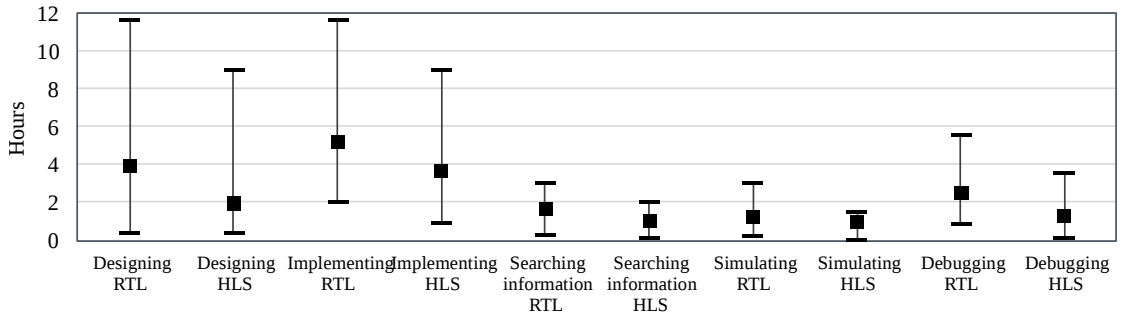

Fig. 4.9 *Maximum, minimum, and average time usage for different categories with RTL and HLS. From [72]*

4.3.2 NPU Design in HLS

In the context of designing Gemini, HLS proved to be exceptionally beneficial. Given that the Gemini IP was initially intended to be configurable, HLS proved invaluable for transitioning between different $(WPAR, MPAR)$ configurations. It also played a crucial role in the validation process, as Gemini needed to be delivered to a client, requiring exhaustive verification involving random NNs across various configurations. They were efficiently performed through C++ simulations, making debugging quick and straightforward. Achieving the same results using RTL simulations would have been virtually impossible due to the significant time constraints.

In this subsection, we will describe the challenges encountered while describing Gemini in HLS, with the help of an illustration using a very simplified version of the C++ code to be synthesized. To aid comprehension, Figure 4.10 illustrates the different data types used for feature maps (*fmaps*). Please note that the same applies to *weights*.

In $C++$, *IFMAPS_RAM* is modeled through the type *NPU_memory_type*, representing the *fmap* RAM, including all its banks (as shown in Listing 4.1). The IFMAPS variable, which is of type *N_fmaps_type* (as seen in Listing 4.2), corresponds to one line within *IFMAPS_RAM*. This data is handled by the PEs array and contains a total of N *fmap* pixels (in this example, N is equal to 8). Finally, *SINGLE_IFMAP* (presented in Listing 4.3) is of type *fmap_type*, representing a single pixel with $fmapbits$ bits (here equal to 8). This data is used by an individual PE.

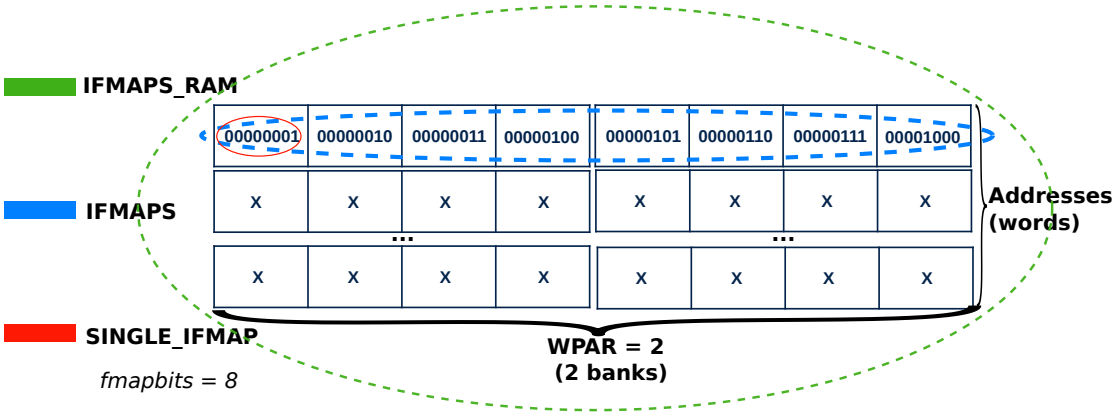

Fig. 4.10 *Different* fmap *types used in Gemini architecture*

4.3.2.1 NPU Top Level Function in HLS

Concerning Gemini design, the top level contains the NPU and RAMs. The HLS is targeting the NPU. The RAMs are set as external. It means that they are considered as external blocks and are not handled by the HLS. This is operated thanks to a *CATAPULT* command. The RAMs will be instantiated when the NPU RTL is generated (Subsection 4.3.3). This choice is made because, in HLS, the $C++$ arrays are mapped to a standard RAM model (with only D, Q, clk, reset pins) that does not correspond well to the SPREGHD RAMs that we are using. As is it was mentioned in Subsection 4.2.5, there is no simultaneous read and write from the FMAP RAM during the execution. We made the choice to design Gemini with two separate RAMs for *fmaps*: an IFMAP RAM for reading *fmaps* and an OFMAP RAM for writing them. Opting for a single RAM in the design would have introduced potential issues when attempting simultaneous reads and writes during the same cycle or when scheduling them in different cycles to prevent conflicts. It is important to note that in practice, such conflicts would never occur, as the scheduling prevents their occurrence (Subsection 4.2.5) and in case of conflict they are resolved outside the HLS process (Subsection 4.3.3). In Listing 4.1, we can observe the top-level function that outlines the design of the NPU. In this context, three RAM variables are employed: IFMAPS_RAM, OFMAPS_RAM, and WEIGHTS_RAM. However, at the RTL level, where actual RAM instances (SPREGHD) are instantiated,

the same physical RAM is utilized for both IFMAPS_RAM and OFMAPS_RAM.

Listing 4.1: *NPU function in C++*

```
void execute_NPU(NPU_memory_type& IFMAPS_RAM,
                 NPU_memory_type& OFMAPS_RAM,
                 NPU_memory_type& WEIGHTS_RAM,
                 bool START) {
    NPU_state_type NPU_STATE = INITIALIZATION;
    bool END_MODEL = FALSE;
    #pragma hls_pipeline_init_interval 1
    WHILE (1) {
        SWITCH (NPU_STATE) {
            CASE INITIALIZATION:
                NPU_STATE=initialize_NPU(START);
                BREAK;
            CASE READ_LAYER_PARAMETERS:
                NPU_STATE=read_layer_parameters(
                   WEIGHTS_RAM);
                BREAK;
            CASE EXECUTE_LAYER:
                NPU_STATE=execute_layer(IFMAPS_RAM,
                   OFMAPS_RAM,WEIGHTS_RAM);
                BREAK;
            CASE END_LAYER:
                NPU_STATE=prepare_next_layer();
                BREAK;
            CASE END_EXECUTION:
                END_MODEL=TRUE;
                BREAK;
        }
        if (END_MODEL) {
            BREAK;
        }
    }
}
```

As observed in Listing 4.1, the NPU function is described thanks to a *while(1)* loop, it helps to ensure that each loop will be executed during one clock cycle. The loop is pipelined with an initiation interval of 1 (*i.e.* the pipeline input is updated every cycle) thanks to the *CATAPULT* library line *#pragma hls_pipeline_init_interval 1*. Within the

loop, there exists a finite state machine that describes the execution of the NN. This state machine operates as follows:

- *INITIALIZATION:* in this state, all NPU variables are initialized.
- *READ_LAYER_PARAMETERS:* this state corresponds to the phase where layer parameters are read before commencing layer execution (as stated in Subsection 4.2.5)
- *EXECUTE_LAYER:* this is the primary state where the actual layers are computed.
- *END_LAYER:* as a transitional state, it serves to update variables in preparation for launching a new layer computation by transitioning to the *READ_LAYER_PARAMETERS* state.
- *END_EXECUTION:* this final state marks the conclusion of NPU execution.

4.3.2.2 NPU Blocks in HLS

During the *EXECUTE_LAYER* phase, the function responsible for executing the layer on Gemini is called. In Listing 4.2, the function *execute_layer* is detailed. Within this function, all NPU block functions (refer to Subsection 4.2.2) are called sequentially. First, we encounter the *ifmap_mixer* and *weight_mixer* functions, which respectively read a line and perform mixing operations from the *IFMAPS_RAM* and *WEIGHTS_RAM*. Their purpose is to ensure that the PE array has the correct data for the layer computation. The behavior of these mixers depends on the type of layer being computed. Following the mixer operations, the PE array performs computations using the data provided by the mixers (IFMAPS and WEIGHTS), and it generates an output (*ACCU_OUT*) that is stored within the *OFMAPS_RAM*.

Listing 4.2: *Layers execution function in C++*

```
void execute_layer(NPU_memory_type& IFMAPS_RAM,
                   NPU_memory_type& OFMAPS_RAM,
                   NPU_memory_type& WEIGHTS_RAM) {
    PEs_command_type CMD = layers_control(); gives
       address
    N_fmaps_type IFMAPS = ifmaps_mixer(IFMAPS_RAM);
    N_weights_type WEIGHTS = weights_mixer(WEIGHTS_RAM);
    N_accu_type    ACCU_OUT = 0;
    ACCU_OUT=PEs_execution(IFMAPS,WEIGHTS,CMD);
    storing_stage(OFMAPS_RAM,ACCU_OUT); update the
       boolean LAST_EXECUTION
    if (LAST_EXECUTION == True){
```

```
        NPU_STATE = EXECUTE_LAYER;}
    else{
        NPU_STATE = EXECUTE_LAYER;}
}
```

4.3.2.3 Processing Elements in HLS

As described in Subsection 4.2.2, the PEs array is responsible for executing the operations required for layer computation. It comprises N PEs, each handling an output pixel calculation. Listing 4.3 illustrates a simplified version of the PEs array function. To begin with, the *CATAPULT* command *#pragma_hls_design ccore* indicates that this function will be synthesized as a *CCORE* (Catapult C Optimized Reusable Entity [58]). This designation means that it is a reusable block synthesized and optimized independently of the rest of the design. This decision was made to enhance hierarchy control and reduce synthesis time. On the other hand, the function involves a loop iterating over the N PEs, where each operation is performed individually for every PE. For example, in the case of the *MULTIPLICATION* command (*CMD*), each of the N PEs within the array computes the multiplication between a $weighbits$ *weight* and $fmapbits$ *fmap*. It is crucial to note that the *CATAPULT* command *#pragma hls_unroll yes* is applied above the for loop. This signifies that the loop is unrolled, and the N loop iterations are mapped to N operators that execute the tasks within the loop simultaneously. To achieve this, it is essential to ensure that the tasks inside the loop are independent.

This function operates also the output scaling and the activation seen in Subsection 4.2.2

Finally, we encountered numerous challenges during the optimization of the code with the objectives of reducing chip area and improving latency. The coding style significantly influenced the performance of the synthesized NPU. The main difficulties were about the need to rewrite the code to minimize data dependencies and promptly utilize available data. This strategic adjustment resulted in a notable reduction in the critical path and the removal of unnecessary registers storing data. Other optimizations included the reduction of multiplier sizes by employing smaller ones and the execution of multiplications over multiple cycles (in pipeline).

Listing 4.3: *PEs array function in C++*

```
#pragma hls_design ccore
N_accu_type PEs_execution(N_fmaps_type& IFMAPS,
                          N_weights_type& WEIGHTS,
```

```
                                PEs_command_type CMD) {
    static N_accumulator_type ACCU_IN = 0;
    #pragma hls_unroll yes
    for (int p=0, p<N, p++){
        fmap_type   SINGLE_IFMAP   = IFMAPS.slc<fmapbits>
            (p*fmapbits);
        weight_type SINGLE_WEIGHT = WEIGHTS.slc<
           weightbits> (p*weightbits);
        accumulator_type SINGLE_ACCU = ACCU_IN.slc<
           accubits> (p*accubits);
        SWITCH (CMD){
            CASE MULTIPLICATION:
                SINGLE_ACCU = SINGLE_WEIGHT *
                   SINGLE_IFMAP;
            CASE MAC:
                SINGLE_ACCU = SINGLE_ACCU + SINGLE_WEIGHT
                    * SINGLE_IFMAP;
            CASE MAX:
                if (SINGLE_ACCU < SINGLE_IFMAP){
                    SINGLE_ACCU = SINGLE_IFMAP;
                };

        ...
        }
        ACCU_IN.set_slc(p*accubits,SINGLE_ACCU)
        ACCU_IN= output_scaling_stage(ACCU_IN,
           scaling_factor,offset,relu);
        ...
    return ACCU_IN;
}
```

4.3.3 RTL Wrapper

As described in Subsection 4.2.1, the top level of the Gemini architecture comprises the NPU, along with *WEIGHTS_RAM*, *IFMAPS_RAM*, and *OFMAPS_RAM*. The RAM models generated by HLS for array mapping do not match the actual RAMs used in practice. Additionally, the Gemini top level produced by HLS includes two separate RAMs for the *fmaps*, whereas in the real design outlined in Subsection 4.2.3, there is only one RAM. This disparity led us to wrap the Gemini RTL generated by *CATAPULT*

within a top-level structure, using SPREGHD RAMs for *WEIGHTS_RAM* to replace the basic array model provided by HLS. We also introduced a logic block that combines the signals shared by the NPU with *IFMAPS_RAM* and *OFMAPS_RAM*, allowing them to communicate exclusively with a single *FMAPS_RAM* containing the data for both read and write operations. This process transforms the two RAMs into a single one of SPREGHD type. During HLS, the use of two RAMs was to prevent conflicts between read and write operations within the same RAM during synthesis.
Furthermore, we introduced a logic block that deactivates the chip select of a RAM if the address being read in a particular cycle matches the address read in the previous cycle. This approach deactivates the read operation, giving the opportunity to write inside the same RAM without conflicts due to simultaneous read and write. It also save dynamic power.
Finally, although the scheduling ensures that in most cases, read and write operations occur in different cycles, we implemented the capability to stall the NPU if simultaneous read and write operations are detected, with a priority given to write operations.

In conclusion, the implementation of Gemini, considering its configurability and extensive validation, was greatly facilitated through HLS design. However, it is important to note that only the NPU was fully designed using HLS, which presented its own set of challenges due to the need for optimization. The RAMs, which were not well-handled by HLS, had to be incorporated through a wrapping of the NPU RTL. This provided an opportunity to address and resolve potential read and write conflicts that might occur during execution but also enabled the use of a single RAM for *fmaps*.

4.4 Gemini Tape-outs and Benchmark

The Gemini architecture, as discussed in Section 4.2 and implemented to tackle the challenges outlined in Section 4.3, has successfully undergone a tape-out in the P18 STMicroelectronics technology [123], as detailed in Subsection 4.4.1. Furthermore, its performance can be effectively compared to benchmark NN accelerators in Subsection 4.4.2.

4.4.1 Tape-outs

The Gemini project completed two tape-outs, both utilizing the advanced P18 technology.

The first tape-out, conducted at the beginning of my thesis, employed an earlier version of the Gemini architecture (only $WPAR$ parallelization was possible), not the final one described in this chapter. It was based on the Gemini-1 project outlined in Section 1.1. Thanks to this tape-out, with had our first Gemini demonstrator. However,

We encountered several complications during this initial tape-out. Firstly, the technology was not yet ready to support SRAMs, so the FMAP and WEIGHTS RAMs were mapped to registers instead of SRAMs. Additionally, there were issues with the setup measurement, preventing us from accurately measuring the performance of Gemini. Nevertheless, this tape-out proved to be highly efficient in promoting the P18 technology, as it facilitated a pipe-clean of the design flow and collected vital information for technology development. I was responsible for several implementation steps during this process, as detailed in Appendix A.

The second tape-out, carried out for the Gemini-2 project (Section 1.1), was more efficient and featured an architecture very similar to the one presented in this chapter for $(WPAR, MPAR) = (16, 8)$. Additional optimizations were implemented; however, they are not presented in this work because I was not directly involved in their implementation. This tape-out benefited from a more mature technology than the first one, making measurements more accessible. For this tape-out, my involvement was primarily in the design of the NPU, and I was not part of the entire process as with the first tape-out. The chip is ready, but we cannot present its performance measurements at this time. Nevertheless, we can extract key performance indicators from gate-level simulations on the placed and routed netlist, taking into account the routing, including back-annotations [95].

4.4.2 Benchmark

In this section, we conduct a comparative analysis between Gemini and various accelerators using the KPIs described in Section 3.3. Gemini offers configurability with different possible architectures, but for this comparison, we have chosen to focus on the test-chip of Gemini-2 with the configuration $(WPAR, MPAR) = (16, 8)$, which utilizes P18 technology, and offering the most precise estimations. Estimations and simulations for other architectures discussed in Chapter 5 are less accurate as they are operated before placement and routing.

However, it is important to note that this comparison is not entirely equitable due to variations in technology and metrics used by different accelerators, as discussed in Section 3.3. For example, in Table 4.1, Eyeriss, UNPU, and Orlando consider the entire chip for area measurement, whereas all others focus solely on the accelerator. This discrepancy poses challenges because not all accelerators are designed to accommodate all types of neural networks. For instance, our Gemini accelerator has RAM limitations that make it incompatible with networks like AlexNet, unlike Eyeriss. Furthermore, for some accelerators such as UNPU, ORLANDO, and COMPAC (for which it is not explicitly specified), determining the number of PEs posed challenges because they do not adhere to any of the standard data-flow paradigms presented in Section 3.2.

Regarding Gemini's area efficiency, it is very competitive due to the regularity of its architecture and, honestly speaking, advances in technology. This advantage is also evident in the $GOPs/mm^2$ KPI.

In terms of energy efficiency (TOPs/W), as mentioned in Subsection 3.3.4, the measurement can be complex due to variations in how operations are counted. When counting 2 operations per PE (equivalent to MAC operations), Gemini achieves an energy efficiency of $1.9TOPs/W$. However, if we count 3.1 operations per PE, accounting for the scaling operation within the PE (which is specific to Gemini and not performed by all accelerators), we achieve an efficiency of $3.1TOPs/W$.

Accelerator	PEs	Freq & Bits	Area	TOPs/W	GOPs/mm²
Envision (28nm) [92]	512	200 MHz & 8	1.87 mm²	3.80	-
ShiDiaNao (65nm) [40]	64	1 GHz & 16	0.66 mm²	-	293
Eyeriss(65nm) [25]	384	200 MHz & 8	12.25	0.246	-
UNPU (65nm) [79]	-	200 MHz & 8	16 mm²	4.30	43
QNAP (28nm) [91]	144	470 MHz & 8	1.9mm²	11.3	745
COMPAC(65nm) [107]	(128)	25 MHz & 8	1.74mm²	1.044	-
SCNN (65nm) [100]	64	1 GHz & 16	7.9mm²	-	-
Orlando (28nm) [38]	-	1.75 GHz & 8 and 16	34mm²	2.9	-
Gemini (18nm)	128	350 MHz & 8	0.655mm²	1.9/3.1	2560

Table 4.1 *KPIs comparison for different accelerators*

As discussed in Section 3.3, in my opinion, energy efficiency is not consistent for bench-marking, as different papers measure it for different NNs and for different setups. Some use peak performance on specific layers [107], others like us use averages, and still others base their measurements on pruned and adapted NNs to exploit sparsity. To address this, we conducted a comparison with accelerators using MobileNet for latency in Table 4.2, where Gemini proves to be highly advantageous. This is primarily because Gemini follows an output stationary architecture using $WPAR$ and $MPAR$, which aligns well with the depthwise layers found in MobileNet (Figure 2.13). We also measured power consumption while processing this network, resulting in $3.4mW$ of leakage and $4.8mW$ at $35MHz$.

Accelerator	Throughput (Fps) on Mobilenet
Eyeriss	1282
Edge TPU [96]	416
Gemini	1976

Table 4.2 *MobileNet latency of different accelerators*

Although direct comparisons may be challenging, it is evident that Gemini remains competitive. In general, its specifications align well with the requirements of low area, low power, and high speed essential for the edge computing domain. More specifically,

our customer has adopted the IP because it meets their requirements as outlined in their specifications.

4.5 Conclusion

In conclusion, this chapter has presented an overview of the design considerations for the Gemini IP architecture and its implementation. Gemini is purposefully designed to meet specific customer constraints. Firstly, it adheres to the quantization requirements following the TensorFlow paradigm. Additionally, Gemini is designed to support the processing of various layers used in feed-forward neural networks. The design of this accelerator was motivated by the need for configurability to accommodate the typical performance trade-offs required by different NNs applications. This configurability is achieved by adjusting just two key parameters, namely $(WPAR, MPAR)$, which configure the entire design. While the design choices were made to ensure configurability, it is important to note that they also maintain efficiency in optimizing area, power consumption, and latency for each specific configuration. The chapter provides a comprehensive description of the NPU architecture, elucidating each component, from PEs to mixers, and encompassing the storing stage. Additionally, the RAM organization is detailed, considering its adaptation to the NPU architecture and layer scheduling requirements. The chapter also addresses the implementation challenges of Gemini. As a reminder, the RAM data reorganization before execution is not managed by Gemini. NPUs are synthesized using High-Level Synthesis for different facilities. The NPU is then encapsulated within a top-level structure that instantiates the actual RAM modules. The preference for employing HLS as much as possible was driven by the flexibility it brings to the environment. Once the coding style is mastered, it improves readability, maintainability, configurability, and exhaustive validation capabilities.

Ultimately, this project resulted in a P18 tape-out, which provided a Gemini demonstrator. Furthermore, Gemini's performance demonstrates its strong competitiveness in the embedded AI domain due to its compact size, low power consumption, and impressive latency for a specific configuration. All these compelling attributes that led to the client's adoption of Gemini.

Thanks to a comprehensive understanding of Gemini architectural aspects and their practical implementation, it becomes feasible to derive precise close formulas that characterize Gemini. This enables the efficient selection of $(WPAR, MPAR)$ values to tailor the architecture to specific performance trade-offs and effectively target diverse markets.

CHAPTER 5

Gemini Performances Evaluation

Neural network accelerators are designed to optimize three key performance indicators (KPIs) in processing Neural Networks (NN): latency, power consumption, and chip area. Typically, there is a trade-off among these KPIs. Within the scope of the Gemini project, a configurable hardware NN accelerator (whose design, implementation, and benchmark are presented in Chapter 4), sharing its name with the project, was developed for the edge computing market. Its configurability aimed to make it versatile and adaptable to various NN applications with differing performance requirements, leading to specific trade-offs among the KPIs. However, determining the appropriate configuration for a client's NN application can be challenging. This challenge is magnified by the fact that measuring the KPIs requires time-consuming and resource-intensive simulations.

To address this client need, this chapter introduces a high-level practical estimator capable of rapidly predicting the KPIs based on the NN and the Gemini configuration. The latency is accurately derived using an analytical model based on the architecture, the operators scheduling and the NN characteristics. The power and the chip area are computed analytically, and the models are calibrated using simulations. Finally, we show how to use the estimator to derive Pareto optima for choosing the best Gemini configurations for a VGG-like NN. The research has resulted in a publication scheduled to appear at the SBAC-PAD conference in 2024 [98].

This chapter is organized as follows. We start in Section 5.1 by explaining the importance of having an estimator capable of predicting the KPIs of a NN execution based on accelerator architecture and neural network characteristics. This is followed by a comprehensive review of existing research in predicting KPIs for accelerators (Section 5.2). We then delve into the methodology adopted for this study in Section 5.3. Subsequently, Section 5.4 exposes the simulation environment used to gather data (thanks to industrial tools) used in determining KPIs model. Section 5.5 details the estimation model of the KPIs as well as its accuracy. Section 5.6 illustrates how the configuration can be chosen once the performances have been estimated. Finally, the conclusion is made in Section 5.7.

5.1 Importance of Performance Estimators

Deep Neural Networks (NN) have become incredibly popular [76]. We can find NN-based solutions in every field, which led it to become a field on its own. NNs present various structures and have different hardware requirements [75]: some applications need very low latency chips, such as cloud computing, while others require low power and small area, such as the edge computing market. Then, for each application, the designer has to usually find a compromise between the 3 KPIs: latency, chip area and, power.

The architecture of Gemini was imposed and primarily designed to be streamlined and highly configurable, facilitating effortless adaptation to various applications. Therefore, achieving pure performance on a specific NN was not the objective. As aforementioned in Section 4.2, Gemini is a configurable output stationary NN accelerator [119] with mainly two structural architectural parameters $(WPAR, MPAR)$. Chip area, latency, and power consumption depend on both the NN to be used and the two architectural parameters. The configurability of Gemini allows it to adapt to the NN structure.

Choosing the best configuration according to the NN for Gemini is too time-consuming. There are around 1000 possible Gemini configurations. For a fixed NN, measuring the KPIs requires simulating the NN execution. It cannot be done for all the configurations in a reasonable time, as it takes in average 2 hours to obtain the netlist and 3 min for the NN execution simulation. However, using accurate KPIs models, one could rapidly estimate all the possible configurations for a fixed NN. Thus, the challenge lays in obtaining the best KPIs estimation depending on the NN and the two structural parameters.

In this chapter, we consider only the scenario where the entire NN can fit into on-chip RAM. Consequently, only the accelerator's performance will be investigated. Considerations regarding off-chip communications are not taken into account, since they are not influenced by the choice of the Gemini configuration.

5.2 State of the Art

If the chip is available, the accelerators' KPIs are typically measured directly on the system for specific neural networks, eliminating the need for high-level KPI estimations. Complications may arise during the design phase when measurements are not yet possible. An important research area is the one dealing with the Design Space Exploration of accelerators (generally using FPGAs) [68, 9, 133, 16, 52, 4]. Their objective is to find the best architectural parameters according to KPIs. They use KPIs models and optimizing algorithms to find the best design solutions.

Most of the authors evaluate the latency using analytical formulas based on opera-

tions scheduling, accelerator architecture, and NN parameters [89, 42]. For example, Erdem et al. [42] evaluate the latency of the computation according to channel and kernel parallelization. The predictability of a NN execution renders this analytical approach the most commonly employed, it is even used by authors interested into evaluating the latency of a NN on heterogeneous machines composed of CPUs and GPUs [23].

Concerning the power or energy consumption, we cannot draw inspiration from what is typically done with GPUs and CPUs since they can directly measure task consumption, which is not the case for ASICs for which we cannot have a chip for each hardware configuration. In the context of ASICs, the strategies are often based on the power estimation of components [121, 124, 133]: for example, Wu et al. [124] developed Accelergy, a tool that evaluates the energy of different architectures accelerators. Firstly, a designer describes the architecture with compound components characterized by primitives components for which the power is known; RAMs power is evaluated with CACTI [13] and other primitives such as Multiplications And ACcumulations (MACs) are given by libraries. Secondly, the designer lists the actions of each component and their use rate. Accelergy estimates the total energy by combining all these data. Zhao et al. [133] also evaluate the consumption by listing the accelerator components but with more simplified energy models. They use also CACTI for RAMs power estimation, and they consider registers, MACs, and communication networks for the other components.

Concerning the chip area, Shahshahani and Bhatia [110] rely on machine learning models to predict it. The main drawback of this method is its lack of interpretability. For instance, the impact of each resource is difficult to estimate. Wu et al. [125] and Tang et al. [121] simply consider the area contribution of each component to evaluate the chip area.

5.3 Methodology

This section presents a method to estimate KPIs of an NN output stationary accelerator based on its configuration and NN parameters. The study aims to provide insights into the performance metrics without optimizing the architecture.

The proposed estimation methodology can be utilized by anyone using output stationary accelerator architectures. This is due to the fact that the estimation methods rely on principles inherent to this type of architectures, which are universal across all accelerators classified as such.

In Gemini, the latency is estimated analytically depending on the architecture, the operators scheduling, and the NN parameters. This estimation comes from the predictability and the regularity of the operations schedule.

In this study, we choose to model the power rather than the energy. The energy is impacted by the power of the system as well as its latency. Considering then the energy is less efficient when dealing with trade-offs between consumption and speed (the energy combines both of them). The power will be split into leakage and dynamic power. As a reminder, the leakage is the power dissipated when the device is powered up, but the gates are not toggling; it does not depend on the inputs. The dynamic one is the power dissipated when the gates switch their states; it depends on the inputs. Splitting the power allows us to estimate the power as a function of the clock frequency because the dynamic power scales linearly with the clock frequency while the leakage remains constant [35]. This statement holds true since we are operating with a constant voltage VDD. Below the maximum frequency (chosen during the synthesis), there is no requirement to adjust VDD in order to achieve the desired frequencies.
The power consumption of Gemini cannot be measured using tools such as Accelergy [124]. Indeed, the computing part of Gemini is designed using High-Level Synthesis (HLS); the number, the type and the use rate of components are then difficult to predict because operators schedule and optimizations (such as resource sharing) depend on the configurations. However, we assume that main compound operators such as registers, MACs or multiplexers must be synthesized during the HLS. A power model for Gemini is then exhibited based on a linear equation of the complexity of main operators and calibrated through simulations of NNs executions. An advantageous characteristic of this model resides in its inherent simplicity, as it necessitates a minimal quantity of data regarding the architecture and the NN (accessible from a high-level of abstraction) for its effective utilization. Conversely, this model also possesses the advantage of being explainable. The power estimator is based on gate-level simulations, which is sufficient to have accurate power values to compare several configurations.

Finally, the chip area will also be modeled with the area contribution of main operators multiplied by constants.

To evaluate the estimator accuracy, it is chosen to consider the Root Mean Square Error (RMSE). It has the advantage to be homogenous to the modeled parameter. The estimated RMSE for area and leakage is 0.005 mm^2 and 0.57 µW, respectively. The latency and power models of an NN are derived from the models of its individual constituent layers, encompassing all potential parameters. Therefore, these models are validated and universally applicable to any feed-forward NN. Latency is generally estimated with an error of less than 10 cycles, and dynamic power with a RMSE of less than 20 µW. We illustrate our results on a VGG-like NN, as presented in Figure 2.12, which is inspired by VGG-16 [115]. This network offers the advantage of encompassing diverse NN layer types and, notably, is extensively employed for evaluating NN accelerator performance [19, 133].

5.4 Building the Simulation Data set

The objective is to gather data on KPIs (latency, area, and power) via simulations for several NNs and $(WPAR, MPAR)$ couples. These data will be used to build an estimator based on analytical models predicting those indicators.

In this section, we will start by introducing the simulation environment in Subsection 5.4.1, which serves for collecting the essential data required for constructing our model. Following this, in Subsection 5.4.2, we will outline our methodology for choosing specific configurations and neural networks to create the data set.

5.4.1 Simulation Environment

For our utilization, both $WPAR$ and $MPAR$ vary from 2 to 32 ($N = WPAR \times MPAR$ could then vary from 4 to 1024). Each $(WPAR, MPAR)$ couple is called a configuration. As aforementioned, it was chosen to perform simulations at the gate level stage directly before the Placement and Routing (P&R).
We choose to work with 8 bits for both $fmapbits$ and $weightsbits$ since it is the most used quantization mode. As a reminder, $fmapbits$ represents the number of bits allocated for *feature maps* (*fmaps*) pixels, while $weightsbits$ represents the number of bits allocated for *weights*.
The SRAM capacity is fixed (65KB for *fmaps* RAM and 16KB for WEIGHTS RAM), only the aspect ratio between RAMs width and depth undergoes variation across different configurations.
The technology chosen is CMOSC40. The simulation environment is summarized in Figure 5.1.

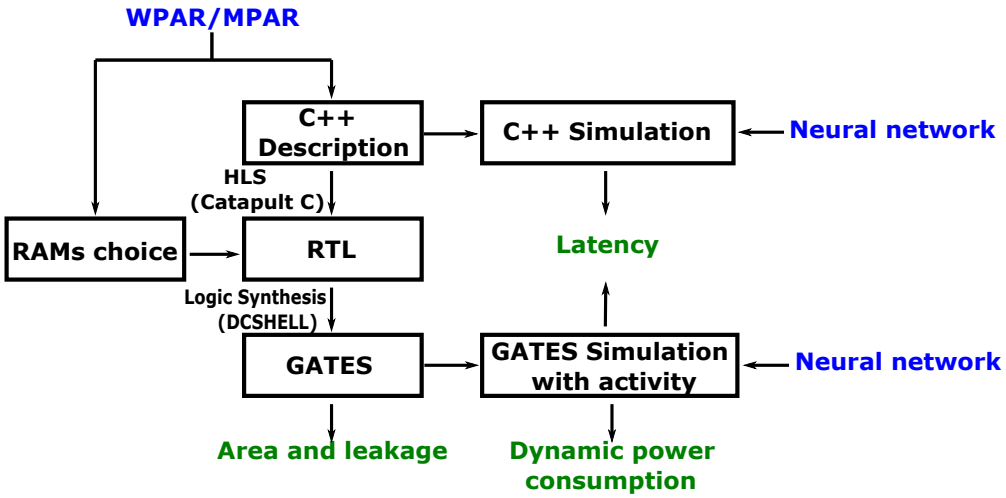

Fig. 5.1 *Simulation environment*

As a reminder, the design of the NPU is described in C++ and the HLS is performed by *SIEMENS CATAPULT®*. The NPU execution is described using loops, ensuring that

one loop execution in C++ corresponds to one clock cycle. Then the latency of a NN processing can already be measured at this level by counting the number of loops.
Once the NPU RTL is obtained for each configuration, the top level of the accelerator is built, instantiating the NPU and its corresponding RAMs. If several cuts are possible for one RAM, we choose the option giving the smallest area. Once the full RTL is ready (including NPU and RAMs), we operate the logic synthesis using the *SYNOPSYS DCSHELL®* tool with the same constraints and corners for all configurations: the synthesis is done at 200 MHz, 1 V, and 125 °C for the slowest corner. This corner represents the worst case in terms of timing. The library used is RVT (regular threshold voltage) in CMOSC40. It was decided to not mix different threshold voltage libraries to prevent the introduction of complex optimization by the synthesis tool that are challenging to model. Finally, when the gate netlist is ready, area and leakage power estimations are given by *DCSHELL®* without any simulation as they do not depend on the NN to be computed.
Latency and dynamic power estimations can be obtained by doing gate-level simulations. Tailored test benches are employed to fill the RAMs with requisite data and to launch the NPU. Gate simulations are performed using *CADENCE XCELIUM®* environment. The simulations are run at 1 MHz for the typical corner at 1.1 V and 25°C. The toggle rate is then exploited by *SYNOPSYS PRIMEPOWER®* to evaluate the dynamic power on the NN execution: the cycles where the RAMs are filled are not taken in account. Utilizing *SYNOPSYS PRIMEPOWER®*, accurate power estimation is achieved by employing an activity file that contains toggle rates for all cells and nets. It is more accurate than choosing arbitrarily a constant switching activity for all signals. Different methods exist for estimating dynamic power, with the most accurate being the time-based mode at the gate netlist level. This mode calculates power per cycle, accounting for all signal states during this cycle. However, this approach is time-intensive. An alternative mode is the average mode, which takes 10 to 20 times less time than the time-based mode. It computes the average power consumption over the entire simulation by considering the total toggling count of each signal. A limitation is that certain signal toggles might not result in power consumption, as demonstrated by scenarios like RAM operations where certain pins'(D and S) toggling does not lead to power use when chip select or clock enable pins are inactive. Comparing both modes in our simulation environment, we achieved under 2% on power estimation difference, despite significant differences in simulation time. Consequently, we opted for the average mode. More accurate modes are available, necessitating the generation of (Standard Delay Format) SDF files containing extensive delay data (paths, interconnects, etc.). However, as our study does not involve Placing and Routing the circuit, these modes lie beyond its scope.

5.4.2 Architectures and NNs to Build the Data set

A total of 214 $(WPAR, MPAR)$ configurations were meticulously selected to constitute the simulation data set. These configurations are detailed in Figure 5.1. They offer a comprehensive insight into how the structural parameters impact various KPIs. When considering which configurations to include, we had different approaches available. Initially, a straightforward strategy was considered, involving the uniform sampling of KPIs throughout the entire range where $WPAR$ and $MPAR$ vary from 2 to 32. However, this method assumes a linear behavior of circuits, which is not accurate given the known optimizations involving powers of two or exceptions for prime numbers. Moreover, such an approach would involve performing resource-intensive synthesis and simulation processes, especially for high $WPAR$ and $MPAR$ values, which might be unnecessary. Instead, we adopted a more gradual strategy, incrementally simulating configurations and observing their impact on KPIs. This approach allowed us to capture nuanced behaviors while also incorporating specific points like power-of-two and prime-number configurations. This progressive methodology led to the systematic construction of the final data set comprising 214 configurations. It will be used for interpolation and extrapolation to cover all the possible Gemini configurations.

$WPAR$	$MPAR$	**Number of architectures**
$2 \mapsto 32$	4	33
4	$2 \mapsto 32$	33
$2 \mapsto 24$	5	25
5	$2 \mapsto 24$	25
$2 \mapsto 20$	6	21
6	$2 \mapsto 20$	21
$2 \mapsto 16$	8	17
8	$2 \mapsto 16$	17
$2 \mapsto 8$	16	9
8	$2 \mapsto 16$	9
16	16	1
32	16	1
16	32	1
32	32	1

Table 5.1 *Gemini's configurations considered building the data set*

For gate-level simulations, we choose wisely which NN must be run to extrapolate the result of simulations into other NNs. For that, we run simulations on single-layer NNs varying all the possible parameters to cover all cases. The performance of any multi-layer NN is then obtained from the information of single-layer ones. The chosen NNs were run for the 214 architectures.

For fully connected layers, the number of input neurons N_{in} and output neurons

N_{out} are the parameters characterizing the layer. Table 5.2 presents the various NNs under consideration for this specific layer type, along with the objective behind their selection.

N_{in}	N_{out}	**Objective**
$25 \mapsto 500$	1	Observe the impact of N_{in}, N_{out} is fixed. 11 NNs tested.
50	$1, 4, 8, 16, 32$	Observe the impact of N_{out}, N_{in} is fixed. 5 NNs tested.

Table 5.2 *Simulated fully connected layers*

For convolutions, we vary the number of filters, filter sizes, 2D *ifmap* sizes, *strides*, and *padding*. Table 5.3 summarizes all the NN simulated and their objective. For depthwise and pooling layers, different *ifmap* sizes, *strides* and, *padding* are chosen. Their values coincide with those chosen for convolutions for the simulation data set. In total, 94 single-layer NNs are taken into account to build the data set: 16 for fully connected layers, 56 for convolutions, 11 for depthwise layers, and 11 for pooling layers. This selection suffices for depthwise and pooling layers since certain observations are analogous to convolutions, for which the scheduling approach is highly similar. Overall, the number of tests for each parameter was not predetermined. Instead, it was progressively increased to capture various behaviors. As a consequence, the parameters range was not consistently linear.
Finally, the NNs usually used by Gemini (in Subsection 2.3.2) were used to illustrate the accuracy of the estimations. However, they were not taken in account to build the model.

To offer a time estimate for assembling all this data, on average, the combined duration of HLS and logic synthesis is approximately two hours, and NN execution within an optimized simulation environment takes around 3 minutes. Data collection is a one-time process, necessitating repetition solely if the environment undergoes changes. For instance, opting for a different technology or quantization would mandate re-performing this step.

5.5 Key Performance Indicators Estimation

The objective of this section is to estimate the performances of the accelerator according to its configurations for each NN. Those estimations are done thanks to an analytical model based on simulations discussed in Section 5.4. This model gives latency (Subsection 5.5.1), area (Subsection 5.5.2), leakage (Subsection 5.5.2) and, dynamic power (Subsection 5.5.3).

M	$W.H$	$S.R.C$	*stride*	*padding*	**Objective**
X	X	X	X	*valid/same*	Observe the *padding* impact on 6 random convolutions.
24	1024	9	$s \times s$, $s \in \{1, 2, 3\}$	*same*	Observe the *stride* impact. 3 tests
$7, 14, 24$	1024	9	1×1	*same*	Observe the impact of filters number. 3 tests
24	$80 \mapsto 4096$	9	1×1	*same*	Observe the impact of large *ifmaps* size. 18 tests
24	$16 \mapsto 80$	9	1×1	*same*	Observe the impact of small *ifmaps* size. 10 tests
24	1024	9256	1×1	*same*	Observe the impact of filters size. 16 tests

Table 5.3 *Simulated convolution layers*

5.5.1 Latency Modeling

Latency in cycles is obtained at the C++ description level. As the design is fully pipelined, the difference between the number of cycles given by the C++ execution and the one obtained after gates simulations corresponds only to the ramp-up of the pipeline. This was observed for several NNs. As the NN's layers are processed serially and separately, the latency of the neural network execution corresponds to the sum of the layers' latencies added to a constant overhead independent of the NN (it includes the pipeline ramp-up). For this work, only the meaningful terms will be detailed. For example, the bias cycles will be neglected.
The following paragraphs detail the latency modeling of each layer type and the latency behavior of a NN of multiple layers.

5.5.1.1 Convolution Latency

The execution of the convolution is fully predictive. It can be computed based on the output stationary paradigm where $WPAR$ (among 2D *ofmap* pixels) pixels of $MPAR$ filters (among M filters) are processed simultaneously. The number of 2D *ofmap* pixels calculated corresponds to the size of the 2D *ifmap* excluding the vertical *padding* pad_v. This is a consequence of the execution duration being unaffected by the *stride* and the horizontal *padding* usage. Only the vertical *padding* is impacting the number of pixels calculated. So the number of 2D *ofmap* image pixels is $W(H - (R - 1)pad_v)$, with, W and H the width and height of the *ifmap* and R the height of the filter. Thus, the

number of cycles N_{cycconv} needed to compute a convolution is:

$$N_{\text{cycconv}} = \left\lceil \frac{W(H-(R-1)pad_v)}{WPAR} \right\rceil \left\lceil \frac{M}{MPAR} \right\rceil \times K_c \tag{5.1}$$

where K_c is the number of cycles required for one-pixel computation. As stated in Subsection 4.2.1, 3 stages are pipelined for the computation of one pixel; the latency of the full system is then approximately the latency of the slowest stage. The slowest one is the output computation stage of the PEs array. Every cycle, one filter *weight* is read, so the number of cycles needed to compute one pixel is $K_c = S.R.C$ where S, R, and C are respectively the filter width, height and, channels. The latency of maxpool and depthwise layers are derived from the same formula.

5.5.1.2 Fully connected Latency

Concerning fully connected layers, $Nout$ output neurons are processed simultaneously by N processors. It takes Nin cycles to process them; Nin is the number of input neurons. So the latency N_{cycfc} of a fully connected is:

$$N_{\text{cycfc}} = \left\lceil \frac{N_{out}}{N} \right\rceil Nin$$

5.5.1.3 Estimator Validation

Combining the last equations, the general shape of the latency Lat of a NN of $L+K$ layers follows Equation.5.2:

$$Lat = \sum_{l=1}^{L} \left(\left\lceil \frac{\alpha_l}{MPAR} \right\rceil \times \left\lceil \frac{\beta_l}{WPAR} \right\rceil \gamma_l \right) + \sum_{l=1}^{K} \left\lceil \frac{\delta_l}{N} \right\rceil \epsilon_l \tag{5.2}$$

with L the number of convolution layers, K the number of fully connected layers and α_l, β_l, γ_l, δ_l, ϵ_l are constants depending on the layer l type.
We deduce from Equation 5.2 that the latency is a decreasing curve with N.
Given the predictive nature of the execution, there are only few clock cycles difference (that can be calibrated) between predictions and simulations across all conceivable layer types and their associated parameters. Consequently, this characteristic extends to multi-layer NNs as well. We illustrate this result using the VGG-like network for $MPAR = 8$ shown in Figure 5.2. This choice is made for the sake of clarity in the curves, even though the observation remains consistent when varying $MPAR$ and $WPAR$. The estimation and simulation curves are nearly indistinguishable.

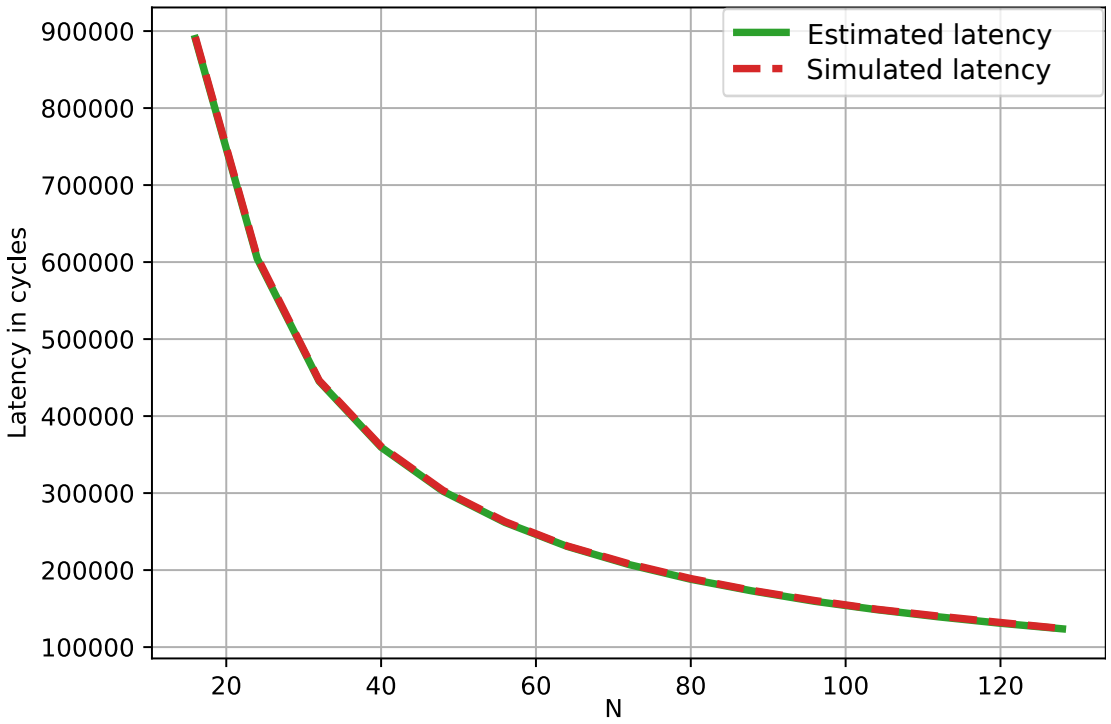

Fig. 5.2 *VGG-like estimated and simulated latencies for MPAR = 8*

5.5.2 Area and Leakage Modeling

In this paragraph, we will discuss the area and leakage model for RAMs and NPU as well as the calibration of the model by the identification of constants.

5.5.2.1 RAM Modules Area and Leakage

RAM modules leakage and area are dependent on the memory capacity chosen (total number of KBs). Even if the organization of RAMs ($width \times depth$) changes with configurations, the difference of leakage and area is only impacted by RAMs technology variation. Then RAM modules area and leakage will be considered as constants and only the NPU will be considered. A simple model of area and power is computed by multiplying the capacity K by a constant C determined by linear regression on several examples of RAM areas and power.

5.5.2.2 NPUs' Area and Leakage Modeling

5.5.2.2.1 NPU main operators' complexities Estimating the NPU power and area knowing only $(WPAR, MPAR)$ is challenging. The RTL is obtained by HLS, so the tool can adapt the number and types of operators and their scheduling to optimize the synthesis performance for each configuration; HLS can then generate different netlists for two close but different configurations.

It was decided to model the leakage and area with a linear combination of the expected main operators' complexities and then identify the constants thanks to a linear regression (which are different for area and leakage). These constants (c_0, c_1, c_2, and c_3) are all positive values that represent the consumption of fundamental operators. They are set positives to prevent negatives values for power and area. The compound operators taken into account are:

- Operators that do not depend on the configuration: the term c_0 corresponds to all the constant operators. As a matter of example, there are all the registers and logic units of finite state machines.
- PEs array input registers (*fmaps* and *weights* registers, accumulators) and arithmetic logic units (MACs): all these components scale with N. They will then be modeled with $c_1 \times N$.
- Mixers: there are 3 mixers in the circuit (*ifmap*, *weight* and storing stage mixers) ensuring that the data is well sorted at the input and output of PEs array and RAMs. These mixers are mapped into shifters implemented with multiplexers that have a complexity of $N\lceil \log_2 K \rceil$ with N is the total number of data sorted and K is the number of possible shifts for each data. Mixers are then modeled by: $c_2 \times N\lceil \log_2 WPAR \rceil$.
  Their power and area cannot be neglected, especially for a large N.
- Storing stage operators: they eliminate the non-useful pixels (due to *stride* or *padding*). They scale with $WPAR$. They are modeled with $c_3 \times WPAR$.

Area and leakage follow Equation.5.3 with different constants (G represent the area or the leakage):

$$\begin{aligned} G = c_0 + c_1 \times N \\ + c_2 \times N\lceil \log_2 WPAR \rceil + c_3 \times WPAR \end{aligned} \tag{5.3}$$

This modeling method can be adapted to any output stationary accelerator as the compound operators (except the storing stage that was optimized for Gemini) are consistently necessary: MACs and registers for computation and storage are always synthesized along with mixers used for data transfer from RAMs to PEs. Furthermore, the FSM operators are also present in any design. If a user has a customized accelerator with additional significant operators, their complexity can be included in Equation 5.3 by adding the complexity of the operators multiplied by a constant factor, which can also be determined during regression.

5.5.2.2.2 Identification of constants and validation of area and leakage models

The constants c_0, c_1, c_2 and c_3 are identified using the data set of 214 configurations.

They are determined quasi-instantly by linear regression optimizing the $RMSE$ and the correlation coefficient (R^2). For linear regression, we experimented with various tools and ultimately settled on the Python library Scikit-learn [102], which was slightly customized to ensure that the constants are forced to be positive.

	Leakage	Area
Root mean square error (RMSE)	0.57 µW	0.005 mm^2
Average	9.4 µW	0.11 mm^2
Correlation coefficient (R^2)	0.98	0.99

Table 5.4 *Leakage and area estimation characteristics*

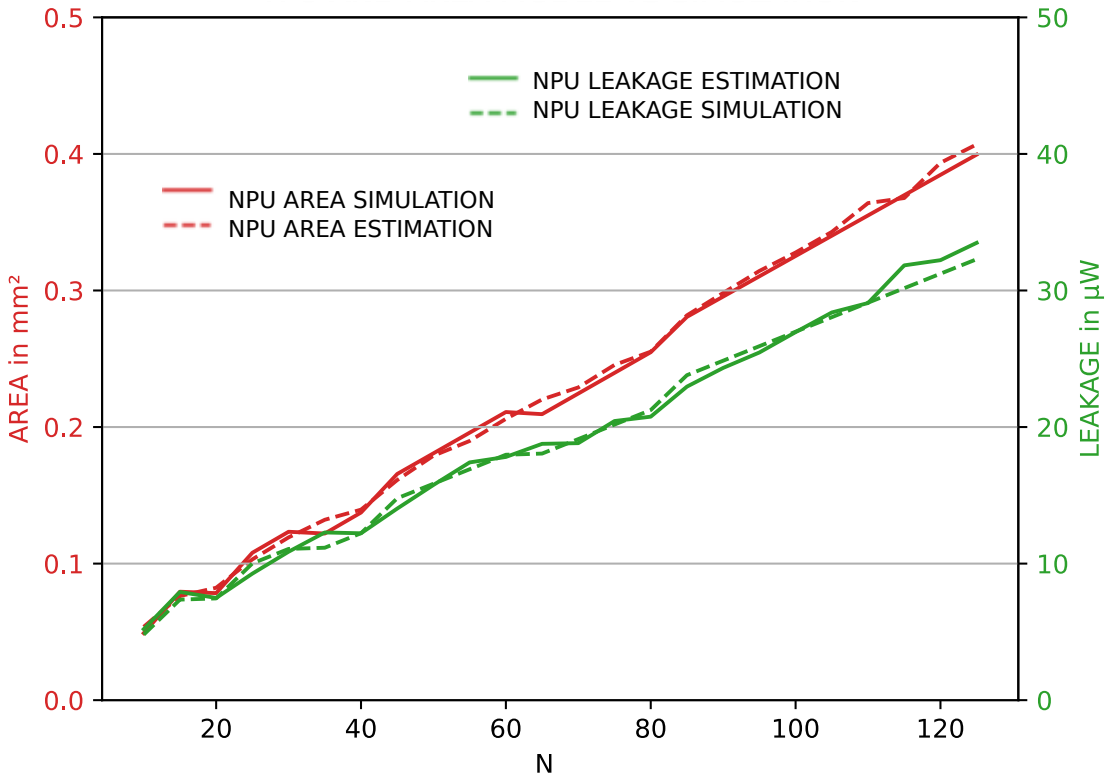

Fig. 5.3 *Area and leakage estimations and simulations for MPAR = 5*

Figure 5.3 illustrates the modeling results for $MPAR = 5$ configurations (chosen for clarity in the curves). It is observed that the significant increases in leakage and area, such as those observed at 40 and 80 Ns, are accurately captured by the modeling: they correspond to an increase of the value of $\lceil \log_2 WPAR \rceil$ ($WPAR$ is a power of 2). Table 5.4 displays the modeling results of all the 214 Gemini configurations. The low RMSE validates the accuracy of the estimation. The correlation coefficient R^2 close to 1 confirms that our modeling with Equation.5.3 is meaningful. The same approach can be applied in case of changes in the number of bits ($fmapbits$ or $weightsbits$) or

the process technology. Only the regression step needs to be rerun, using the updated simulation data.

5.5.3 Dynamic Power Modeling

This paragraph describes how the dynamic power of the execution of a NN on Gemini can be evaluated for each architecture configuration. The dynamic power is calculated by summing the internal power (consumption due to the power dissipation of the capacitance inside a standard cell) and the switching one (dissipation of load capacitance) [33].
First, the RAMs dynamic power will be discussed, then the NPU dynamic power for each layer type will be detailed. The dynamic power of any NN can then be estimated by combining the power consumption of its layers.

5.5.3.1 RAM Modules' Dynamic Power

For all NNs tested, the dynamic power of both RAMs (FMAPS and WEIGHTS) remains almost constant while sweeping $(WPAR, MPAR)$. It is since for each RAM, the number of KB is fixed for all configurations, thus the total amount of data read is the same; only the RAM modules widths and depths are changing affecting the number of read cycles and the size of the buffer to be read: for example, several reading cycles are needed when the RAM modules width is small while only a few of them are needed to read the same memory amount when the width is large.
Figure 5.4 shows the dynamic power of RAMs and NPU for the VGG-like NN for the configurations $MPAR$ = 8 ($WPAR$ is swept from 2 to 16). For this example, the SRAMs capacity is 1.3 MB. For our study, we neglect the impact of RAMs, as they do not impact the configuration choice.
However, a simple model of RAMs dynamic power consumption can be given based on their capacity K in KB. As done in Section 5.5.2.1 the dynamic power is obtained multiplying K by a constant C' determined also by linear regression.

5.5.3.2 NPU Dynamic Power Modeling

For estimating the dynamic power consumption of the NPU, we employed the same compound operators discussed in Section 5.5.2.2. However, in this case, the constants c_i in Equation 5.3 should be functions dependent on the specific NN parameters. It is important to highlight that individual models for each NN were not constructed by running separate regressions. Instead, a dynamic power single model, valid for all NNs, was developed through regression performed only once. This model incorporates data from all simulated NNs and is universally applicable across all NNs.

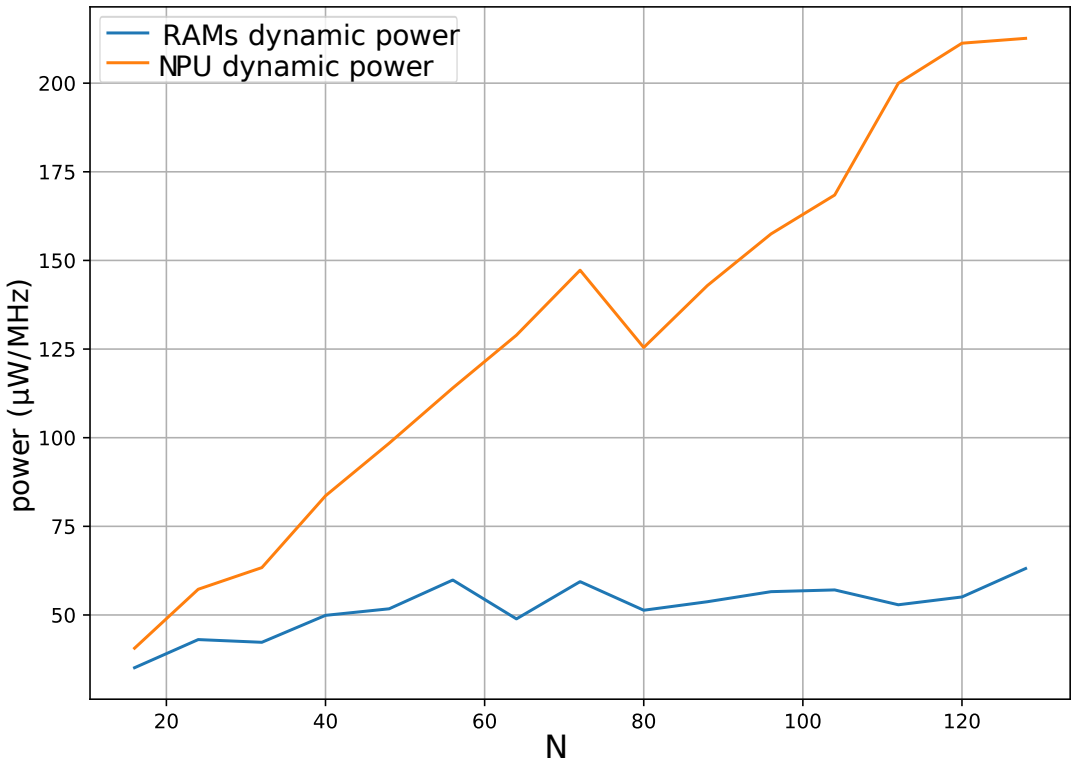

Fig. 5.4 *Dynamic power of NPU and RAMs on VGG-like for MPAR = 8*

The first statement is that the dynamic power of the NPU is globally increasing with N for all the NNs tested (as shown in Figure.5.4). Furthermore, there are some local optimums reached for some $(WPAR, MPAR)$. They are the same for all NNs tested, but they change according to the architecture. It means that they depend only on the architecture and not on the NN. However, the power magnitude of those optimums depends on the NN.

The identification of c_i is different between convolutions and fully connected layers, as they have different parameters.

For Convolutions, there are 5 different parameters characterizing the layer: *ifmap* 2D dimensions ($W \times H$), filter dimensions ($S.R.C$), number of filters (M), *strides* and, *padding*.

Firstly, as was specified in Subsections 4.2.2 and 4.2.5, all the blocks except the storing stage work, in the same way, considering different *strides* or *paddings*. Then a low dynamic power dependency on those parameters is expected. For the 214 configurations of $(WPAR, MPAR)$, we choose 6 different parameters NNs. We run them with and without the *padding* to evaluate their impact on the dynamic power. The RMSE is 2.94 µW for an average of 157.2 µW.

Concerning the *stride*, we took one NN with a *stride* of 1x1, another with 2x2, and another with 3x3. All the other parameters are the same. The RMSE is 0.72 µW for an average of 155.6 µW.

Due to this low RMSE (compared to the average), it was then decided to neglect the impact of *stride* and *padding* on dynamic power consumption.

The number of filters M should also be neglected. Actually, the mixers and PEs array are duplicated in $MPAR$; it means that the execution is the same considering any number of filters between 1 and $MPAR$; additionally, when $M > MPAR$ several same executions are operated which is not affecting the average dynamic power. Considering 3 different NNs, fixing all the parameters except M (respectively set to 7, 14, 24), the RMSE is: 6.04 µW for an average of 152.5 µW. This parameter can also be neglected because of this low RMSE value.

The dynamic power is increasing with the 2D *ifmap* pixels number ($W \times H$) before reaching a saturation level where the dynamic power consumption is almost the same for all *ifmap* 2D pixels number.

As it was specified before, $WPAR$ 2D pixels from the 2D *ofmap* are processed simultaneously. Considering the *padding* to simplify calculations, the number N_{exec} of executions to calculate all the 2D *ofmap* pixels is the following:

$$N_{exec} = \left\lceil \frac{W \times H}{WPAR} \right\rceil = \left\lfloor \frac{W \times H}{WPAR} \right\rfloor + r + o. \tag{5.4}$$

with r equal to 0 when $\frac{W \times H}{WPAR}$ is an integer and 1 otherwise. o corresponds to the few overhead cycles. They do not consume a significant amount of power. The term $\left\lfloor \frac{W \times H}{WPAR} \right\rfloor$ corresponds to executions where 100% of $WPAR$ are working and r to the execution where only a few of them are used (because there are less than $WPAR$ pixels to calculate). Thus, when $W \times H$ is large (large 2D *ifmap*), N_{exec} is quasi equal to $\left\lfloor \frac{W \times H}{WPAR} \right\rfloor$. As the dynamic power is measured with an average on all the convolution processing, the dynamic power will then correspond to the power of the executions where all $WPAR$ are working (because it is repeated $\left\lfloor \frac{W \times H}{WPAR} \right\rfloor$ times). It explains the power saturation when the *ifmap* is large enough (number of pixels higher than 80). Figure 5.5 shows the dynamic power of the NPU sweeping the number of the *ifmap* pixels for some configurations of $(WPAR, MPAR)$, all the other NN parameters are the same. The saturation comes with relatively small images for small $WPARs$; N_{exec} becomes quasi equal to $\left\lfloor \frac{W \times H}{WPAR} \right\rfloor$. For a large $WPAR$ value, the saturation happens for bigger images (higher $W \times H$).

For the power modeling, it was decided to consider the saturation by modeling the power of all *ifmaps* having more than 80 pixels by the power of 1024 pixels *ifmap*. Tested on 28 NNs with different *ifmap* sizes, the RMSE is 11.5 µW for an average of 134 µW. This assumption is relevant because the *ifmaps* used are usually in the range of saturation (even for a large $WPAR$). For small images (below 80 2D pixels), we have 2 models corresponding to *ifmaps* with respectively 16 and 36 pixels. The maximum

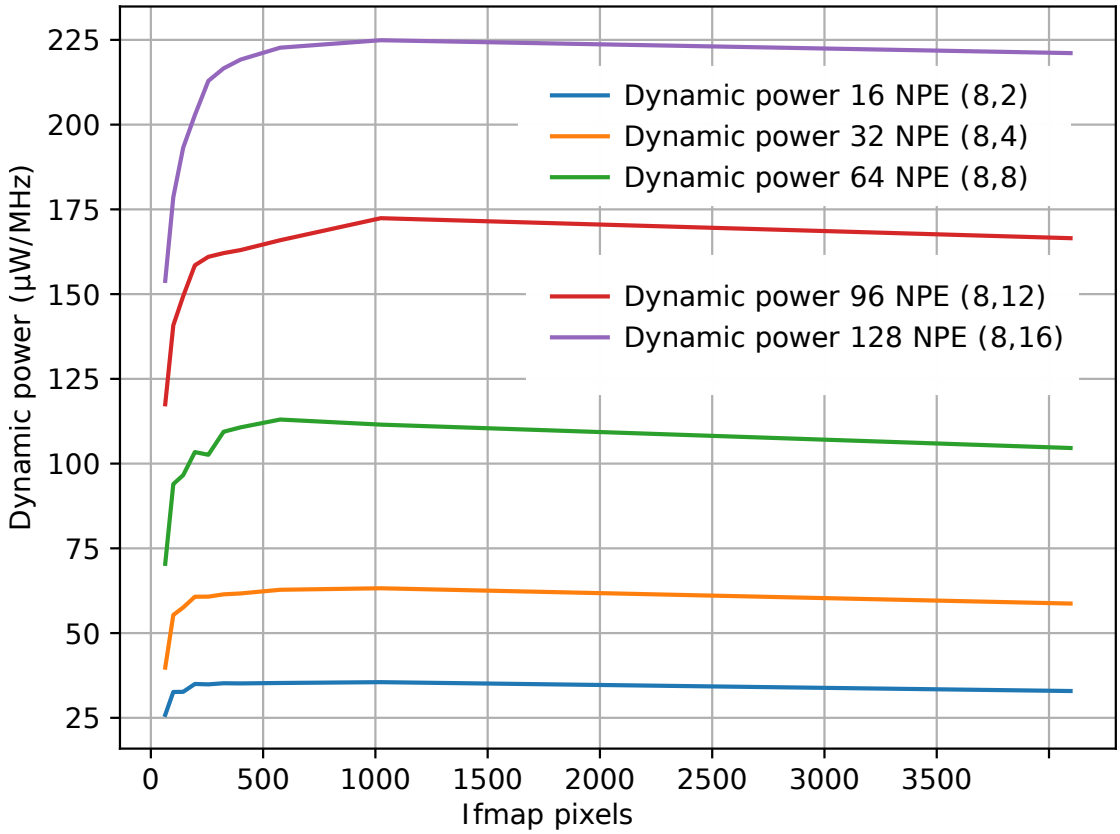

Fig. 5.5 *Dynamic power of the NPU executing convolutions sweeping 2D ifmap pixels*

RMSE is 10 µW.

Regarding the influence of filter dimensions, the dynamic power demonstrates a descending trend as the filter size ($S \times R \times C$) increases. The major impact is on the general slope. Figure 5.6 exhibits this statement on 3 NNs having the same parameters except the filter size. The dynamic power of convolutions is decreasing with filter dimensions because, as one *weight* is processed each cycle, the larger the filter dimensions, the higher the number of accumulation cycles required before the 5 scaling factors cycles. Thus, on average, the 5 cycles of scaling (see Subsections 4.2.2 and 4.2.5) do not have an impact on the overall power of big filters; on the opposite, for small filters, the 5 cycles have more impact because there are fewer accumulation cycles. This interpretation is confirmed observing the power consumption of the PEs array: Figure 5.7 depicts the dynamic power of the PEs array block as a function of the number of PEs executing a convolution, with varying filter sizes. The observation aligns with the NPU's behavior. The larger the filter window, the lower the average dynamic power consumption. Hence, this block is responsible for the observed behavior. To model this behavior, we choose to make the function multiplying N dependent on the filter dimensions, as it is the function affecting the slope. We estimated it for different filter sizes, and we generalize it with a regression (a power function was chosen as it gives the lowest RMSE

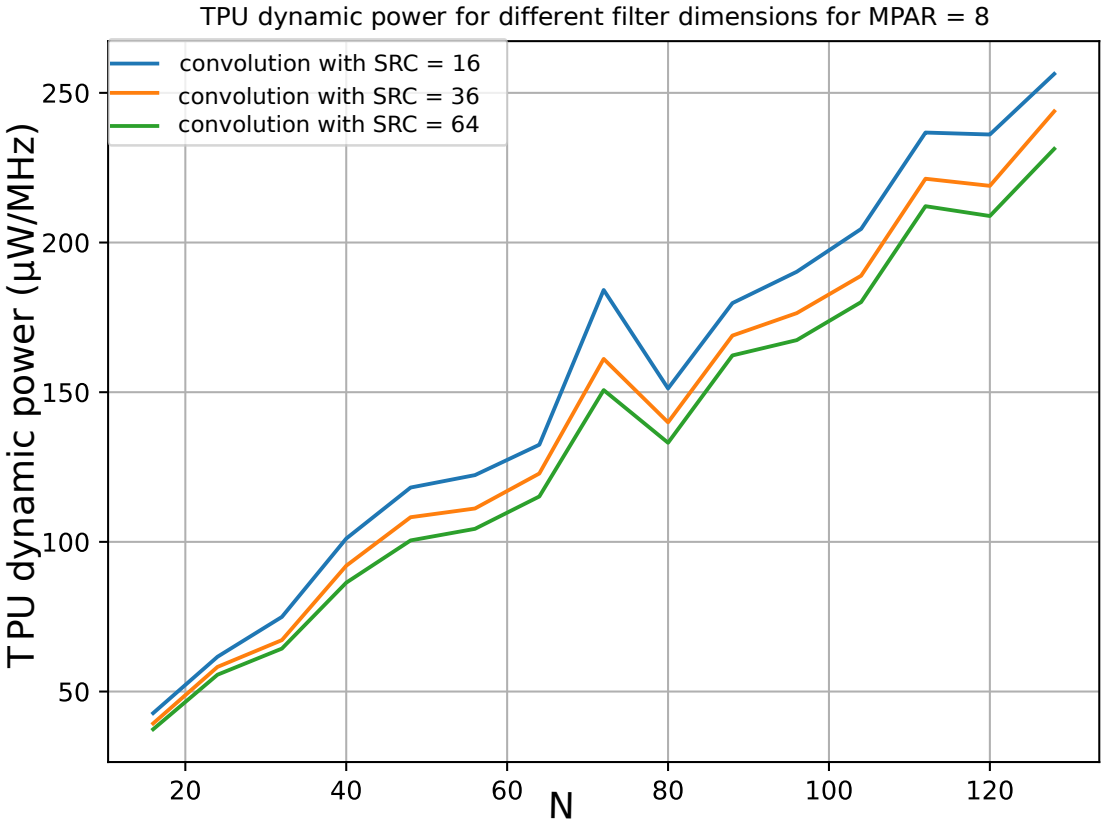

Fig. 5.6 *Dynamic power of the NPU executing convolutions with different filters sizes for MPAR = 8*

and highest correlation coefficient). The model explained below was tested on 16 NNs with different filter sizes for the 214 configurations: the RMSE is 16 µW for an average of 152 µW.

Finally, Equation.5.5 models the dynamic power of convolution layers P_c (as well as maxpool and depthwise layers):

$$\begin{aligned} P_c = c_0 + c_1 \times S.R.C^{c_2} \times N \\ + c_3 \times N\lceil \log_2 WPAR \rceil + c_4 \times WPAR \end{aligned} \tag{5.5}$$

Constants $c_i, i \in [0, 4]$ were determined by linear regression. The maximum error is theoretically lower than the sum of the errors of each approximation. The error generated using Equation.5.3 is impacting all the estimations.

Fully connected layers are characterized by N_{in} and N_{out}. The parameter N_{out} should not impact drastically the power consumption because as the architecture is output stationary, each PE calculates one N_{out}; $\lceil \frac{Nout}{N} \rceil$ executions are then needed to compute all the N_{out}. The N processors work the same way, even if N_{out} are less than N. 5 NNs are run with the same N_{in} varying N_{out} from 1 to 32. Only 5 µW of RMSE was observed (the average dynamic consumption is 78 µW). The impact of N_{out} is then

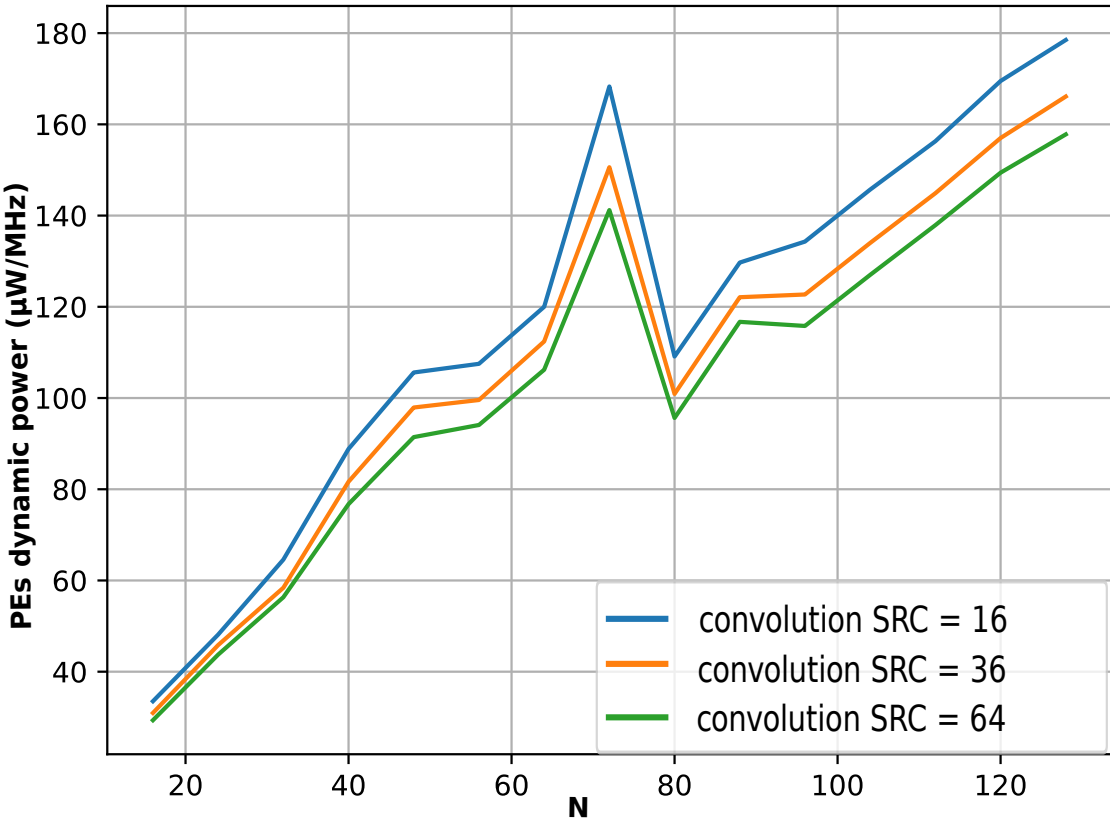

Fig. 5.7 *Dynamic power of the Processing Elements executing convolutions with different filters sizes for MPAR =8*

neglected.

For our applications involving fully connected layers, the value of N_{in} ranges from 25 to 500. The dynamic power model was developed based on simulations that encompassed this entire N_{in} range. As N_{in} increases, the dynamic power also increases. Nevertheless, this upward trend in power becomes less important as N_{in} reaches higher numerical values. It is due to the increase of the number of accumulations that are leading to higher nets activities. When N_{in} is already high, more accumulations are not affecting drastically the activity, so the dynamic power does not increase excessively. Figure 5.8 illustrates the dynamic power behavior of the NPU as a function of N when $MPAR = 8$, with N_{in} ranging from 200 to 500. Notably, we discern that as N_{in} increases, there is a corresponding increase in power (for the same N). Additionally, a comparison between the curves for N_{in} values of 400 and 500 reveals a less substantial difference compared to the contrast between 200 and 300, for instance. Given that this behavior primarily affects the PEs array, the associated constant will be adjusted accordingly. We opt for a logarithmic function (in terms of N_{in}) due to its gradual

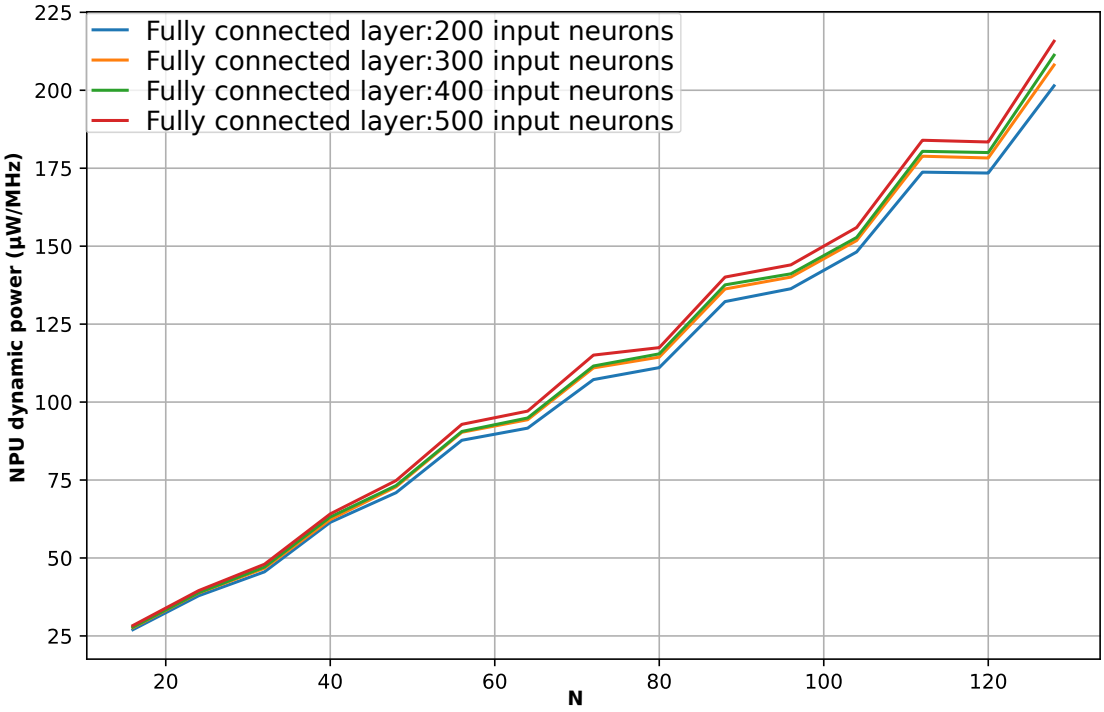

Fig. 5.8 *Dynamic Power of the NPU executing fully connected layers with different number of input neurons for MPAR = 8*

increase. Equation.5.6 models fully connected layers power consumption P_{fc}:

$$\begin{aligned} P_{fc} = c_0 + (c_1 + c_2 \times \log(N_{in})) \times N \\ + c_3 \times N\lceil \log(WPAR) \rceil + c_4 \times WPAR \end{aligned} \tag{5.6}$$

Constants $c_i, i \in [0, 4]$ were determined by regression. Tested on 9 fully connected NNs with N_{in} varying from 25 to 500 for 214 configurations of $(WPAR, MPAR)$, the RMSE is 12.3 µW and the average is 110 µW.

The dynamic power of ℓ-layers NN is estimated by combining the dynamic power of each layer. Once the dynamic power is estimated for each type of layer, the dynamic power $Pdyn_{NN}(f)$ of a NN composed of ℓ layers at the frequency f is calculated thanks to the average power of each layer weighted by its latency.

$$Pdyn_{NN}(f) = \frac{\sum_{l=1}^{\ell} Lat_l \times Pdyn_l \times f}{\sum_{l=1}^{L} Lat_l} \tag{5.7}$$

Lat_l is the latency of the layer l calculated with Equation.5.2 and $Pdyn_l$ is the dynamic power calculated with Equation.5.5 or Equation.5.6 according to the layer type.
Since the model has been validated (with low RMSE) across all layer types with all possible parameters, and considering that the dynamic power of a multi-layer NN is a combination of the individual layer powers, the model remains valid for the entire

NN as well. The model was subsequently tested on multiple NNs, yielding low errors. For example, on the VGG-like network (Figure 2.12) on 214 $(WPAR, MPAR)$, the RMSE is 16 µW for an average of 150.4 µW (11.6% of error). While our power model exhibits a higher RMSE when compared to alternative methods (such as [124]), it offers the distinct advantage of simplicity. The model relies solely on two structural parameters and raw information from NN characteristics, accessible at a high-level design abstraction. Despite the method's slightly elevated RMSE, it remains sufficiently low for effectively selecting the optimal configuration. Consequently, it continues to serve as a valuable tool in practical scenarios.
The estimation error arises from underestimating the influence of structural parameters, which likely affect operators with behaviors that are not consistently explainable (and thus not incorporated into the model). For instance, the local minimum at 80 in Figure 5.3 was excluded from the model (resulting in estimation errors) due to a lack of complete understanding (observed only for fixed $MPAR$ at 8). This could potentially be attributed to complex optimizations during High-Level Synthesis, which involve dividing operators into multiple units to allocate portions for non-concurrent tasks.

Finally, the energy E can be computed combining the latency and power models P (sum of dynamic and leakage power) as stated in Subsection 3.3.3 :

$$E = P(f) \times Lat \tag{5.8}$$

No estimation of error is required when approximating energy. The latency error is nearly negligible, so the overall error in energy estimation aligns with that of power modeling.

5.6 Configuration Choice

Some of the considered KPIs have antagonistic behaviors. By increasing N, the latency decreases but the power and area increase. To find the best trade-offs when selecting the $(WPAR, MPAR)$, we use Pareto fronts to determine the optimal architectures. Figure 5.9 shows sweet spots for the VGG-like NN considering the latency and power. The area is usually a specification, so only the points below a certain area could be considered. For each point of the curve, there are no other points that simultaneously have lower power and latency. The final choice between these $(WPAR, MPAR)$ points is made according to the application's specifications.

Finally, while the Pareto plot helps select $(WPAR, MPAR)$ for a specific technology (CMOS C40), we believe that the evolution of KPIs with respect to structural parameters may remain consistent across different technologies. Therefore, the choice of

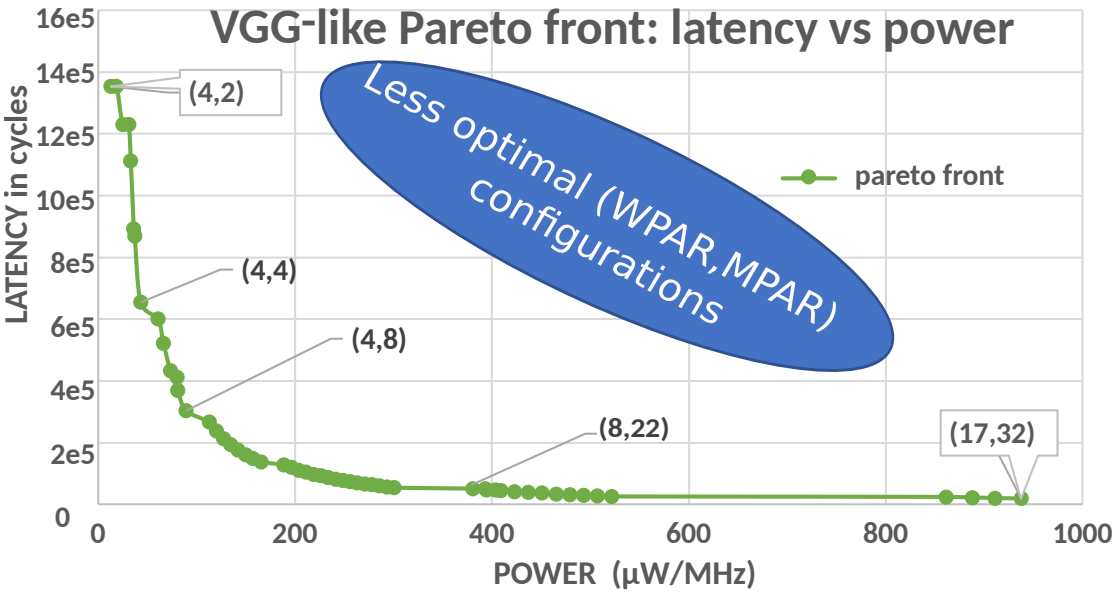

Fig. 5.9 *VGG-like network sweet spots*

$(WPAR, MPAR)$ could potentially be applicable across various technologies. However, the KPIs predictions are valid only for the chosen technology, and the models need adjustment when estimating for other technologies.

5.7 Conclusion

In this chapter, we presented a practical and explainable method to estimate three KPIs, latency, area, and power consumption of Gemini, which is, as a reminder, an output stationary near memory computing configurable accelerator for NN inference. Its architecture can be easily configured thanks to two parameters, $WPAR$ and $MPAR$. This method is based on simulations obtained with industrial tools. In fact, KPIs estimations are specific to any feed-forward NN specified as input of the estimator. They are accurate for all the KPIs but the dynamic power. The error is small enough to allow the user to determine the most accurate configuration for their application (NN).

This estimator is used at STMicroelectronics to evaluate Gemini's performance in the CMOS C40 technology. It is important to note that the company is transitioning to other technologies for different projects. Unfortunately, this estimator cannot be directly applied to estimate performance in these new technologies; it must be rebuilt using the same approach. However, even without reconstruction, it remains valuable for architectural decisions, as KPI behavior, relative to structural architectural parameters, is expected to remain consistent across technologies.

The KPI estimation method presented in this chapter is adaptable for utilization with any output stationary accelerator (except few optimizations done for Gemini). Fi-

nally, this method can be extended to applications based on the execution of several NNs with different use rates. It also opens up possibilities for high-level optimizations of accelerators, since we can derive closed-form performance formulas, especially the pipeline-based optimizations presented in Chapters 6 and 7.

CHAPTER 6

Fixed Pipelined Neural Network Accelerators

As previously discussed, Neural Networks (NN) accelerators have been introduced with optimized architectures aimed at enhancing the performance of NN executions. By increasing parallelization through the addition of more Processing Elements (PEs), the processing latency (*i.e.* the number of cycles required to process one image) of the Neural Processing Unit (NPU) is improved until a point where further parallelization of PEs does not yield more efficient processing.
However, when using a unique NPU, the images are usually processed one by one, which limits the throughput (*i.e.* the number of images inferred per second). This limitation becomes especially significant for stream applications, where throughput plays a crucial role. In this chapter, unlike the traditional approach of enhancing parallelization within the NPU for general Key Performance Indicators (KPI) optimization, our approach involves utilizing multiple NPU instances with fewer NPUs. These instances are organized into a pipeline to concurrently process multiple images. To clarify, this is a pipeline involving multiple accelerators, not to be confused with an execution pipeline within the same layer of a single accelerator, as presented in Subsection 4.2.2.

We focus on addressing the problem that we call the fixed hardware scenario: it pertains to the user's perspective where the NPUs and RAMs in the pipeline are predetermined and fixed (the NPUs number, order, and parallelization along with RAMs capacities are fixed). The aim is then to determine, for each NN, the allocation of each layer on NPUs ensuring that the NN is executed in an optimal manner. This problem is known and referenced as a Simple Assembly Line Balancing Problem [18]. Nevertheless, it is important to note that it encompasses additional constraints and incorporates more objectives than the initial description of the Simple Assembly Line Balancing Problem.

The objective of this study is not to conduct a direct comparison with state-of-the-art hardware acceleration solutions. Instead, our aim is to present a methodology that can be applied to accelerators for more optimizing KPIs. In this regard, our investigation will be illustrated through the implementation of a pipeline scenario using Gemini NPUs.

With this perspective in mind, our approach begins in Section 6.1 by examining the limitations of utilizing a single NPU, which subsequently leads us to the concept of a

pipeline. We will then delve into the details of the pipeline's structure in Section 6.2, along with an explanation of the KPIs objective to optimize in Section 6.3, supported by pertinent references from the literature on analogous challenges (in Section 6.5). Once this foundation is laid, we formalize the problem in Section 6.4, and we detail strategies for optimizing throughput and latency separately in Section 6.6, or simultaneously in Section 6.7 for the fixed hardware scenario. This will encompass the presentation of a linear modeling, polynomial solutions. Finally, empirical results on the Gemini architecture for each case (separate and joint optimization) are performed in Section 6.8.

6.1 Single NPU Limits

In this chapter, we will present the limits of processing *input feature maps ifmaps* by a NN on a single NPU.
Feed-forward NNs constrain the NN accelerators to process the layers sequentially. The drawback is that they usually allocate the same number of PEs to all layers. These accelerators typically utilize multiple parallelized PEs to reduce latency until a certain stage where further increases are no longer as efficient. Initially, the PEs are useful for all layers, but as the parallelization increases further, they become beneficial only for large layers (as the smaller layers are not impacted by the increase in parallelization). However, eventually, they reach a saturation point where the latency no longer decreases when N is too large. A majority of PEs could be then underused as they are working only when they process large layers. It has an even more significant impact on regular accelerators like Gemini, where PEs are not deactivated when they are not actively used for computing. As a result, these PEs continue to consume power unnecessarily, regardless of the computing necessity.
In addition, enhancing parallelization by significantly increasing N presents challenges. The need to increase N may demand additional resources that could be complex to handle during the placement and routing step or might lead to an increase in the critical path, making it impossible to achieve the targeted sign-off frequency.

Considering the Gemini latency model described by Equation 5.2, we deduce that when $MPAR$ is higher than α_i, increasing $MPAR$ will not result in lowering more the latency. The conclusion is the same for $WPAR > \beta_i$ and $NPE > \delta_i$. Figure 6.1 illustrates the latency of the NNs P-Net (Figure 2.14), VGG-like (Figure 2.12), and MobileNet (Figure 2.13) as a function of N with $MPAR$ fixed at 8. It is evident that for all three NNs, the latency experiences a significant reduction initially before gradually slowing down and reaching saturation: the augmentation of $WPAR$ impacts all layers with β_i greater than $WPAR$. Initially, when $WPAR$ is increased, the latency is signif-

icantly reduced as it affects all layers. As the increase continues, several layers reach a state of saturation, and only layers with large β_i values are affected. Consequently, the latency reduction becomes less important, benefiting only the larger layers (it is useless for the layers already in saturation). The saturation points occur at different stages for each NN. The observation is the same when fixing $WPAR$ and varying $MPAR$.

In addition to latency, as the *ifmaps* are processed sequentially, this latency wall is also impacting the throughput (inverse of latency) that also reaches a saturation point.

These observations question the PEs utilization: for a given NN, the number of PEs is chosen according to all NN layers. With this strategy, some PEs are useless for certain layers. Considering Gemini processing the VGG-like network, only 1 PE is necessary for the last fully connected layer (as only one output neuron is calculated). Processing the entire Network with large PEs number is beneficial for the other layers, but for this one, all the PEs except one are not working. A strategy where PEs are assigned per layer and not per NN is more optimal.

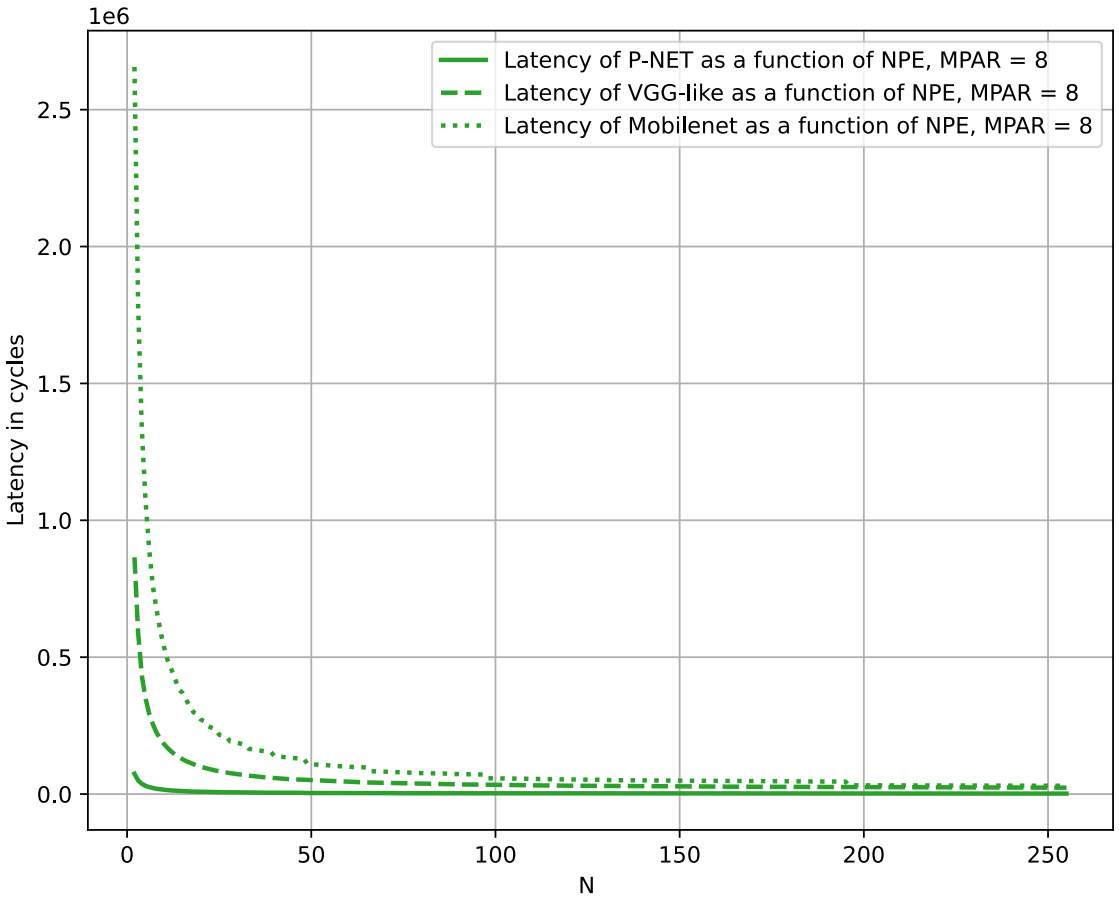

Fig. 6.1 *Latency as function of PEs number N for three NNs*

6.2 Description of the Pipeline Environment

In this section, we will explain the pipeline structure, which can be used with a wide range of accelerators. Subsection 6.2 will give a brief overview of how NN accelerators work, emphasizing the important requirements for our pipeline. Moving on to Subsection 6.2.2, we will focus on the NNs that will be run on the accelerator pipeline and provide clear notation for the layers and the intermediate *fmaps*. Subsection 6.2.3 will provide a detailed look at the pipeline's architecture, including the NPUs and RAMs. In addition, Subsection 6.2.4 will discuss the constraints for mapping the NN layers in the pipeline system. To wrap up, Subsections 6.2.5 and 6.2.6 will cover the capacity of the RAMs and the time it takes for the NPUs to execute, which are crucial details for our study of this issue.

6.2.1 NPU Accelerator Working Principle Reminder

Evaluating the potential performances of using multiple accelerator instances becomes a logical step after designing an instance (Chapter 4) and evaluating its proper KPIs (Chapter 5). While this approach was initially developed with a primary focus on Gemini, it is important to note that the applicability of accelerators to the pipeline extends beyond Gemini. It can be effectively applied to various other accelerators featuring distinct architectures, as the ones published by Zhou et al. [135], Parashar et al. [100], Sze et al. [119] and Du et al .[39]. The building block of the pipeline system can be any Near Memory Computing accelerator. It consists of a NPU responsible for layer computations. The NPU contains N parallelized PEs, and on-chip RAMs for storing NN *weights* and *fmaps* as illustrated by Figure 6.2. The partitioning of RAMs into multiple

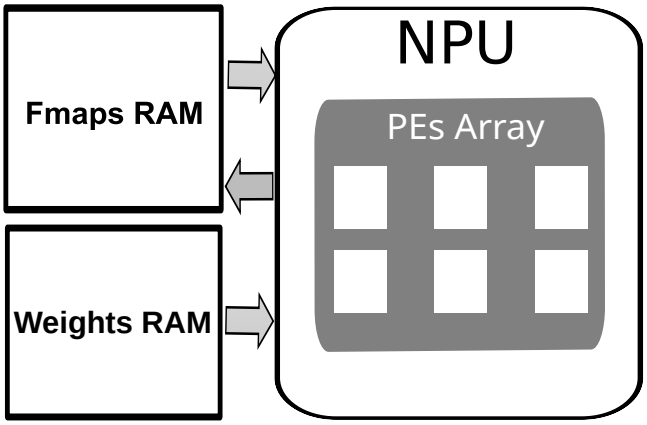

Fig. 6.2 *Accelerator general architecture*

cuts or considering them as multi-bank instances does not affect the problem at hand. The NPU reads and writes *fmaps* in the FMAPS RAM. The *weights* are read from the

WEIGHTS RAM. To streamline the study, we consider that the input *fmaps* and the NN *weights* are preloaded into their corresponding RAMs. This means that any time required for gradual writing of these data into the RAMs, such as through the use of a DMA (Direct Memory Access), is not taken into consideration. Typically, this gradual writing process is employed for large-sized NNs. If taken into account, it would impose limitations on the throughput but would not affect the problem's description, modeling, or solution. Finally, it is assumed that the NPU process the NN layers sequentially.

6.2.2 Description of the NN and Intermediary *Feature Maps*

The fixed feed-forward NN to be processed is composed by a set of $\ell > 0$ successive layers $\mathcal{L} = [L_0, L_{\ell-1}]$, as presented by Figure 6.3. The intermediary *fmaps* are denoted by $\mathcal{I} = [I_1, I_{\ell-1}]$; for any $j \in [1, \ell - 1]$, I_j is the *fmap* of size s_j (corresponding to the number of pixels) produced by the layer L_{j-1} and input of L_j.

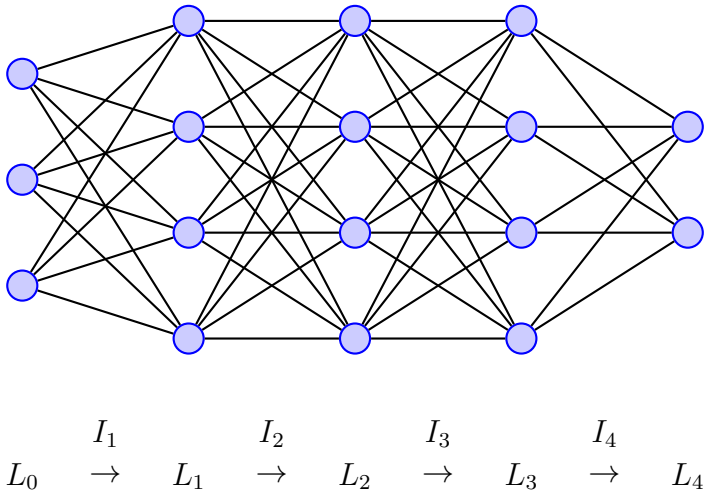

Fig. 6.3 *A feed-forward NN of $\ell = 4$ layers and the corresponding intermediary* IFMAP

6.2.3 Description of the Pipeline Architecture

The pipeline is composed of n NPUs instances $\mathcal{G} = [G_0, G_{n-1}]$ and $n + 2$ SRAMs $\mathcal{R} \cup \{\text{IFMAP}, \text{OFMAP}\}$ with $\mathcal{R} = [R_0, R_{n-1}]$ linked following Figure 6.4. All these components are integrated into a same chip. The two SRAMs IFMAP and OFMAP contain respectively the successive input and output *fmaps*; the other SRAMs are dedicated to intermediary *fmaps*. Each NPU $G_i \in \mathcal{G}$ contains N_i processing elements, while each SRAM $R_i \in \mathcal{R}$ has a capacity denoted as K_i (in KB). Each SRAM R_i is used by G_i for reading and by G_{i+1} for both reading and writing operations. If R_i is a single-port RAM, it is imperative that simultaneous access to the SRAM R_i from both NPUs G_i and G_{i+1} is not scheduled by the execution algorithm. However, if dual-port RAMs are

used, this condition is not necessary as dual-port RAMs allow for simultaneous access from multiple sources. For example, the NPU G_0 process the layer L_0 producing the intermediate *fmap* I_1 (having a size of s_1) stored in the RAM R_0.

On the other hand, our study does not consider WEIGHTS RAMs since they do not impact the optimization: indeed, the quantity of NN *weights* remains constant (using the pipeline or not). For the pipeline system, they are distributed into the multiple RAMs instead of being stored inside one WEIGHTS RAM: for instance, each NPU communicates with a dedicated WEIGHTS RAM (W_i in Figure 6.4), which stores the *weights* of the layers processed by that particular NPU.

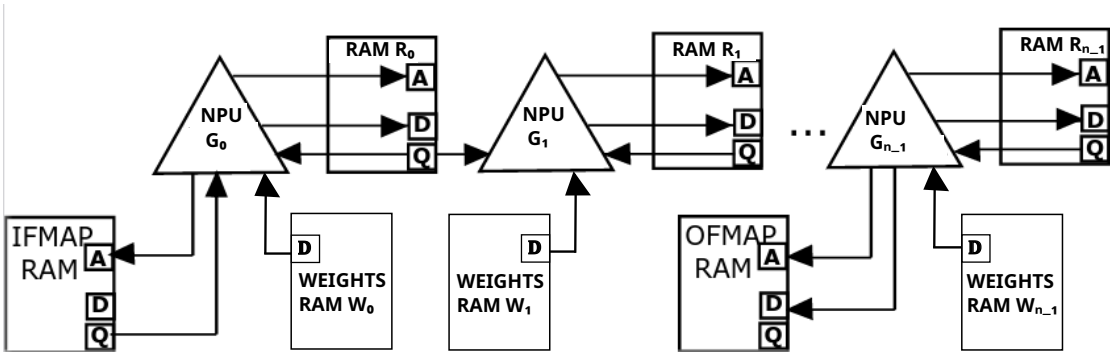

Fig. 6.4 *Description of pipeline of n NPUs with their correspond RAMs*

6.2.4 Layers Mapping on NPUs

As the NN layers are consecutive, the natural way to distribute them to $\mathcal{G}$ is using a pipeline scheme. Let us consider $\pi : [0, \ell - 1] \mapsto [0, n - 1]$ the function that maps the layers to the NPUs that process them: $\pi(j) = i$ means that the layer L_j is processed by the NPU G_i. The allocation function π must fulfil the following conditions:

- The layers L_0 and $L_{\ell-1}$ are processed respectively by G_0 and G_{n-1};
- Each NPU processes at least one layer;
- Each layer is processed by one NPU exactly;
- The mapping π is non-decreasing, i.e. if the NPU G_i processes the layer L_j, then any subsequent layer $L_{j'}$ with $j' > j$ is assigned to an NPU $G_{i'}$ with $i' \geq i$.

This way, the *ifmaps* can be processed simultaneously: when the pipeline ramp-up is completed, the n NPUs are processing at the same time n different *fmaps*. Each NPU executes a set of NN layers for each *fmap*: for instance, as soon as NPU G_i completes processing the allocated layers for image j, it can begin processing the same layers for image $j + 1$, while NPU G_{i+1} simultaneously processes the allocated layers for image j.

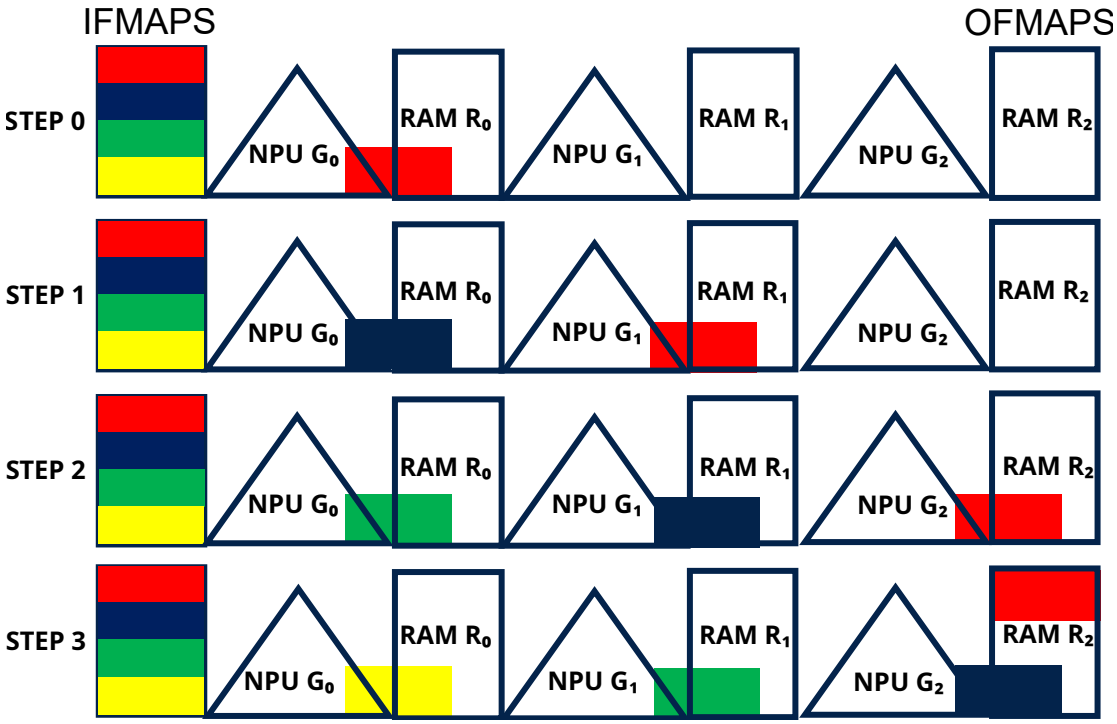

Fig. 6.5 *Pipeline execution principle with 4 ifmaps*

Figure 6.5 illustrates the operational principle for 3 NPUs processing 4 *ifmaps* (red, blue, green and yellow). During step 0, the first *ifmap* (colored in red) is read and processed by NPU G_0. It executes the layers assigned to it and stores the resulting *ofmap* in RAM R_0. At this stage, only G_0 is active. In step 1, after completing the processing of the red *ifmap*, G_0 proceeds to process the second *ifmap* (colored in blue). Simultaneously, NPU G_1 starts processing the red *ifmap*, while G_2 remains idle. During Step 2, NPU G_1 completes processing the red *ifmap* and transitions to the blue one. At the same time, NPU G_2 begins processing the red *ifmap*, and G_0 initiates processing the next one (colored in green). In Step 3, the pipeline is fully engaged, with all three NPUs operating concurrently, processing 3 *ifmaps* simultaneously. The initial three steps represent the pipeline's ramp-up phase. From Step 3 onward, three *ifmaps* are processed simultaneously in a repeating pattern until all the *ifmaps* have been processed.

6.2.5 Intermediary SRAMs Capacity

The two SRAMs, namely IFMAP and OFMAP store respectively the *ifmaps* and *weights* of the NN. Their capacity (measured in KB) will not be taken into account. This is since their size is exclusively predetermined by the NN. It is entirely independent of the mapping process or the architecture of the NPUs. However, it is crucial for the intermediate memories R_i for $(i \in [0, n-1])$, to have sufficient capacities to store the intermediary feature maps effectively. According to Figures 6.4 and 6.5, if a layer L_{j-1} is mapped to a NPU G_i (it means that $\pi(j-1) = i$), then I_j the output *fmap* of

the layer L_{j-1} is stored by R_i. Therefore, we can establish the following Property:

Property 1. *For any couple* $(i, j) \in [0, n-1] \times [1, \ell - 1]$ *and any layers mapping* π*, the intermediary* fmap I_j *is stored by* R_i *if and only if* $\pi(j-1) = i$*.*

Thanks to this property, we can determine if a RAM has enough capacity to store the intermediate *fmaps*. We distinguish two scenarios:

- The NPU G_i processes at least the layers L_j and L_{j+1}. It reads then the intermediate *feature map* I_j from RAM R_i and subsequently writes the intermediate *feature map* I_{j+1} into the same RAM. In this situation, it is imperative to prevent the unintended overwriting of I_j by I_{j+1}, as I_j is still required to complete the computation of all I_{j+1} pixels. To achieve this, the condition $(s_j + s_{j+1}) \leq K_i$ must be satisfied.
- The NPU G_i is processing only the layer L_j (it means that $\pi(j+1) = \pi(j) + 1 = i + 1$). Then, the RAM R_i exclusively holds I_j. It suffices to verify that only $(s_j) \leq K_i$.

These considerations are succinctly summarized by Property 2.

Property 2. *For allocations where* $\pi(j) = \pi(j+1) = i$*, it is necessary to verify* $(s_j + s_{j+1}) \leq K_i$*. If, there exists a unique* $j \in [0, \ell - 1]$ *such that* $\pi(j) = i$*, it must be ensured that* $(s_j) \leq K_i$ *holds.*

This Property can be adapted for a set of layers $[L_g, L_h] \in [0, \ell - 1]^2, g < h$, processed by G_i, to determine an upper bound on the size of RAM R_i that should accommodate the layers $[L_g, L_h]$. This upper bound is denoted as $\widehat{K}_{i,[g,h]}$.

Theorem 6.2.1. *Let us consider a set of layers* $[L_g, L_h] \in [0, \ell - 1]^2, g < h$*, processed by* G_i*. Then,* $\widehat{K}_{i,[g,h[} = \max_{j \in [g,h]}(s_j + s_{j+1}))$ *is the minimum feasible size of the SRAM* R_i*.*

Proof. All the layers L_j for $j \in [g, h[, g < h$ are executed by G_i, and their output *fmaps* I_j are stored in R_i. Therefore, each L_j must satisfy Property 2. Consequently, the maximum of $(s_j + s_{j+1})$ is the minimum required size for R_i. □

6.2.6 Execution Time

We assume that the execution time (or latency on a single NPU) of a layer L_j by the NPU G_i can be represented as $p_{i,j} = y(L_j, N_i) + c$, where y is a monotone function of N_i, and c is a constant value. The function y characterizes the execution of NN on the accelerator, and it depends on the NPU architecture and operators scheduling: naturally, regardless of the parallelization, increasing the parallelization level N_i leads

to a reduction in latency (or remains unchanged if no further parallelization is possible). Thus, it is expected that y does not increase in all cases. The constant c corresponds to the cycles required for initiation and execution that are independent of the specific NN or the value of N_i. The execution time of a set of consecutive layers $[L_g, L_h]$ for $0 \leq g \leq h \leq \ell - 1$ by G_i is then expressed as:

$$p_{i,[g,h]} = \sum_{j=g}^{h} y(L_j, N_i) + (h - g).c \tag{6.1}$$

For any mapping π, we also note $p_{i,\pi}$ the total execution time of the successive layers mapped to G_i.

6.3 Objective Functions Considered (or KPIs)

In the context of the fixed hardware scenario, there exists no flexibility in terms of choosing the architecture of each NPU or their placement within the system. This scenario is regarded as a user-centric challenge, where only KPIs that remain independent of the hardware configuration can be considered in order to determine π. In Subsection 6.3.1, we provide an explanation of how the throughput of a pipeline system is calculated and characterized in terms of the period. Subsequently, in Subsection 6.3.2, the focus shifts to latency, highlighting that it may possess two distinct interpretations contingent upon the specific use case and the objectives of the application.

6.3.1 Throughput and Period

In a pipeline system, n *fmaps* are processed simultaneously (when the pipeline is filled). Actually, an NPU can start executing the layers assigned to it only when its inputs are ready, otherwise, the NPU is stall waiting for the preceding NPU to finish its execution. This way, the slowest NPU determines the throughput because it fixes the cycles that other NPUs have to wait. The **throughput** T of a mapping π is then given by the execution time of the slowest NPU, i.e., $T(\pi) = \min_{i \in [0,n-1]} \frac{1}{p_{i,\pi}}$. The associated **period** P is then defined as $P(\pi) = \frac{1}{T} = \max_{i \in [0,n-1]} p_{i,\pi}$ and has to be minimized. In the following, $P^\star$ denotes a fixed upper bound of the period. It is expressed in clock cycles.

An important remark is that aiming to maximize the throughput results in accelerating the processing of the slowest NPU, which intuitively corresponds to balancing the execution times across NPUs. Accelerating the slowest NPU means slowing down the fastest ones and vice versa, which results in balancing time executions.

6.3.2 Latency

The **latency** (expressed in clock cycles number) corresponds to the total execution time of the NN. Two different definitions Lat_1 and Lat_2, can be considered according to the utilization:

Lat_1: if the NN processes a flow (or stream) of *fmaps*, several successive executions of the NN are launched and *fmaps* are processed simultaneously by the pipeline. The execution time of the treatment for each NPU is fixed to the maximum period P (due to other NPUs needing to wait for the slowest NPU for synchronization), resulting in a total latency of $Lat_1(\pi) = n.P$. In this context, optimizing the latency Lat_1 is equivalent to optimize the throughput. An application example is a wake-up system that operates continuously, such as face detection.

Lat_2: if the NN processes particular *fmaps* that are not part of a continuous flow, we can treat them as being processed sequentially (as if we were using only a single NPU). In this case, we consider the latency Lat_2 as the number of cycles taken from input to output for each individual processing: $Lat_2(\pi) = \sum_{i=0}^{n-1} p_{i,\pi}$ An application example is a system that only needs to process certain images if they have already been selected (through the wake-up system for example) such as face recognition.
Intuitively, when optimizing Lat_2 (in scenarios where *ifmaps* are processed one by one), an optimal approach for layer mapping is to utilize the fastest NPU for the maximum feasible number of layers while adhering to the specified π constraints. This strategy of grouping layers onto a single NPU may potentially conflict with the objective of optimizing throughput (or Lat_1), which involves balancing the load across all NPUs. Consequently, optimizing throughput or Lat_1 may present challenges when concurrently aiming to optimize Lat_2.
Lastly, it is essential to emphasize that the strategy involving multiple NPUs is inherently less advantageous when the goal is to optimize Lat_2 compared to using a single NPU with the combined PEs of all NPUs in the pipeline. This difference arises because adding more NPUs introduces an additional overhead factor (c in Equation 6.1). Nevertheless, it is important to acknowledge that our analysis is conducted within the context of a fixed system architecture, which may have been designed to target objectives beyond the sole optimization of Lat_2.

6.4 Formal Description of the Problem

An instance of our fixed hardware problem is characterized by several fixed parameters: the structure of a NN with a specific number of layers ℓ and the dimensions of intermediate *fmaps* ($s_j, j \in [0, \ell - 1]$), a predetermined pipeline architecture with fixed NPUs' architectures (n and $N_i, i \in [0, n - 1]$) and RAMs' capacities $K_i, i \in [0, n - 1]$, and

an optimization objective denoted as φ. The order of NPUs is also predefined. This objective can be selected from among the KPIs outlined in Section 6.3. The problem consists then to compute a mapping π that minimizes φ.

The described pipeline approach is versatile and can be applied to any accelerator that sequentially processes NN layers, regardless of the hardware architecture and operator scheduling. With this consideration, we can optimize objectives.

6.5 Related Work

The fixed hardware problem, despite being centered on the organization of NN accelerators into a pipeline, is actually part of a broader issue related to task mapping on computational machines (that could be accelerators or not). In this section, we will refer to the computing machines as processors In this context, two distinct research areas emerge: one that focuses on mapping a wide class of algorithms onto processors and another that is particularly concerned with the mapping of NNs specifically.

The first research community deals with general-purpose algorithms, often using less optimized mapping solutions compared to NN-specific approaches. In these cases, the use of CPUs or GPUs is expected, as the applications are generally applicable and not tied to a specific accelerator.

Conversely, the second community is more likely to develop mapping algorithms that leverage the inherent structure of NNs. In such cases, the utilization of pipelines across layers becomes a more prevalent approach. Here, NN accelerators are typically deployed, as the applications are restricted to NNs.

In this section, we present the literature relevant to the topic of NN accelerators in pipelines. Subsection 6.5.1 provides a brief overview of strategies for mapping general algorithms, whether they are NNs or not, onto multiple processors. Following that, Subsection 6.5.2 delves deeper into the literature, specifically focusing on mapping NNs to processors, whether in a pipelined configuration or not. Finally, Subsection 6.5.3 concentrates on the classification of the problem as a research operational issue known as the Simple Assembly Line Balancing Problem, and it presents its resolution for optimizing KPIs equivalent to throughput (or Lat_1).

6.5.1 Mapping General Algorithms onto Heterogeneous Machines

Research on optimizing algorithm execution speed (corresponding to throughput or latency as defined in Sections 6.3.2 and 6.3.1) in heterogeneous computing environments (comprising GPUs, CPUs, and accelerators) is a vast area of study. Two main categories of mapping heuristics exist: those that assign one task at a time and those that map all tasks simultaneously[8]:

Heuristics assigning one task at a time: in the first category, heuristics like Opportunistic Load Balancing, Minimum Execution Time, Minimum Completion Time, or K-Percent Best [27] are characterized by low algorithmic complexity and short mapping times. However, they tend to yield suboptimal results. Researchers such as Li et al. [80] and Zhou and Liu [136] represent algorithms as graphs, with or without precedence constraints, and map them onto heterogeneous systems to minimize completion time. To achieve this, they employ optimization algorithms like heterogeneity ratio-based mapping algorithms, structure rank-based heuristic algorithms, and data partition algorithms, resulting in optimal Integer Programming solutions (IP) [41]. Their approach can be applied within the context of NN mapping, which is relevant to our scenario: feed-forward NNs can be easily represented as graphs, where each layer corresponds to a task. There are precedence constraints, as each layer $L_j, j \in [0, \ell - 1]$, must be executed after the preceding layer. In this context, completion time (of the last layer) corresponds to Lat_2 due to the interdependence of each layer on its predecessor.

Heuristics mapping all tasks simultaneously: on the other hand, mapping heuristics like Min-Min, MaxMin, Simulated Annealing, or Genetic Algorithms [27, 36] yield significantly better solutions but tend to take longer to execute. For example, Alexandrescu et al. [7] employ genetic algorithms with two fitness objectives: makespan, which corresponds to the total duration required to complete a set of tasks (equivalent to latency Lat_2 in our context), and load balance, which pertains to throughput optimization (as it involves optimizing the slowest processor, contributing to overall throughput optimization). The execution time of these algorithms depends on genetic algorithm parameters, such as initial population size and the number of generations. However, they are well-suited for obtaining acceptable solutions within a reasonable time-frame.

In this paragraph, we have demonstrated that general methods used for task mapping on heterogeneous machines can be adapted for mapping NNs onto multiple accelerators. In fact, heterogeneous systems that combine GPUs and CPUs serve a similar function to our system comprising diverse accelerators with varying levels of parallelization and computing capability. Nevertheless, these general methods seem to be less efficient when contrasted with algorithms that take into account the inherent structure of NNs.

6.5.2 Mapping NNs onto Heterogeneous Machines

Several researchers have explored the mapping of NNs onto heterogeneous processors, with strategies categorized as either parallelization of individual layers or layer pipelining:

The parallelization of individual layers involves dividing the computation of a single layer among multiple processors. This approach relies on techniques to distribute the matrix multiplications required for convolutions or fully connected layers across

different processing units. However, this method has certain drawbacks. Firstly, it necessitates communication between processors handling image boundaries. Secondly, synchronization is required among processors processing pixels at different speeds. Lastly, it entails duplicating some NN data, such as the *weights*, which are used by all the processors.
For instance, Zhong et al. [134] parallelize NN execution across various processors, including FPGAs, NEON SIMD engines, and dual-core ARM processors. They have developed a scheduler that partitions different threads across the processors. Additionally, Liu et al. [82] convert the computation of convolutional and fully connected layers into large-scale matrix multiplications and pooling layers into row computations that can be effectively parallelized across multicore machines. Their accelerator achieves a 36.1% speedup compared to NVIDIA V100 GPUs for inferences. Finally, more advanced methods not only parallelize layer execution but also fine-tune the NN to better suit their heterogeneous systems. Actually, Odema et al. [97] have developed MAGNAS. This framework relies on genetic algorithms to schedule operations on processors and simultaneously fine-tunes the NN parameters to better align with their respective processors.

The pipelining layer method shares similarities with the approach outlined in Section 6.2.3. Following this execution strategy reduces runtime without introducing additional memory space requirements compared to a standard execution utilizing a single processor. This is because there is no need to duplicate NN data, as the processors handle distinct data for processing.
For instance, Kim et al. [69] introduce NeuroPipe, a technique that divides the NN into groups of consecutive layers and pipelines their executions using various types of processors. This approach accelerates NN inference, particularly enhancing throughput. Their primary focus lies in optimizing energy efficiency: the acceleration either enables processors to operate at lower voltage and frequency while achieving a targeted throughput, or it can deliver faster inferences at the same energy consumption. Similar considerations in a different context are explored in Subsection 7.7.6. On a system consisting of NVIDIA Jetson AGX Xavier with 64 tensor cores and an eight-core ARM CPU, NeuroPipe demonstrates an average reduction of energy consumption by 11.4% without sacrificing performance, or it can achieve 30.5% greater performance for the same energy consumption. Scheduling is performed after a batch of tests on the different processors, involving adjustments like moving layers from one processor to another, although no specific optimization algorithm is applied to manage the scheduling. Meanwhile, Maleki et al. [85] develop a high-level heterogeneous multicore processing chip scheme for efficient NN processing. Layer mapping onto the various cores is carried out through an algorithm inspired by the branch and bound algorithm to optimize the

load balance across processors [84] (equivalent to optimize the throughput).

Lastly, Symons et al. [117] also group layers and execute them on multicore accelerators, essentially acting as heterogeneous machines. They have developed the *Stream* scheduler (based mainly on genetic algorithms), which optimizes NN execution with a goal of minimizing energy consumption, reducing latency, and/or minimizing memory footprint for constrained edge devices, including both single and multicore architectures.

Certain studies, such as Yang et al. [126], employ a combination of both techniques. They first divide the NN into groups, which they refer to as stages, managed by distinct sets of processors. Within each stage, they further divide the calculations among different processors, a strategy they term "Parallel-pipeline execution." Their specific focus is on the Tiny Yolo algorithm, making them less reliant on general scheduling methodologies for mapping other NNs.

6.5.3 Simple Assembly Line Balancing Problem

We observe that the fixed hardware problem represent specific variants of the Simple Assembly Line Balancing Problem (*SALBP*), as defined by Boysen et al. [18]. The *SALBP* has been extensively applied to describe industrial problems since the last century [1, 44, 17, 21, 43]. The *SALBP* problem is characterized by the following inputs:

- The presence of multiple workstations, labeled as $k = \{1, ..., m\}$, which are typically arranged along a conveyor belt or a comparable material handling device. Work-pieces or jobs move sequentially through these workstations until they reach the end of the production line. In our specific context, based on the definitions provided in Section 6.2.3, the workstations align with the n NPUs, while the conveyor belt facilitating communication between jobs corresponds to the RAMs. Lastly, the work-pieces correspond to the *ifmaps* that need to be processed.
- At each workstation, a specific set of operations is carried out on the work-piece, and the duration between two entries of a work-piece at a station is referred to as the cycle time. The fundamental objective of the SALBP is to efficiently allocate the assembly work across all stations, guided by certain optimization criteria. In the context of our challenge, the cycle time aligns with the execution time $p_{i,\pi}$ of the slowest NPU. The objectives we aim to optimize correspond to those described in Section 6.3.
- The work required to assemble a work-piece is subdivided into elementary operations referred to as tasks, denoted by the set $V = \{1, ..., n\}$. Each task is associated with a processing time, designated as t_j. In the context of NN processing in a pipeline, these elementary tasks correspond to NN layers (a layer cannot

be shared by two NPUs as explained in Subsection 6.2.4). However, the task time t_j in our specific case varies depending on the NPU responsible for processing the task. It is equal to $p_{i,j}$ for a layer L_j processed on the NPU G_i. The *SALBP* is not describing this feature.

- Tasks are subject to precedence constraints, indicating that they must be executed in a particular sequence due to technological or organizational prerequisites. In the context of feed-forward NNs , these constraints align with the inherent order stipulating that a layer $L_j, j \in [1, \ell - 1]$, is processed only if L_{j-1} has been processed in advance.

The set S_k of tasks assigned to a station k constitutes its station load or work content, the cumulated task time $t(S_k) = \sum_{j \in S_k}^{n-1} t_j$ is called station time. It corresponds to the execution time of a machine: $p_{i,[g,h]}$ for an NPU G_i processing the layers $[L_g, L_h]$.

A feasible *Line Balance* must respect the following constraint: all stations have the same cycle time (paced assembly line) c, allowing all stations to start and pass workpieces at the same rate. It means that all stations are synchronized on the slowest one that has a cycle time equal to c. However, if a workstation is completed in less than c, it has an unproductive *idle time*.
Naturally, the pipeline problem that we deal with add more constraints than the one described in the *SALBP*, as the ones highlighted in Subection 6.6.1. The *SALBP* is categorized in different types according to the objective:

- *SALBP-1*: it minimizes the sum of *idle times*. The number of stations is not fixed. It involves maximizing the throughput as it balances the loads.
- *SALBP-2*: it also minimizes the sum of *idle times* for a fixed number of stations.
- *SALBP-E*: the number of stations and the cycle time remain variable and are not fixed. The quality of balance is determined based on the "line efficiency," which is calculated as: $E = \frac{t_{sum}}{m \times c}$, where t_{sum} represents the total operating time of a station (corresponding to Lat_2 in our scenario), and m and c denote the number of workstations and the cycle time, respectively.
- *SALBP-F* consists in finding a feasible balance for a given number of stations and a given cycle time. It corresponds to the problem that we deal with (fixed hardware problem) when optimizing the throughput.

Additional details and assumptions have been introduced to further refine the problem description. This was initially undertaken by Baybars [15] and Scholl and Becker [108], and subsequently expanded upon by Boysen et al. [18]. For instance, new classifications such as "fix" were introduced to account for scenarios where certain tasks must be allocated to specific workstations, aligning with the first three constraints outlined in Subsection 6.2.4. Additionally, the "spec" classification may be added to specify that

the assembly line possesses a particular structure, such as a pipeline in our context. Consequently, the problem can be classified as $SALBP - F, fix, spec$.

As shown by Álvarez-Miranda and Pereira [10], assembly line balancing problems are NP-hard, particularly the *SALBP-1*. However, Held et al. [57] showed that under certain conditions, this problem can be exactly solved using dynamic programming. These simplifying conditions encompass:

- Precedence graph: it must be serial, meaning each job (j) can only be executed after another job (j') is completed. Simultaneous operation of workstations on the same unit are prohibited. This description is similar to the accelerator pipeline.
- Cycle time: it should be constant, which is equivalent to a condition where all NPUs should operate in fewer than a certain number of cycles.
- Execution time and workstations: each job is performed once for each unit, and the execution time is uniform across all workstations. Furthermore, each job can be assigned to any workstation. It is important to note that in the fixed hardware problem, only the first of these three conditions related to execution time is adhered to.

These conditions streamline the problem, enabling a polynomial resolution, primarily because of the serial graph structure, which facilitates the grouping of tasks on a same work station.

Therefore, the prior classification does not totally adhere to the fixed hardware scenario. The aforementioned dynamic program should be tailored to address this issue. It must incorporate additional constraints specific to our problem, such as mapping constraints (in Subsection 6.2.4) and the capacity constraints of the RAMs detailed in Subsection 6.2.5 that limit the flexibility of job-to-workstation assignments. Moreover, the classification and resolution methods do not take into account variable task execution times influenced by the executing NPU, which depends on its parallelization. In terms of objectives, we must also seek solutions that could optimize latency (Lat_2), an aspect that cannot be achieved through pipeline balancing. The co-optimization of throughput and Lat_2 requires also a refinement of the dynamic algorithm to find optimal solutions.

6.6 Optimizing Throughput and Latency Separately

In this section, our objective is to optimize the KPIs of throughput and latency, as described in Section 6.3, separately. We will present two distinct solutions: the first aims to minimize throughput without taking latency Lat_2 into consideration, while the second focuses solely on minimizing latency Lat_2, which pertains to the objective of processing *ifmaps* one by one.

To achieve these goals for both objectives, we will introduce linear models to address the mentioned issues in Subsection 6.6.1. Consequently, solutions can be obtained through the utilization of linear solvers. Furthermore, in Subsection 6.6.2, we will present polynomial-time solutions employing dynamic programming.

6.6.1 Integer Linear Model with Variables in {0,1}

As previously mentioned, the pipelined architecture is fixed, with parameters n, N_i, and k_i for $i \in [0, n-1]$ predefined. The objective is to optimize either latency or throughput. In this subsection, we propose an integer linear model in $\{0, 1\}$ for the fixed hardware scenario to optimize one of the two KPIs. In Subsubsection 6.6.1.1, our focus will be on minimizing latency Lat_2, and we exclude the scenario where *feature maps* (*fmaps*) are processed simultaneously. This specific case will be automatically addressed in Subsubsection 6.6.1.2, which is dedicated to optimizing throughput (equivalent to optimizing Lat_1).

6.6.1.1 Model for Minimizing Lat_2

In this subsection, we formulate an integer linear model with binary $\{0, 1\}$ variables to minimize Lat_2. As discussed in Subsection 6.3.2, the *ifmaps* are processed sequentially as if a single NPU were employed. We introduce the Boolean decision variables $x_{i,j}$ for $(i, j) \in [0, n-1] \times [0, \ell-1]$ defined as $x_{i,j} = 1$ if and only if $\pi(j) = i$, *i.e.* the layer L_j is performed by G_i. Otherwise, $x_{i,j} = 0$. We observe that $\forall j \in [0, \ell-1]$, $\pi(j) = \sum_{i=0}^{n-1} i.x_{i,j}$. The constraints of the problem may be expressed by linear equations. First, there are the mapping constraints presented in 6.2.4:

- L_0 (*resp.* $L_{\ell-1}$) is performed by G_0 (*resp.* $L_{\ell-1}$), thus $x_{0,0} = x_{n-1,\ell-1} = 1$;
- Each layer is processed by exactly one NPU, so $\forall j \in [0, \ell-1]$, $\sum_{i=0}^{n-1} x_{i,j} = 1$;
- π is non-decreasing, then $\forall j \in [0, \ell-2]$, $\pi(j) \leq \pi(j+1)$;
- Each NPU processes at least one layer, then $\forall i \in [0, n-1]$, $\sum_{j=0}^{\ell-1} x_{i,j} > 0$;

Then, there is the SRAMs constraint expressed by Property 2. It can be formulated using the $x_{i,j}$ variables: $\forall j \in [0, \ell-2]$, $s_j + s_{j+1}(1 + \pi(j) - \pi(j+1)) \leq \sum_{i=0}^{n-1} K_i.x_{i,j}$. Let us verify that this constraint is adhering to Property 2: in the scenario where both consecutive layers L_j and L_{j+1} are processed by G_i ($\pi(j) = \pi(j+1) = i$), the condition $s_j + s_{j+1} \leq K_i$ holds true. Moreover, when only L_j is processed by G_i (when $\pi(j) = \pi(j+1) - 1$), the condition $s_j \leq K_i$ is also satisfied.

Using the Boolean variables, the latency Lat_2 (that is the objective) can be then expressed as: $Lat_2(\pi) = \sum_{i=0}^{n-1} \sum_{j=0}^{\ell-1} x_{i,j} p_{i,j}$.

6.6.1.2 Model for Optimizing the Throughput

In this subsubsection, we will optimize the throughput of the pipeline system processing several *ifmaps*. It is done by minimizing the period P, as stated in 6.3.1. It will automatically minimize the latency by considering that the *ifmaps* are processed simultaneously (as highlighted in Subsubsection 6.3.2). The model described in Subsubsection 6.6.1 still remains valid. Only Lat_2 objective is replaced by the throughput maximization that should be expressed linearly using the variables defined in Subsubsection 6.6.1: as seen in Subsubsection 6.3.1, maximizing the throughput is equivalent to minimize P, the objective is then to compute:

$$P = min(\max_{i \in [0,n-1]} (\sum_{j=0}^{\ell-1} x_{i,j} p_{i,j})). \tag{6.2}$$

6.6.2 Dynamic Programs

Since the pipeline problem can be mathematically represented as a linear model, it allows for the derivation of an optimal solution using solvers. Many of these solvers are available as open source tools [45]. However, we have crafted a more efficient approach for finding a mapping π optimizing throughput or latency. This method employs a polynomial-time algorithm based on a dynamic programming scheme to allocate the NN layers within the specified pipeline architecture. Subsubsection 6.6.2.1 details the algorithm that seeks an optimal mapping to minimize Lat_2, while Subsection 6.6.2.2 outlines the algorithm that identifies an optimal mapping to maximize throughput.

6.6.2.1 Dynamic Program Optimizing Lat_2

The algorithm takes as input the parameters of the NN (including layer number, type and parameters, image and filter dimensions) as well as the pipeline architecture details (such as the number of NPUs, their order, the number of PEs per NPU, denoted as N_i for $i \in [0, n-1]$, and the RAM sizes K_i for $i \in [0, n-1]$). The output of the algorithm is an allocation, specifying which NPU processes each NN layer, aiming to minimize latency Lat_2.

Let us consider $M_{i,j}, i \in [0, n-1], j \in [0, \ell-1]$ defined as the latency of the layers $[L_0, L_j]$ on the NPUs $[G_0, G_i]$ assuming that L_j is processed by G_i. Two cases must be considered to compute $M_{i,j}$:

- If L_{j-1} is processed by G_i: G_i is then processing at least L_j and L_{j-i}. The execution time of the layers $[L_0, L_j]$ on G_i is $\gamma_{i,j} = p_{i,j} + M(i, j-1)$. If $s_j + s_{j+1} \leq K_i$, this solution is considered (Property 2 is respected). Otherwise, $\gamma = +\infty$ because the RAM R_i cannot store both I_{j-1} and I_j.

- If L_{j-1} is not processed by G_i, it is subsequently handled by G_{i-1}. The processing time of the layers $[L_0, L_j]$ on G_i is $\delta = p_{i,j} + M(i-1, j-1)$. If $s_j \leq K_i$ this solution is considered (Property 2). Otherwise, $\delta = +\infty$ (R_i is too small to hold I_j).

$M(i,j)$ limits are the following:

- If $s_0 > K_0$, $M(0,0) = +\infty$ (and there is no solution for π), otherwise, $M(0,0) = p_{0,0}$; This is because the first layer is always processed by G_0, as detailed in Subsection 6.2.4;
- For $j \in [0, \ell - 1]$, if $s_j + s_{j+1} \leq R_0$, $M(0,j) = +\infty$, otherwise, $M(0,j) = \gamma_{0,j}$;
- $M(i,j) = +\infty$ if $j = 0$ and $i \in [1, n-1]$ or if $l - j > i > j$. This final condition pertains to situations in which a mapping fails to satisfy the conditions defined for π (as described in 6.2.4). If this condition is not adhered to, it may lead to scenarios in which an NPU processes zero layers or violating the constraint of non-decreasing layer allocation.

$M(i,j)$ is computed for $i \in [1, n-1]$, $j \in [1, \ell - 1]$ with $M(i,j) = min(\delta_{i,j}, \gamma_{i,j})$.

The minimum latency is given by $M(n-1, \ell-1)$ and layers allocations is obtained backtracking the choices done for $M(i,j)$ between $\delta_{i,j}$ and $\gamma_{i,j}$.

Table 6.1 *Table representing dynamic program states optimizing latency without considering RAMs constraints*

	L_0	L_1	L_2	L_3	L_4	L_5
G_0	4815	5914	14847	35748	$+\infty$	$+\infty$
G_1	$+\infty$	5373	9937	20397	20936	$+\infty$
G_2	$+\infty$	$+\infty$	14306	30838	21432	21673

Table 6.2 *Table representing dynamic program states optimizing latency considering RAMs constraints*

	L_0	L_1	L_2	L_3	L_4	L_5
G_0	4815	5914	14847	35748	$+\infty$	$+\infty$
G_1	$+\infty$	$+\infty$	$+\infty$	$+\infty$	36287	$+\infty$
G_2	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	37024

Tables 6.1 and 6.2 represent the states of a system with 3 Gemini NPUs, each equipped with $\{4, 8, 4\}$ PEs, while processing the PNet NN. In this context, ℓ is set to 6. In Table 6.1, each RAM has the capacity to store the entire NN, whereas in Table 6.2, the RAMs have more constrained sizes, specifically configured as $\{16500, 2750, 10000\}$ KBs. In Table 6.1, cells marked as $M(i,j) = +\infty$ correspond to impossible allocations

due to mapping conditions. The calculation of each cell follows dynamic equations to compute $M(i,j)$. For instance, in Table 6.1, $M(2,5)$ is calculated by adding $p_{2,5}$ to the minimum value between $\gamma(2,5)$, that use $M(2,4)$, and $\delta(2,5)$, that use $M(1,4)$. This process is repeated until arriving at $M(i,j)$ values that are known (limits). Once all cells are computed, backtracking is employed to determine the mapping π. In general, this corresponds to finding the shortest path from the bottom-right cell $M(2,5)$ to the top-left cell $M(0,0)$, with movement options to the left (using the same NPU as before) or diagonally left-down (using the NPU placed at the left following Figure 6.4). In Table 6.1, multiple paths are possible. Starting at $M(2,4)$, we observe that L_5 is processed by G_2 due to the first mapping constraint in Subsection 6.2.4. Moving to layer L_4, we note that $M(1,4) < M(2,4)$, indicating that it is processed by G_1. The same reasoning is applied to L_3, where $M(1,3) < M(0,3)$, leading to processing on G_1. This pattern continues for L_2 and L_1, for which it is more efficient to utilize G_1. Finally, L_0 is processed by G_0 according to the mapping constraints. This sequence results in obtaining the optimized mapping $\pi\star$ for minimizing Lat_2. Thus, $Lat_2(\pi\star) = M(2,5)$. The optimal path that yields mapping $\pi\star$ is depicted in green.
Table 6.2 operates similarly. However, the RAM constraints lead to the elimination of several paths, ultimately resulting in only one feasible mapping, represented by the green path.

Theorem 6.6.2. *The complexity of this algorithm is linear, bound by* $\mathcal{O}(n \times \ell)$.

Proof. It corresponds to the number of calculated states. The calculation of each cell is done using Equation 6.1 that have a constant complexity. □

In cases where there are no constraints on RAMs, assuming that all layers can fit into any RAM R_i (where $i \in [0, n-1]$), a heuristic approach can also be employed using more intuitive algorithms. The idea behind this heuristic is to assign the $\ell - n - 1$ possible tasks dictated by π constraints to the fastest NPU and then allocate the remaining tasks to the other NPUs. This can be accomplished with constant time complexity $\mathcal{O}(1)$. However, it is important to note that this method does not yield the optimal solution. Assigning layers to the fastest NPU fixes the scheduling of the other NPUs independently of the layers and NPUs' capabilities. If a critical, highly computational layer is among the assigned tasks, it would have been more efficient to place it on the fastest NPU. This heuristic approach is expected to produce the same mapping as the green cells in Table 6.1. It may be useful primarily for significantly high values of ℓ and/or n.

6.6.2.2 Dynamic Program Maximizing the Throughput

The process of determining the allocation of NN layers that optimizes the throughput when processing a NN on the pipeline architecture can be broken down into two key steps. Firstly, we establish an allocation that guarantees the NN can operate on the pipeline architecture within a specified upper bound of the period denoted as $P^\star$ (the period is defined in Subsection 6.3.1), measured in cycles: if all NPUs in the pipeline process their assigned layers in less than $P^\star$, the throughput will be then higher than $1/P^\star$. The second step consists in employing a binary search method [86] to identify the smallest value of $P^\star$ that fulfills the requirements.

In the initial step (which we refer to as "step 1"), we employ a dynamic programming approach, with the algorithm's input parameters mirroring those outlined in Section 6.6.2.1. Additionally, an extra parameter, the upper bound $P^\star$, is introduced. The algorithm's output is an allocation that ensures the NN's execution occurs within a time frame of less than $P^\star$. Let us consider $M(i,j), i \in [0, n-1], j \in [0, \ell-1]$ defined as the minimal execution time of the NPU G_i assuming that L_j is processed by G_i and all the NPUs $G_{i'}, i' \in [0, i]$ have a processing time $p_{i',\pi} \leq P^\star$. Two cases must be considered to compute $M(i,j)$:

- If L_{j-1} is processed by G_i, the execution time of the layers $[L_0, L_j]$ on G_i is $\gamma_{i,j} = p_{i,j} + M(i, j-1)$. If $\gamma_{i,j} \leq P^\star$ (to respect the constraint that the NPU is processing its assigned layers in less than $P^\star$ cycles) and $s_j + s_{j+1} \leq K_i$ (RAM constraint characterized by Property 2), this solution is considered. Otherwise $\gamma = +\infty$.
- If L_{j-1} is processed by G_{i-1}, the execution time of the layers $[L_0, L_j]$ on G_i is $\delta = p_{i,j}$. If $M(i-1, j-1) \neq +\infty$ (period constraint) and $s_j \leq K_i$ (Property 2) and $p_{i,j} < P^\star$ (period constraint for the layer L_j) this solution is considered. Otherwise, $\delta = +\infty$.

$M(i,j)$ has the following limits:

- $M(0,0) = p_{0,0}$; if $p_{0,0} \leq P^\star$ (Period constraint) or $s_0 \leq K_0$ (RAM constraint) then $M(0,0) = +\infty$;
- For $j \in [0, \ell-1]$, if $s_j + s_{j+1} \leq K_0$ (RAM constraint), $M(0,j) = +\infty$, otherwise, $M(0,j) = \gamma_{0,j}$;
- $M(i,j) = +\infty$ if $j = 0$ and $i \in [0, n-1]$ or if $l - j > i > j$. This last condition correspond to cases where a mapping does not fulfill π conditions (described in Subsection 6.2.4)

$M(i,j)$ is computed for $i \in [1, n-1]$, $j \in [1, \ell-1]$ with $M(i,j) = min(\delta_{i,j}, \gamma_{i,j})$. The mapping π is obtained backtracking the decisions made to compute $M(i,j)$.

Table 6.3 represents the states of the algorithm that identifies an allocation π capable of achieving a period shorter than $P^\star = 14847$ cycles for $n = 3$, on PNet ($\ell = 6$). In this example, Gemini NPUs are used with respectively $\{4, 8, 4\}$ PEs. The RAMs are equipped with a capacity sufficiently large to avoid imposing additional constraints on the allocation process. Once all cells have been computed, with the selection between γ and δ made at each step, the backtracking process can proceed as follows:

- L_5 is assigned to G_2 in accordance with Subsection 6.2.4. It specifies that the last layer is processed by the last NPU.
- For L_4, as $p_{1,4} < 14847$, it is allocated to G_1.
- The same allocation process is followed for L_3.
- In the case of L_2, where $p_{1,2} > 14847$, it is assigned to G_0.
- Finally, L_0 must be executed by G_0 as per the mapping constraints.

In Table 6.3, the green path represents the allocation that optimizes throughput, while the latency Lat_2 for each NPU is indicated in bold characters.

Table 6.3 *Table representing dynamic program states optimizing the throughput without considering RAMs constraints*

	L_0	L_1	L_2	L_3	L_4	L_5
G_0	4815	5914	**14847**	$+\infty$	$+\infty$	$+\infty$
G_1	$+\infty$	558	4564	10460	**10999**	$+\infty$
G_2	$+\infty$	$+\infty$	8933	$+\infty$	1035	**737**

The throughput maximization is obtained by minimizing $P^\star$. It is done performing a binary search on $P^\star$ between $P_{min} = \min_{i\in[0,n-1]} \min_{j\in[0,\ell-1]} p_{i,j}$ and $P_{max} = P_a$ with P_a the execution time of the system using an arbitrary feasible allocation (respecting π and RAM constraints). "Step 1" is reiterated as many times as necessitated by the binary search process. A pseudocode of this binary search (which we refer to as "step 2") is given in Listing 6.1:

Listing 6.1: *Step2 pseudo code*

```
step2_function(P_min,P_a)
    if step1_function(P_min) has a solution:
        return P_min
    delta = (P_a - P_min)
    if delta = 1:
        return P_a
    P_max = P_a
    while delta>1:
```

```
        P*= int(P_min+P_max)/2
        if step1_function(P*) has a solution:
            P_max= P*
        else:
            P_min = P*
        delta = P_max-P_min
    return P_max
```

In practice, the "step1_function" corresponds to the mapping search for a given period, and it is solved by the dynamic algorithm presented in the preceding paragraph. The "Step2_function" begins by testing if a valid mapping is achievable for the P_{min} constraint. If it is not feasible, we check whether the difference between P_a and P_{min} is exactly 1, which represents the desired precision for the optimal period. If this condition is met, P_a is considered the optimal period. Otherwise, we employ the standard binary search algorithm [86], with the stopping condition being that our precision is less than 1. At each step of this binary search, we assess if a valid mapping solution exists for a specific period $P^{\star}$.

The final step of the binary search yields the solution, providing both the optimal period and, consequently, the optimal throughput. As a reminder, the mapping is determined by backtracking the decisions to identify the values of $M(i,j)$ between $\gamma_{i,j}$ and $\delta_{i,j}$ in "step1."

Theorem 6.6.3. *The complexity of the algorithm is bounded by* $\mathcal{O}(n \times \ell(\log P_{\max}))$.

Proof. The algorithm's complexity for "step 1" is bounded by $\mathcal{O}(n \times \ell)$, while "step 2" is constrained by $\mathcal{O}((\log P_{\max}))$. Since the calculations for the cells in "step 1" and the conditions evaluated during the binary search in "step 2" are constants, the overall complexity of the algorithm is the product of the complexities of these two steps. □

As stated in Subsection 6.3.2, this throughput optimization also optimizes the latency Lat_1 that consider a simultaneous processing *ifmaps* (for a streaming application for example).

6.7 Latency and Throughput Co-optimization

In Section 6.6, we demonstrated the potential for optimizing both the latency and throughput of NN execution on a pipeline system by identifying optimal allocations based on each criterion. This approach proves particularly valuable in applications where the same pipeline is designed to support NNs prioritizing either throughput (streams) or latency Lat_2 (punctual fast inferences). Then, the same pipeline can be

adapted to target a specific KPI or another, with only the layer allocation being adjusted.

However, in Section 6.3, we intuited that the allocation solution that optimizes latency may not necessarily be the same as the one that optimizes throughput. This intuition is later confirmed in Section 6.8. In some scenarios, certain requirements may necessitate the optimization of both factors simultaneously. With this perspective in mind, we have developed a method to enhance latency (Lat_2) while simultaneously taking into account a specific throughput requirement, characterized by an upper bound period $P^\star$, which the overall processing should not exceed. The optimization of throughput T is achieved by subsequently reducing the upper bound period $P^\star : P = 1/T \leq P^\star$. It is important to note that the latency Lat_2 assumes that *ifmaps* are processed one by one. If Lat_1 is considered, the solution that optimizes throughput will also optimize Lat_1.

In this section, we will begin by introducing a new linear model that is better suited for co-optimizing Lat_2 and P. Subsequently, we will propose a solution based on dynamic programming to address this problem.

6.7.1 Integer Linear Model with Variables in $\{0, 1\}$ Optimizing Latency Lat_2 for a Specific Period

The models introduced in Section 6.6.1 are no longer applicable because, in this new problem, throughput is as a constraint rather than an objective. For this model, we consider Boolean decision variables $x_{i,[g,h]}$, defined as $x_{i,[g,h]} = 1$ if and only if, for all $j \in [g, h]$, $\pi(j) = i$. In other words, it indicates that layers L_g to L_h are executed by G_i. Otherwise, $x_{i,[g,h]} = 0$. These variables are defined for $i \in [0, n-1]$ and $0 \leq g \leq h \leq \ell - 1$. The constraints for the pipeline structure (as expressed in Subsection 6.2.4), RAMs size (highlighted in Subsection 6.2.5), and the period upper bound constraint are all formulated using linear equations:

- L_0 (*resp.* $L_{\ell-1}$) is performed by G_0 (*resp.* $L_{\ell-1}$), thus:
  $\sum_{g=0}^{\ell-1} x_{0,[0,g]} = \sum_{h=0}^{\ell-1} x_{n-1,[h,\ell-1]} = 1$;
- Each layer is processed by exactly one NPU, so:
  $\forall j \in [0, \ell-1], \sum_{i=0}^{n-1} \sum_{g=0}^{j} \sum_{h=j}^{\ell-1} x_{i,[g,h]} = 1$;
- π is non-decreasing:
  $\forall j \in [0, \ell-2], \sum_{i=0}^{n-1} \sum_{g=0}^{j} \sum_{h=j}^{\ell-2} i.x_{i,[g,h]} \leq \sum_{i=0}^{n-1} \sum_{g=1}^{j+1} \sum_{h=j+1}^{\ell-1} i.x_{i,[g,h]}$
- Each NPU processes at least one layer (it means that at least one subset of $\mathcal{L}$ is processed by each NPU): $\forall i \in [0, n-1], \sum_{j=0}^{l-1} \sum_{g=0}^{j} \sum_{h=j}^{l-1} x_{i,[g,h]} > 0$;
- Each NPU processes its layers in less than $P^\star$ cycles:
  $\forall i \in [0, n-1], \sum_{j=0}^{\ell-1} \sum_{g=0}^{j} \sum_{h=j}^{\ell-1} x_{i,[g,h]} p_{i,[g,h]} \leq P^\star$

- The RAMs constraint originally expressed by Property 2 and particularly Property 1 in Subsection 6.2.5: $\forall j \in [0, \ell - 1]$:

$$\begin{aligned} &\sum_{i=0}^{n-1} \sum_{g=0}^{j} \sum_{h=j+1}^{l-2} x_{i,[g,h]} \max_{k \in [g,h]} .(s_k + s_{k+1}) + x_{i,[j,j]}.s_j \\ &\leq \sum_{i=0}^{n-1} \sum_{g=0}^{j} \sum_{h=j+1}^{l-2} x_{i,[g,h]}.K_i + x_{i,[j,j]}K_i. \end{aligned} \tag{6.3}$$

  Indeed, thanks to Theorem 6.2.1, we are aware that for $g < h$, $\max_{j \in [g,h]}(s_j + s_{j+1})$ represents the minimum feasible size requirement for the RAM storing layers $[L_g, L_h]$, which must be validated. On the other hand, if $j = g = h$ (corresponding to a single layer processed by G_i), we solely need to confirm that RAM R_i has a size K_i greater than s_j.

The latency to be minimized corresponds to the total number of cycles needed for processing one image, thus:
$Lat_2(\pi) = \sum_{i=0}^{n-1} \sum_{j=0}^{\ell-1} \sum_{g=0}^{j} \sum_{h=j}^{\ell-1} x_{i,[g,h]} p_{i,[g,h]}$. This model exhibits certain similarities with the one introduced by Boysen et al. [18] to represent the SABLP-1 problem (as presented in Subsection 6.5.3). However, it incorporates novel constraints, particularly those related to RAMs, and introduces the objective of optimizing Lat_2, which cannot be resolved through load balancing alone.

6.7.2 Dynamic Program Optimizing the Latency for a Given Throughput

We developed a polynomial-time algorithm based on a dynamic programming scheme that provides a solution for optimizing latency Lat_2 under specific throughput constraints, defined by an upper bound on the period, denoted as $P^\star$. To clarify, throughput is determined by $T = 1/P$ with the constraint $P \leq P^\star$. This algorithm is inspired by [57] with more constraints and different objectives. The algorithm inputs are: the period's upper bound $P^\star$, the NN layers $\mathcal{L}$, their parameters, and NPUs $\mathcal{G}$ and RAMs $\mathcal{R}$ parameters. The outputs are the latency Lat_2, the throughput ($T(\pi) = \min_{i \in [0,n-1]} \frac{1}{p_{i,\pi}}$) and the layers allocations.

The main idea of our algorithm is to build a valued directed state graph $\mathcal{H} = (V, E, w)$ defined as follows: the set of vertices is $V = \{s, p\} \cup V_1$ with $V_1 = \{[i, g, h], i \in [0, n-1], 0 \leq g \leq h \leq \ell - 1\}$. Each vertex $u = [i, g, h] \in V_1$ models the successive layers $[L_g, L_h]$ mapped to the NPU G_i. It is important to note that certain layer sets $[L_g, L_h]$ may not possibly mapped to certain G_i if they do not satisfy the conditions outlined in Sections 6.2.4 and 6.2.5. The set of arcs $E = E_s \cup E_p \cup E_1$ is defined as:

- $E_1 = \{a = (u, u') \in V_1^2, u = [i, g, h]$ and $u' = [i+1, h+1, m]$ for $p_{i,[g,h]} \leq P^\star$ and: if $g = h, s_g \leq K_i$ or if $g < h$: $\widehat{K}_{i,[g,h[} \leq K_i\}$. It models that $[g, h]$ can be mapped to NPU G_i and $[h+1, m]$ can be mapped G_{i+1} with $h+1 \leq m \leq \ell - 1$. However, the vertex is built only if the processing of $[L_g, L_h]$ on G_i satisfy the period constraint $P^\star$ and that the RAM R_i is large enough to store the intermediary *fmaps* ;
- $E_s = \{(s, u), u = [0, 0, h] \in V_1\}$ it models that the layers $[L_0, L_h]$ are mapped to G_0;
- $E_p = \{(u, p), u = [n-1, g, \ell - 1] \in V_1\}$ it models that the layers $[L_g, L_{\ell-1}]$ are mapped to G_{n-1}.

The valuation $w : E \mapsto \mathbb{N}$ of the arcs is defined based on the objective φ, which represents latency. It is formulated as follows:

- For each arc $a = (u, p) \in E_p$, $w(a) = 0$;
- Each arc $a = (u, v) \in E_p \cup E_1$ with $u = [i, g, h]$ is valued based on the execution time: $a = p_{i,[g,h]}$

Figure 6.6 displays the graph for 3 NPUs and 5 layers. For example, the path $s \to [0, 0, 0] \to [1, 1, 3] \to [2, 4, 4] \to p$ is associated to the mapping $\pi : [0, 4] \to [0, 2]$ with $\pi(0) = 0$, $\pi(1) = \pi(2) = \pi(3) = 1$ and $\pi(4) = 2$. As the arcs are valued based on the assumption that all π and RAM properties remain valid, and that all $NPUs$ complete their execution within P cycles, the path with the minimum value from s to p represents a feasible mapping that minimizes latency. Consequently, our algorithm constructs the state graph $\mathcal{H}$ and identifies the shortest path within it using the Dijkstra algorithm [31].

Theorem 6.7.4. *The time complexity of the algorithm belongs to* $\mathcal{O}(\ell^4 \log \ell)$.

Proof. The number of vertices (resp. arcs) of the state graph belong to $\mathcal{O}(\ell^2)$ (resp. $\mathcal{O}(\ell^4)$). Moreover, the complexity of computing the value of each arc $(u, u') \in E_1$ is linear. The equivalent complexity is then bounded by to $\mathcal{O}(\ell^4 + \ell \times l^2)$. Now, the complexity of Dijsktra algorithm Cormen et al. [31] is bounded by $\mathcal{O}(\ell^4 \log \ell)$, thus the theorem holds. □

This algorithm is implemented thanks to the Python library Networkx [51]. It can also be represented using states (without using graphs based approach), as shown in Appendix B.

6.8 Applications to Gemini

In this section, we will apply the pipeline structure, as described in Section 6.2, to enhance the performance of the Gemini accelerator, whose design is detailed in Chapter 4.

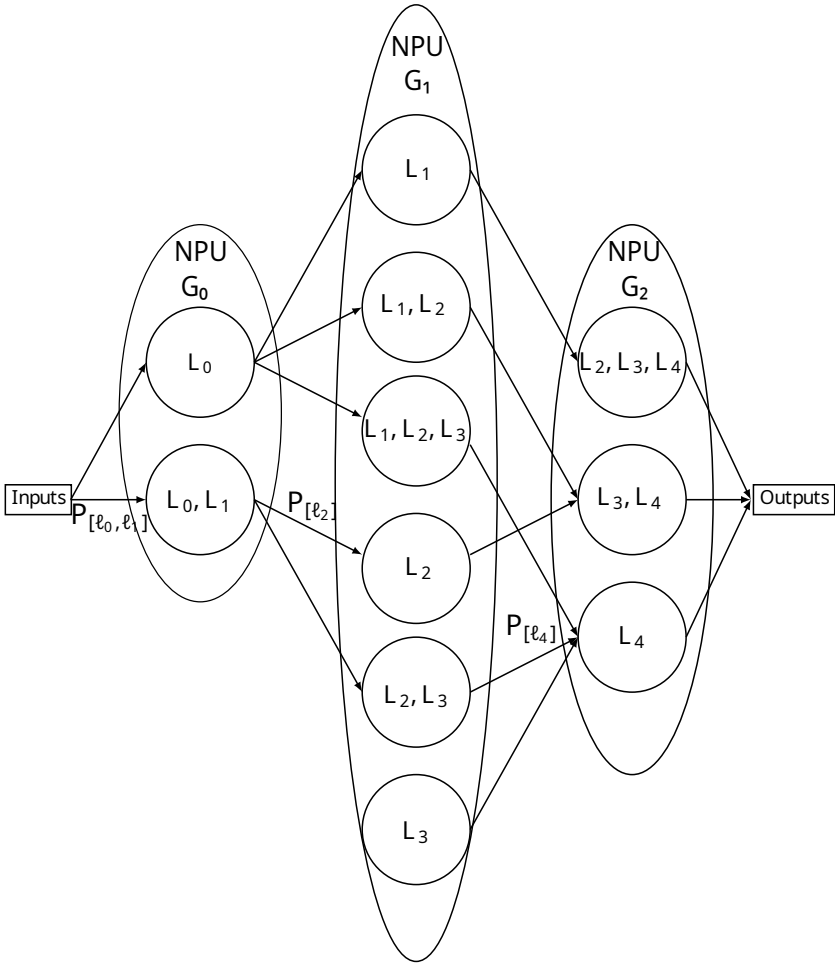

Fig. 6.6 *Graph* $\mathcal{H} = (V, E, w)$ *for* $n = 3$ *and* $\ell = 5$

Our approach will begin by illustrating how multiple instances of Gemini can be interconnected to align with the fixed hardware scenario (as explained in Subsection 6.8.1). Subsequently, we will employ the algorithms to optimize KPIs described in Subsection 6.3. Our optimization will start with the separate optimization of latency Lat_2 and throughput T (characterized by the period P), following the methodologies outlined in Section 6.6. This will be elaborated upon in Subsection 6.8.2. Following this, we will delve into the joint optimization of latency Lat_2 and throughput in Subsection 6.8.3, utilizing the methodology presented in Section 6.7.

6.8.1 Hardware Feasibility on Gemini and Execution Time

As a reminder, when utilizing a single Gemini NPU, a single FMAPS RAM instance with a single port suffices for both reading and writing purposes (as explained in Section 4.2.3). This is due to the fact that NPU execution does not necessitate reading the *fmaps* during every cycle. As highlighted in Section 4.2.5, during reading cycles, the NPU fetches sufficient data to sustain multiple execution cycles, thereby avoiding the need for continuous reading. The cycles during which *fmaps* are not read are utilized for

writing the *ofmaps* into the RAM.
In a pipelined system, the utilization of the same RAM for both reading and writing purposes across two distinct NPUs does not present a concern, as the aforementioned principle can still be applied: consider a NPU G_i ($i \in [0, n-1]$). It reads its initial *fmaps* from RAM R_{i-1}. During a certain number of cycles, the reading operation is not performed by NPU G_i. During this time, NPU G_{i-1} can utilize these cycles to write the subsequent *(feature maps) fmaps* into R_{i-1}. As emphasized in Section 4.3.3, a few exceptions exist where continuous reading can occur for multiple cycles. In such scenario, the operation of NPUs G_{i-1} and G_i is temporarily stopped, resulting in a stall in reading. This pause is just long enough to facilitate the writing of the output *fmap*, preventing any inadvertent overwriting.

However, the architecture of Gemini depends on the $(WPAR, MPAR)$ configuration (as outlined in Section 4.2.1), which introduces complexity when attempting to share RAMs between two NPUs. The geometry of the RAMs is intrinsically tied to the values of $WPAR$ and $MPAR$. As a reminder, the WEIGHTS RAM is composed of $WPAR \times MPAR \times weightbits$ bits and the *fmaps* RAMs are composed of $WPAR$ memory banks, each encompassing $MPAR \times fmapbits$ bits.
Given that the width of the FMAPS RAM instances is determined by the parameter $MPAR$, it can be challenging to dynamically adjust the RAM width to meet the constraints of the two NPUs that share it. Therefore, our solution involves having the same $MPAR$ value for all NPUs. The only differing parameter for these NPUs is $WPAR$. Fortunately, the WEIGHTS RAM does not pose any challenges in this context, as it is not shared among the NPUs.
The parameter $WPAR$ determines the the FMAPS RAM banks number. Implementing distinct NPUs G_i with varying $WPAR$ values demands adjustments, given that an NPU G_i handles $WPAR_i$ memory banks, whereas another NPU G_{i+1} operates with $WPAR_{i+1}$ memory banks.
To address this concern, a solution was implemented by introducing an extra logic block denoted as *L*. This logic block facilitates the translation of addresses and memory banks identifiers (IDs) requested by NPU G_i into addresses and banks IDs that correspond physically to a RAM containing $WPAR_{i+1}$ banks. This unit takes the memory bank ID and the address specified by NPU G_i as input and outputs the appropriate bank ID and address within a system comprising $WPAR_{i+1}$ memory banks. The two equations managed by the logic block *L*, responsible for this conversion, are as follows:

$$Bank_{i+1} = (WPAR_i \times ADDR_i + BANK_i) \mod (WPAR_{i+1}) \tag{6.4}$$

$$ADDR_{i+1} = \lfloor \frac{WPAR_i \times ADDR_i + BANK_i}{WPAR_{i+1}} \rfloor \tag{6.5}$$

For a RAM R_i positioned between NPUs G_{i-1} and G_i, this logic block *L* is inserted in two feasible locations, always selecting the configuration with the larger $WPAR$:

- After the RAM R_i, if $WPAR_i < WPAR_{i-1}$, it means that this block will adapt the *fmaps* read by $WPAR_i$.
- Before the RAM R_i, if $WPAR_i > WPAR_{i-1}$, it means that this block will adapt the *fmaps* written by $WPAR_{i-1}$.

In every scenario, the adaptation of the address and bank IDs is consistently implemented for the NPU with a smaller $WPAR$. Figure 6.7 provides a visual representation of this system for the two scenarios mentioned earlier. As illustrated, the block labeled as *L* is strategically positioned to adapt either the reading operation when $WPAR_i < WPAR_{i-1}$ or the writing operation when the opposite condition holds.

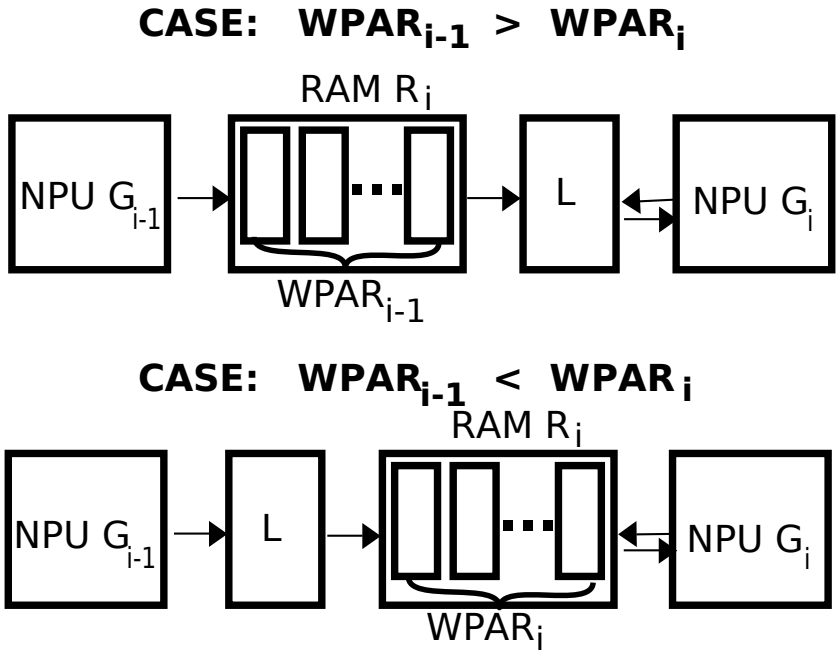

Fig. 6.7 FMAPS *RAMs geometry adaptation*

To sum up, with the utilization of Gemini NPUs, implementing a pipeline is entirely feasible. It necessitates the integration of a logical block to accommodate FMAPS RAMs when NPUs possess different $WPAR$ values. Nevertheless, it is essential that all NPUs share the same $MPAR$. For illustrative purposes, in the forthcoming examples involving Geminis, the number of PEs will exclusively refer to $WPAR$. Appendix C gives an idea on how $MPAR$ can be chosen. The WEIGHTS RAMs, which are not shared among NPUs, do not require any adjustments.

As observed in Subsection 4.2.1, the PEs number can be adjusted before the logic synthesis. The execution time $p_{i,[g,h]}$ on a Gemini NPU G_i of a set of consecutive layers $[L_g, L_h]$ with $(g,h) \in \{[0,\ell-1]^2 g \leq h\}$ is given by Equation 5.2 in Subsection 5.5.1. Since $MPAR$ is constant (as explained in the preceding paragraph) and N_i refers

exclusively to $WPAR_i$: $p_{i,[g,h]} = \sum_{j=g}^{h} f(N_i) + (h-g).Cst$. In this context, Cst represents a constant value that remains independent of the specific layer being processed. The function f is dependent on the number of PEs and the parameters of the layer L_j. These parameters include factors such as the layer type, *fmap* sizes, the number and sizes of filters (for convolution-like layers), stride, padding, and other relevant characteristics. As established in Subsection 5.5.1, for a fixed layer L_j, the function $f(N_i)$ is expressed as follows:

- For convolution-like layers: $f(N_i) = \left\lceil \frac{W(H-(R-1)pad_v)}{N_i} \right\rceil \left\lceil \frac{M}{MPAR} \right\rceil \times K_c$;
- For fully connected layers: $f(N_i) = \left\lceil \frac{Nout}{N_i \times MPAR} \right\rceil N_{in}$.

It is essential to emphasize that $p_{i,[g,h]}$ exhibits a decreasing (monotone) trend as N_i increases, as elaborated upon in Subsection 6.2.6.

6.8.2 Results of Separate Throughput and Latency Optimization on Gemini

In this subsection, Gemini NPUs are used within a pipeline configuration to optimize either the latency Lat_2 or the period P. To illustrate this, we use the PNet neural network architecture depicted in Figure 2.14. The pipeline structure employed here is the same as briefly described in Subsubsections 6.6.2.1 and 6.6.2.2. Specifically, the pipeline consists of three NPUs, each equipped with a different number of processors, namely $\{4, 8, 4\}$ (notably, these values correspond to $WPAR$, as $MPAR$ is fixed at 8). Additionally, the FMAPS RAMs have capacities of $\{16500, 2750, 10000\}$ KBs, respectively.

We start by implementing the algorithm outlined in Subsubsection 6.6.2.1 to determine the mapping that optimize latency Lat_2 (*ifmaps* are processed sequentially), taking into account or disregarding the RAM constraints. This results in two solutions: Solution $S1$ (ignoring RAM constraints) and Solution $S2$ (RAMs capacities considered). Following that, we utilize the algorithm described in 6.6.2.2 to identify the optimal layers' allocation on the NPUs, maximizing throughput (and consequently improving Lat_1). We label this solution as $S3$ when disregarding RAM constraints and $S4$ otherwise. The solutions are presented in Table 6.4. The results clearly show that $S1$ and $S2$ produce different outcomes, highlighting that the same pipeline can be adjusted to prioritize either Lat_2 or P. However, when RAM constraints are considered, fewer mapping options are available, and it is possible to find the same solution given the two dynamic algorithms.

To put it simply, $S1$ can reduce latency Lat_2 by around 40% (best vs. worst mapping) without RAM constraints and approximately 11% with RAM constraints ($S2$).

Solution	Latency (Lat_2) in cycles	Period (P) in cycles	Mapping (π)
$S1$	21673	16121	$\pi(0) = 0, \pi(1) = \pi(2) = \pi(3) = \pi(4) = 1, \pi(5) = 2$
$S2$	27079	14847	$\pi(0) = \pi(1) = \pi(2) = 0, \pi(3) = \pi(4) = 1, \pi(5) = 2$
$S3/S4$	33151	22673	$\pi(0) = \pi(1) = 0, \pi(2) = 1, \pi(3) = \pi(4) = \pi(5) = 2$

Table 6.4 *Solutions optimizing Lat_2 and P separately*

On the other hand, $S3$ can reduce the period by up to 59% across various allocations without RAM constraints and up to 36% with RAM constraints ($S4$).

In summary, these algorithms work well for optimizing one KPI individually, making them suitable for scenarios where the same pipeline is used for streaming applications (favoring throughput) or for single-image inferences where minimizing latency is crucial. However, finding a balance between both criteria in scenarios requiring simultaneous optimization is challenging with these solutions.

6.8.3 Results of Co-optimized Latency and Throughput on Gemini

We initially applied the algorithms described in Subsection 6.7.2 and Appendix B (which are essentially the same algorithm) to the PNet network. The same pipeline architecture, as discussed in Subsections 6.8.2, 6.6.2.1, and 6.6.2.2, was used, utilizing Gemini NPUs. Figure 6.8 illustrates the minimum latency Lat_2 (with the assumption of sequential *ifmap* processing) achieved by the pipeline for a given throughput, represented by the period P: Each point correspond to the minimum latency that we could have for the given P. This figure reflects the solutions previously discussed in Subsection 6.8.2

Unfortunately, utilizing PNet on this configuration does not yield insightful conclusions. The limited number of feasible allocations and the RAM constraints overly restrict the available scenarios. Therefore, we conducted tests using MobileNet x0.25 (presented in Figure 2.13) on an alternative configuration B, featuring larger and less restrictive RAMs (though still eliminating certain allocations). This pipeline architecture comprises five Gemini NPUs, each with a corresponding number of PEs: $\{20, 7, 7, 7, 7\}$. These NPUs are complemented by RAMs with capacities of $\{300000, 120000, 15000, 15000, 15000\}$ KB respectively. The allocation options are limited by the smaller sizes of the last three RAMs. First of all, we observe in Figure 6.9 that obviously the scenario where RAMs sizes are considered has less solutions than the one ignoring them. It is since some layers allocation are no more possible on the last three RAMs. In a broader context, it is observed that throughput and latency exhibit contradictory characteristics. Latency is optimal when there are no stringent constraints on throughput, whereas it

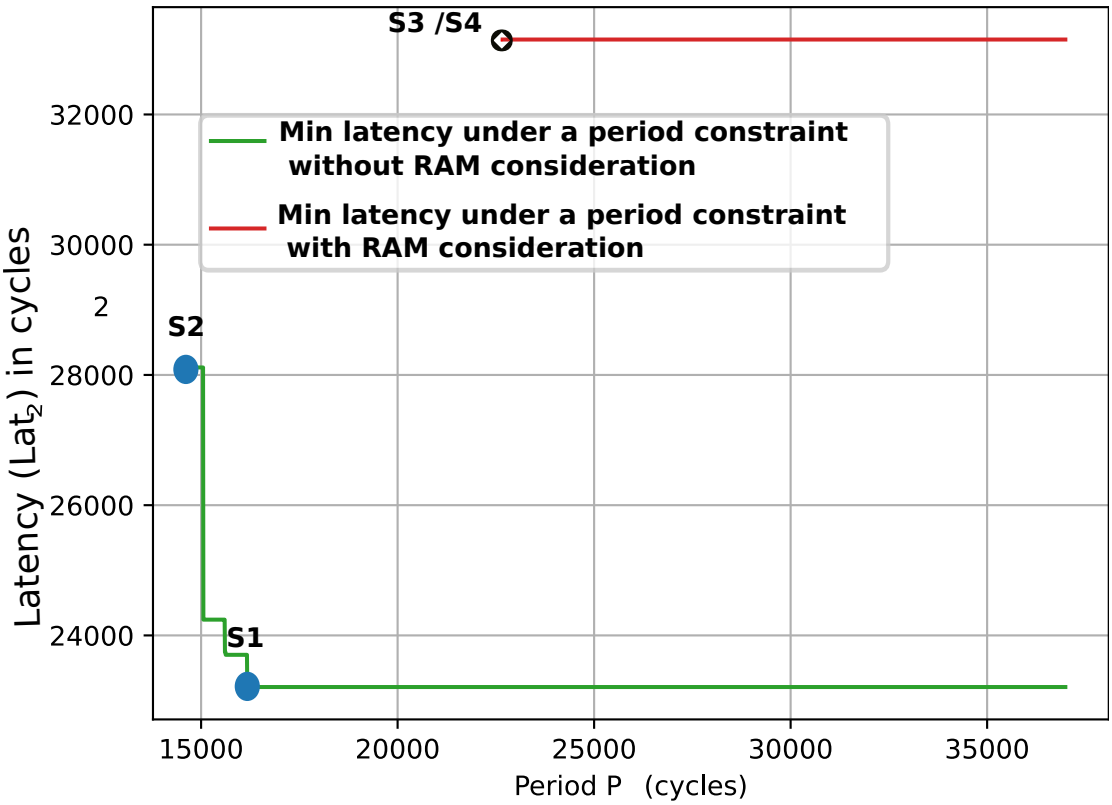

Fig. 6.8 *Latency as function of period constraint, considering PNet on architecture A*

is minimized when strict criteria are imposed on throughput. The staircase-like shape of the curve is a result of the presence of ceiling operators in the execution time equation of Gemini (Equation 5.2). Table 6.5 presents the three optimal solutions depicted in Figure 6.9. The solution S'2 achieves the best throughput by allocating layers in a manner that balances the execution time on each NPU. On the other hand, the solution S'1 achieves the best latency (Lat_2) by maximizing the number of layers on the most powerful NPU (the first one) and minimizing the layers on the less powerful one (as observed in other examples). These two strategies are contradictory, thus explaining the antagonistic relationship between latency and throughput. The solution S'3 is more difficult to interpret because of RAMs constraints however the algorithm seems to balance the execution times also when the allocations are possible.

Table 6.5 *Optimal solutions for throughput and latency for RAMs A scenario*

	NPU 0	NPU 1	NPU 2	NPU 3	NPU 4
S'1 $\mathcal{L}$ allocation	$0 \rightarrow 22$	23	24	25	26
S'1 latencies	258608	2305	29009	67	274
S'2 $\mathcal{L}$ allocation	$0 \rightarrow 9$	$10 \rightarrow 12$	$13 \rightarrow 16$	$17 \rightarrow 20$	$21 \rightarrow 26$
S'2 latencies	113852	91979	124792	124792	94037
S'3 $\mathcal{L}$ allocation	$0 \rightarrow 16$	$17 \rightarrow 22$	23	24	$25 \rightarrow 26$
S'3 latencies	191516	187186	2305	29009	337

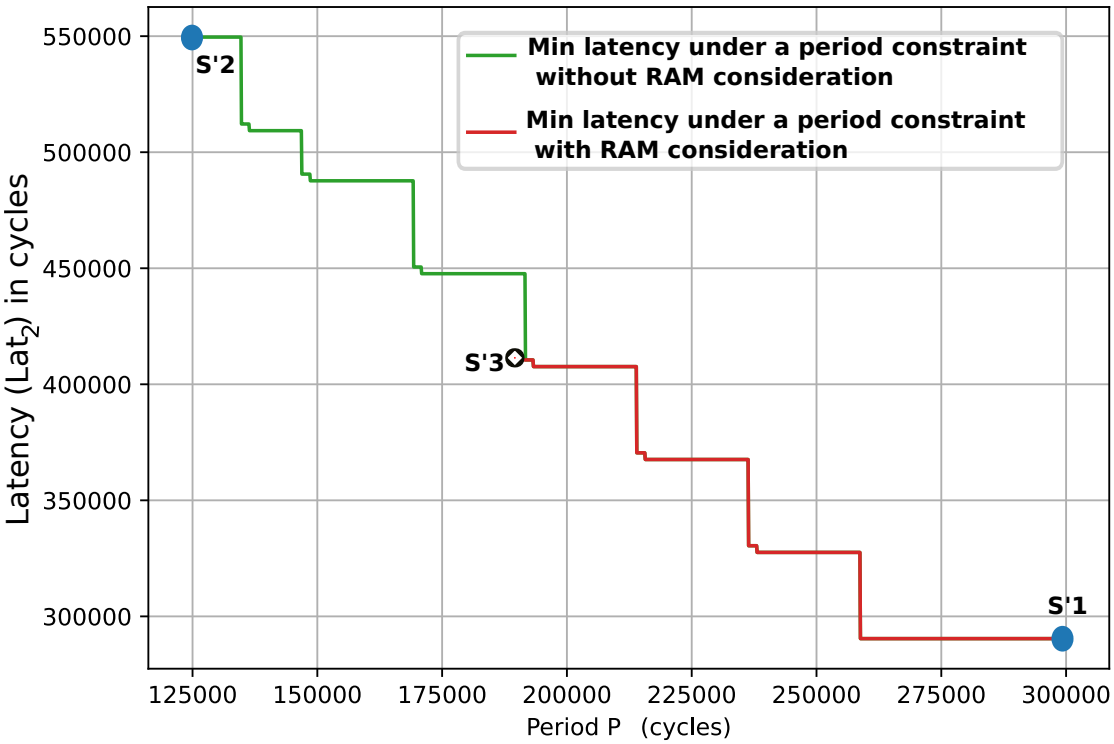

Fig. 6.9 *Latency as function of period constraint considering MobileNet on architecture B*

To sum up, the algorithm outlined in Subsection 6.7.2 provides an effective method to determine the optimal balance between latency and throughput for a predefined architecture of pipelined NPUs, tailored to a specific neural network. This approach accounts for RAM constraints and allows us to target to applications requiring fast punctual inferences while still meeting desired throughput targets (for streams).

6.9 Conclusion

This chapter introduces the concept of pipelining multiple accelerators to process neural networks more efficiently. We explore this idea after realizing that using a single accelerator reaches a limit where further increasing parallelization becomes ineffective. Additionally, this approach is suboptimal because the level of parallelization suits some layers but not others, leading to limited throughput as images are processed sequentially. Given these considerations, and in light of the serial structure of feed-forward neural networks, we opt to employ several instances of NPUs in a pipeline. The chapter outlines this approach within a fixed pipeline scenario, where the number and order of accelerators, along with their various architectures and RAM sizes, are predetermined. The only variable is the mapping of layers to the accelerators. Nevertheless, the mapping is constrained by the pipeline structure and the size of the RAMs. This pipelining strategy is employed to optimize either throughput or latency. Latency optimization is applicable in cases where images are processed in a streaming fashion, and in such

cases, it is indirectly optimized with throughput. Conversely, latency can also be defined for individual inferences when images are processed sequentially, where latency and throughput are in competition.

Throughout the chapter, we review relevant literature on similar problems, encompassing the mapping of general tasks to fixed machines and the specific mapping of neural networks to machines. We also examine related literature in different contexts over the last century. We classify the pipeline optimization problem as reassembling SALBP-1 and provide an integer linear model to formalize the pipeline. We propose dynamic programs that polynomially optimize throughput or latency independently when they are not antagonistic or jointly when they are. This optimization is based on the availability of closed formulas for evaluating accelerators performances, as detailed in Chapter 5. The chapter concludes by illustrating these algorithms for Gemini, for which the study is significant for adjusting the layer mapping based on the target objective. This study is entirely generic and can be tested on other accelerators and other NNs to confirm its practical efficiency.

We observe that this pipelining method is efficient for optimizing throughput as it allows for simultaneous image processing. Additionally, by distributing layers across the best NPUs, processing elements within NPUs are better utilized. In reality, the potential of this pipeline could be further enhanced if we could adapt the architecture of each accelerator to the layers being processed. The ability to tailor the architecture of each accelerator to the specific layers being used is a promising idea explored in Chapter 7, where the pipeline is not fixed, and designers can modify the number of accelerators, their parallelization, and RAM sizes. This flexibility enables the selection of suitable hardware, further optimizing the performance of NN processing.

CHAPTER 7

Non-Fixed Pipelined Neural Network Accelerators

In this chapter, we delve into an extended version of the challenge presented in Chapter 6, known as the non-fixed hardware scenario. This variation is particularly oriented toward hardware designers, and it involves pipelining the layers of a Neural Network (NN) across multiple Neural Processing Units (NPUs), similar to what was discussed in Section 6.2. However, in this case, the number of NPUs (n), their parallelization configuration ($N_i, i \in [0, n-1]$), and the RAM storage ($K_i, i \in [0, n-1]$) are not predefined or fixed. Note that all the NPUs along with their RAMs are placed on the same chip. The objective remains the optimization of the key performance indicators (KPIs) of the NN. The pipeline was primarily designed to optimize throughput more efficiently than using a single NPU. Therefore, our primary focus in this chapter is on throughput characterized by the period P (as detailed in Subsection 6.3.1). The goal is to determine the optimal pipeline architecture that processes the NN within a specified throughput target while optimizing a specific criterion, such as latency, chip area, power consumption, energy efficiency, and more. The layers' allocation on NPUs has also to be determined. This will also provide an opportunity to evaluate the relevance of the pipeline-based solution compared to a single NPU solution with an equivalent number of PEs.

As previously mentioned in the preceding chapter, the idea of the pipeline is intuited from observing the feedforward neural network structure. The notion behind the pipeline is to allocate layers or groups of layers on NPUs based on their specific requirements, rather than employing a one-size-fits-all approach for the entire neural network. This approach enhances the efficiency of processing elements within the NPUs (Section 6.1).

In this chapter, we present a methodology to solve the problem for any accelerator that adheres to the description provided in Section 6.2. This methodology facilitates the determination of the optimal architecture (number of NPUs, number of PEs per NPU, size of RAMs) and the mapping π (Subsection 6.2.4) of the NN layers to NPUs based on a given throughput requirement characterized by an upper bound of the period. This paradigm is illustrated using Gemini accelerators, for which closed-formulas were

derived in Chapter 5.

Similar to the preceding chapter, the objective is not to conduct direct comparisons of the KPIs in the pipeline implementation against other optimized accelerators. Instead, our focus remains on presenting a high-level optimization methodology applicable to a broad range of configurable accelerators (already optimized or not).

It is important to point out that the methodology presented in this chapter is complementary to the non-fixed hardware problem discussed in Chapter 6. Thanks to the methodology of the non-fixed hardware scenario, we can design a pipeline and deploy a mapping that optimizes a KPI, such as area, for example. After the design step is completed and the chip is ready, this same pipeline can be retargeted for other NNs with different applications. In this context, it can be treated as a fixed hardware problem, and a new mapping can be targeted to optimize one of the criteria presented in the previous chapter in Section 6.3, either for the same NN or for a new one.

To begin, in Section 7.1 we reference how this problem can be classified as a Simple Assembly Line Balancing Problem and mention relevant literature that addresses related topics. Following that, in Section 7.2, we introduce essential lower bounds necessary for optimization. In Section 7.3, we introduce the new objectives that become relevant when the hardware is not fixed, allowing for the optimization of new multiple key performance indicators (KPIs). Then, in Section 7.4, we provide a formal description of the problem. In Section 7.5, we formulate the problem using a Boolean linear model, enabling the utilization of solvers to obtain optimal solutions. Additionally, we have developed a polynomial-time algorithm based on a dynamic programming scheme in Section 7.6, which provides a solution to the problem. For improved readability, we represent the dynamic algorithm as a shortest path problem in a directed graph. In Section 7.7, we utilize the Gemini accelerator to illustrate the effectiveness of the pipeline strategy. In comparison to a solution based on a single NPU, it not only maximizes throughput beyond what a single NPU can achieve but also offers optimization possibilities for KPIs including latency, the number of processing elements, area, power, energy, and more. This is particularly advantageous in terms of power efficiency, as it enables operation at lower frequencies. In Section 7.8, we explore potential extensions of this problem. A conclusion is finally given in Section 7.9.

7.1 Description of the Non-fixed Hardware Scenario and Literature Review

In this chapter, the hardware is no more fixed: thus, for a given throughput, we can optimize various hardware KPIs φ that are impacted by the allocation.

Considering the Simple Assembly Line Balancing Problem (SALBP) classification

introduced by Boysen et al. [18], the non-fixed hardware problem is classified as a particular case of *SALBP-1*. It has been proved to be polynomially solved using a specific dynamic programming algorithm inspired by Held et al. [57] under certain conditions (detailed in 6.5.3). However, the problem that we are interested in has more objectives than only minimizing the sum of *idletimes*. The objective is, for a given throughput, to find the optimal balancing that optimizes another KPI that is dependent on the *line balance*. Thus, the dynamic algorithm has to be adapted for this purpose.

Many researchers and designers are actively exploring similar problems where the hardware is not fixed. For instance, they pipeline ASICs or FPGAs to enhance NNs processing performance. Actually, compared to using existing machines, being at the accelerator's design level allows for adapting the hardware architecture to the application, optimizing both the architecture and the execution of the considered application. As examples, various approaches have been explored for implementing FPGA-based pipelines between layers [127, 11, 109, 132]. Additionally, NeuroSim V1 leverages In-Memory Computing accelerators, with each layer assigned to a specific tile [111]. These instances focus on optimizing resources on a layer-by-layer basis. However, significant enhancements can be achieved by distributing computations across grouped layers, rather than isolating them. This approach can be particularly challenging when dealing with layers that have significantly different computational workloads.

Our approach for optimizing the pipeline architecture and NN layers allocation (which layers are processed by which NPU) involves efficient layer grouping into sets. Recently, Cai et al. [22] developed Autoseg, a new approach that optimizes KPIs by grouping layers onto different configurable NPUs. They demonstrate experimentally that it provides a speedup of $1.2\times$ to $6.3\times$ compared to benchmarked ASICs. However, the optimization problem is modelled and solved using a Mixed Integer Linear Programming, and they developed a heuristic to get an efficient method. In this chapter, we show that this problem can be solved exactly using a polynomial time efficient algorithm.

7.2 Lower Bounds to Respect Allocations Constraints

As detailed in Subsections 6.2.6 and 6.2.5, certain features of NPUs and RAMs are crucial for the optimization process. Specifically, knowledge of the execution time required for a specific set of NN layers on a given processor and an estimation of the RAM size needed to store all intermediate images for a set of NN layers are essential. These considerations were previously discussed in the context of the fixed hardware scenario in the previous chapter.

However, the non-fixed hardware scenario introduces new requirements. These in-

clude determining the minimum number of PEs within an NPU needed to process a set of NN layers in less than a specified number of cycles. Additionally, it necessitates establishing a lower bound on the minimum RAM size required to store intermediate *feature maps* for a given set of layers. These new features are essential for addressing the hardware variability in the non-fixed scenario and ensuring optimal hardware design. In Subsection 7.2.1, we discuss how to compute the minimum number of PEs required for a given execution time, while in Subsection 7.2.2, we present a lower bound for RAM size that is better suited to the non-fixed hardware scenario.

7.2.1 Lower bound on the Number of NPU PEs Required to Execute a NN within a Given Execution Time Constraint

An important aspect is the capability to determine the minimum number PEs required to process a layer set $[L_g, L_h]$ within a specified time frame of Ts cycles. We introduce $\widehat{N}([g,h], Ts)$ as the minimum number of PEs needed to process the layers within the range $[L_g, L_h]$ in less than Ts cycles. This is achieved by finding N_i in Equation 6.1 while setting $p_{i,[g,h]} = Ts$. When we set $g = 0$ and $h = \ell - 1$, we can determine the minimum number of PEs required to process the entire NN within Ts cycles, utilizing only one NPU.

Likewise, we can introduce the notation $\widehat{N}_i(\pi, Ts)$, which denotes the minimum number of PEs required to process the consecutive layers mapped to G_i within a timeframe of Ts cycles. This notation is particularly useful when working with mappings π that specify which layers are processed on each NPU.

Theorem 7.2.5. *Let us consider that for $i \in [0, n-1]$, the layers L_j mapped to G_i follow $q \leq j \leq h$. Let also suppose that a maximum time frame Ts is fixed. Then, the minimum number of PEs of G_i, denoted by $\widehat{N}_i(\pi, Ts)$, can be computed in time complexity $\mathcal{O}(\log N_{\max})$ where $N_{\max}$ is an upper bound of the number of PEs.*

Proof. Since y is a monotone function of N_i, $\widehat{N}_i(\pi, Ts)$ can be computed by simply using a binary search [31] on N_i. □

Note that this complexity is an upper bound, for some architectures, $\widehat{N}_i(\pi, Ts)$ can be computed with linear complexity.

7.2.2 Min RAMs Capacity for an Allocation

In the subsequent discussion, we demonstrate that it is possible to establish a minimum size $\widehat{K}_{([g,h])}$ for the intermediate memories R_i that stores the layers $[L_g, L_h]$ while adhering to Property 2. In a broader context, $\widehat{K}_i(\pi)$, where $i \in [0, n-1]$, denotes the

minimum size necessary for the intermediate memories R_i to accommodate the layers where $\pi(j) = i$ for $j \in [0, \ell - 1]$. We can state then the following theorem::

Theorem 7.2.6. *Let us consider a layers mapping π and $i \in [0, n-1]$. Let us also consider the value $\bar{j}$ (resp. $\underline{j}$) as the maximum (resp. minimum) value $j \in [1, \ell-1]$ such that $\pi(j-1) = i$. Then, $\widehat{K}_i(\pi) = \max(s_{\underline{j}}, \max_{j \in [\underline{j}, \bar{j}-1]}(s_j + s_{j+1}))$ is the minimum feasible size of the SRAM R_i.*

Proof. According to Property 1, all the layers L_{j-1} for $j \in [\underline{j}, \bar{j}]$ are performed by G_i and their output *fmap* I_j are stored by R_i.

Now, the output $I_{\underline{j}}$ is first solely stored by R_i. Then, for $j \in [\underline{j}, \bar{j} - 1]$ the *fmap* I_j and I_{j+1} need to be stored in R_i simultaneously to evaluate the layer L_j. Thus, the lower bound on the memory size of R_i is proved. The introduced notations will be particularly useful when describing the KPIs of an NPU and finding solutions to the non-fixed hardware problem. □

7.3 Objective Functions Considered (or KPIs)

The objective of this chapter is to find an NPU pipeline architecture, along with the allocation of the NN layers on the accelerators, to optimize a KPI for a given throughput t expressed in period P. This is achieved by identifying an architecture capable of processing the NN within a period constraint denoted as $P^\star$ (upper bound of the period). In this chapter, the throughput, as discussed in Section 6.3.1, transitions from being an objective to becoming a constraint.

Several objectives φ can be taken into consideration to determine π and the associated pipeline architecture. $\varphi_i(\pi)$ denotes the objective constrained to the layers allocated to NPU G_i. The only requirements are that φ_i can be evaluated (either through a closed formula or a polynomial-time algorithm), that it demonstrates monotonicity with respect to the number of PEs, and that it is aggregable: $\varphi(\pi) = \sum_{i=0}^{n-1} \varphi_i(\pi)$. Thus, the objective $\varphi(\pi)$ encompasses latency as presented in Subsection 6.3.2, as well as hardware design KPIs that were not possible to optimize in the previous chapter where the hardware was fixed. Several hardware KPIs can be then considered:

- A first KPI to consider is the **total PEs number** $\widehat{N}$ that is the sum of the PEs of each NPU: $\widehat{N} = \sum_{i=0}^{n-1} \widehat{N}_i(\pi)$. Implicitly, this KPI also characterizes the efficiency of the PEs: for example, completing execution under throughput constraints with a minimum number implies that the PEs are being used efficiently (adding more PEs can lead to a situation where some of them may not be used to their full capacity).
- The **area** of the pipeline system is the sum of the area of each NPU G_i with the corresponding SRAM. For $i \in [1, n-1]$, they depend on the size $\widehat{K}_i(\pi)$

of the SRAM R_i and the number of PEs $\widehat{N}_i(\pi)$ of G_i. We thus note $a(\pi) = a_{NPU}(\pi) + a_{RAM}(\pi)$.

- Another commonly considered objective is the NPUs **power consumption** $pw(\pi)$; it is computed summing up the static and dynamic power of each NPU in the pipeline system. The static power depends only on the PEs number, whereas the dynamic power varies also according to the layers being executed. The NPUs' power can be aggregated as NPUs operate simultaneously.
- Lastly, the **energy** consumed by frame is aggregable and computed by summing the product of latency and power: considering that *fmaps* are processed simultaneously (Lat_1), the energy is computed by: $e(\pi) = P.pw(\pi)$. If they are processed sequentially (Lat_2): $e(\pi) = \sum_{i=0}^{n-1} p_{i,\pi}.pw(\pi)$.
- Other types of objectives could be considered to jointly optimize multiple KPIs. Actually, a solution that is optimal for one KPI might not necessarily be the best for another. To find a solution that optimizes both aspects, one could optimize a **convex linear combination** of these KPIs. If there are m KPIs denoted as $\phi^i(\pi), i \in [0, m-1]$ to be optimized, we can start by normalizing all of them between 0 and 1, denoted as $N\phi^i(\pi)$ (N refers to normalized). The function to be optimized is then $\varphi(\pi) = \sum_{i=0}^{m-1} \lambda_i \times N\varphi^i(\pi)$ with $\sum_{i=0}^{m-1} \lambda_i = 1$. It allows us to combine multiple criteria and strike a balance between them.

7.4 Formal Description of the Problem

The flexibility of the hardware involves having an extended characterization of a problem instance compared to the one presented in Section 6.4. An instance of our problem is then defined by a fixed NN with the number of layers ℓ and the size of the intermediary *fmaps* ($s_j, j \in [0, \ell - 1]$), the description of the configurable NPU engine, an upper bound of the period $P^\star$ that the processing period should not exceed and an objective function φ to minimize. The problem consists then to compute a mapping π that minimizes φ. As stated by Theorems 7.2.5 and 7.2.6, the pipeline architecture and layers mapping is deduced from π.

The described pipeline approach is versatile and can be applied to any accelerator that sequentially processes NN layers, regardless of the hardware architecture and operator scheduling. With this consideration, we can optimize objectives. The optimization objective can be diverse, with the only prerequisites being that their evaluation and aggregability are feasible.

7.5 Integer Linear Model with Variables in $\{0,1\}$ to Minimize φ while Adhering to a Throughput Constraint $P^\star$

Firstly, we tried to start from the preceding model formulated in Subsection 6.7.1 and generalize it for other KPIs. As the architecture is no more fixed, this naive model will not be linear anymore. For example, the latency equation will not be linear if the N_i are no more fixed.

In order to optimize $\varphi(\pi)$ while executing a NN of ℓ layers in less than $P^\star$ cycles, we consider the Boolean variables $x_{g,h} \in \{0,1\}$ with $(g,h) \in [0,\ell-1]^2, g \leq h$. $x_{g,h} = 1$ if there is one NPU that processes only the layers $[L_g, L_h]$, and $x_{g,h} = 0$ otherwise. If $x_{g,h} = 1$, the NPU will possess $\widehat{N}([g,h])$ PEs which constitutes the minimal number of PEs required to process these layers as stated in Subsection 7.2.1. Additionally, the RAM will encompass $\widehat{K}([g,h])$ KB, representing the minimum required size for storing these layers as indicated in Subsection 6.2.5. The system will execute then the NN in less than $P^\star$ cycles, since all NPUs are respecting this condition by definition.
The linear constraints are the following: $\forall j \in [0,\ell-1], \sum_{g=0}^{j}\sum_{h=j}^{l-1} x_{[g,h]} = 1$. By construction all the applicable constraints in Subsection 6.2.4 and 6.2.5 are satisfied:

- The layers L_0 and $L_{\ell-1}$ are inherently processed by G_0 and G_{n-1}, respectively, as the NPU handling the layers $[L_0, L_h]$ is referred to as G_0, and the one executing $[L_{h'}, L_{\ell-1}]$ is designated as G_{n-1} (with h and h' in $[1,\ell-1]\times[0,\ell-2]$). Furthermore, it is important to note that, by definition, the mapping cannot be decreasing, as $x_{[g,h]}$ is defined for $g \leq h$.
- Each NPU process at least one layer per definition, as each set of layers is assigned to one NPU per definition.
- Each layer is processed by exactly one NPU: if there were at least two NPUs processing the same layer L_j, it would imply the existence of two layer sets: $x_{[g,h]} = x_{[k,l]} = 1$ with $j \in [g,h]$ and $j \in [k,l]$. Consequently, the sum in the constraint would exceed 1 for j.
- Finally, the throughput constraint and RAMs capacity constraints are, by definition, respected, as the number of PEs of each NPU ($\widehat{N}([g,h],P^\star)$ and the size of RAMs $\widehat{K}([g,h])$ are chosen in such a way as to respect these constraints.

The objective is to minimize $\varphi(\pi)$. Let us introduce $\widehat{\varphi}_{[g,h]}(\pi)$ the restriction of φ to the layers $[L_g, L_h]$ computed on the NPU $\widehat{N}([g,h],P^\star)$: since φ is monotonous with respect to the number of PEs, $\widehat{\varphi}_{[g,h]}(\pi)$ is the minimal φ of an NPU exclusively processing $[L_g, L_h]$. $\varphi(\pi)$ can be then expressed using the boolean variables: $\varphi(\pi) = \sum_{j=0}^{l-1}\sum_{g=0}^{j}\sum_{h=j}^{l-1} x_{g,h} \times \widehat{\varphi}_{[g,h]}(\pi)$.

7.6 Description of the Dynamic Programming Algorithm Minimizing φ while Adhering to a Throughput Constraint $P^\star$

Similar to what was mentioned in the previous chapter, integer linear modeling is sufficient to provide heuristics, but we prefer to use a polynomial-time algorithm to exactly and quickly solve the non-fixed hardware scenario problem. The inputs of the algorithm are: $P^\star$, the period constraint characterizing the throughput, the NN layers $\mathcal{L}$ and their parameters. The output is the pipeline hardware architecture (n, N_i and $K_i, \forall i \in [0, n-1]$), and the layers allocations π optimizing φ while executing the NN in less than $P^\star$ cycles. This intuitive dynamic programming algorithm is inspired from [57] as mentioned in 7.1.

The main idea of our algorithm is to build a valued directed state graph $\mathcal{H} = (V, E, w)$ defined as follows: the set of vertices is $V = \{s, p\} \cup V_1$ with $V_1 = \{[g, h], 0 \leq g \leq h \leq \ell - 1\}$. Each vertex $u = [g, h] \in V_1$ models the successive layers $[L_g, L_h]$ mapped to a same NPU. The set of arcs $E = E_s \cup E_p \cup E_1$ is defined as:

- $E_1 = \{a = (u, u') \in V_1^2, u = [g, h]$ and $u' = [h + 1, m]\}$ models that u' can be mapped to an NPU just after u;
- $E_s = \{(s, u), u = [0, h] \in V_1\}$;
- $E_p = \{(u, p), u = [g, \ell - 1] \in V_1\}$.

Lastly, the valuation $w : E \mapsto \mathbb{N}$ of the arcs is defined following the objective φ as follows:

- For each arc $a = (u, p) \in E_p$, $w(a) = 0$;
- Each arc $a = (u, v) \in E_p \cup E_1$ with $u = [g, h]$ is valued by the restriction of φ to the layers $[L_g, L_h]$ assuming that these layers are mapped to a same NPU and that this NPU computes the layers $[L_g, L_h]$ in less than $P^\star$. The number of PEs of this NPU will be $\widehat{N}([g, h], P^\star)$ to fulfill this last condition, and the RAM size should be $\widehat{K}([g, h])$ to adhere to the RAMs constraints.

Figure 7.1 presents the state graph $\mathcal{H}$ for $\ell = 4$ without the valuation of the arcs. One can observe that paths of the state graph $\mathcal{H}$ from s to p model all the feasible mapping π. For our example, the path $s \to [0, 1] \to [2, 2] \to [3, 3] \to p$ is associated to the mapping $\pi : [0, 3] \to [0, 2]$ with $\pi(0) = \pi(1) = 0$, $\pi(2) = 1$ and $\pi(3) = 2$. The minimum number of PE's for each NPU and the minimum memory size are evaluated using Theorems 7.2.5 and 7.2.6. Since the arcs are valued from the criteria restriction on each NPU, the path of minimum value from s to p models a feasible mapping minimizing φ. Thus, our algorithm simply builds the state graph $\mathcal{H}$ and finds

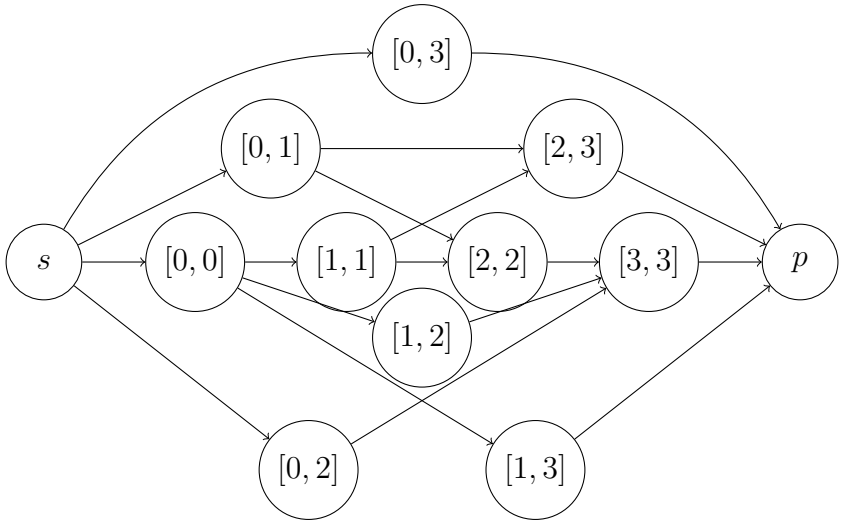

Fig. 7.1 *A state graph $\mathcal{H}$ for $\ell = 4$. The valuations are not presented.*

its shortest path using Dijsktra algorithm [31]. The implementation of the graph was done using Networkx [51].

Theorem 7.6.7. *The time complexity of the dynamic programming algorithm belongs to $\mathcal{O}(\ell^4 \log \ell + \ell^2 \log N_{max})$ where N_{max} is an upper bound on the number of PEs.*

Proof. The number of vertices (resp. arcs) of the state graph belong to $\mathcal{O}(\ell^2)$ (resp. $\mathcal{O}(\ell^4)$). Moreover, for each vertex $u \in V_1$, determining the minimum number of PEs required to reach a given throughput takes $\mathcal{O}(\log N_{max})$ instructions (see Theorem 7.2.5), while determining the minimum memory size requires $\mathcal{O}(\ell)$ instructions (see Theorem 7.2.6). So, the computation of $\mathcal{H}$ belongs to $\mathcal{O}(\ell^4 + \ell^2(\ell + \log N_{max}))$, which is equivalent to $\mathcal{O}(\ell^4 + \ell^2 \log N_{max})$. Now, the complexity of Dijsktra algorithm [31] is bounded by $\mathcal{O}(\ell^4 \log \ell)$, thus the theorem holds. □

This algorithm could generate solutions based on a single NPU, as the path exists within $\mathcal{H} = (V, E, w)$. Finally, a state version of this algorithm is presented in Appendix D.

7.7 Applications on Gemini

This section aims to present our experiments for the non-fixed hardware scenario. Gemini for which analytical close formulas for execution times and KPIs are available (Chapter 5) is used. As a reminder, Gemini is designed for edge applications. It is tailored for NNs with low RAM requirements. Our experiments do not serve demonstrations, they highlight the practicality of our generic approach. Importantly, while Gemini is limited to moderate size NNs, there should be no expected issues when using larger ones with other accelerators, as our allocation algorithm (Subsection 7.6)

maintains polynomial complexity. The aim of this section is to demonstrate the relevance of this methodology for optimizing KPIs for a given throughput constraint. To achieve this, we compare the solutions provided by the pipeline architecture with those achievable by a single NPE: we calculate the KPIs for a single NPU configured with the minimum number of PEs required to meet the throughput constraint.

Subsection 7.7.1 discusses the adaptation of Gemini features to the non-fixed hardware scenario, explaining how the lower bound of NPEs is calculated and how KPIs are tailored to this problem. In Subsection 7.7.2, the pipeline architecture's ability to significantly reduce the period while increasing latency is demonstrated. Subsection 7.7.3 proves that the pipeline architecture provides flexibility in adjusting the total number of PEs. Subsequently, Subsections 7.7.4, 7.7.5, and 7.7.6 delve into area, power, and energy minimization, respectively. These sections illustrate that, even when considering SRAM constraints, the pipeline architecture outperforms single NPE configurations, especially for small period values. Finally, Subsection 7.7.7 outlines how to find a mapping that targets the optimization of multiple KPIs simultaneously. In this section, we utilize the three neural networks described in Subsection 2.3.2. When considering RAMs for some optimizations, we prefer to use VGG-like as an illustration. PNet has too few weights to significantly impact the optimization, while MobileNet, on the other hand, requires extensive RAM support, which obscures several aspects.

7.7.1 Gemini Features for Non-fixed Hardware Scenario

As aforementioned in Subsection 6.8.1, in the case of Gemini instances, the PEs number N exclusively refers to $WPAR$, while $MPAR$ remains fixed. As mentioned in Subsection 7.2, the ability to calculate $\widehat{N}([g,h],Ts)$ is crucial. It is achieved thanks to a binary search [86] between a lower bound N_{min} and an upper bound N_{max} of $p_{i,[g,h]}$ (it is possible because $p_{i,[g,h]}$ is monotonous). The bounds are given by the property of the ceiling function:$\forall x \in \mathbb{N} - \{0\}$, $x < \lceil x \rceil \leq x + 1$. $\widehat{N}$ is then found with a binary search between $N_{min} = \frac{\sum_{j=g}^{h} f(L_j)}{Ts-(h-g).Cst}$ and $N_{max} = \frac{\sum_{j=g}^{h} f(L_j)}{Ts-(h-g).Cst-1}$.
Figure 7.2 represents the latency $p_{i,[0,7]}$ of PNet network (see Figure 2.14) on a Gemini G_i as a function of N_i as well as the lower and the upper bound of p_i. For example, finding $\widehat{N}([0,7],4000)$ is done by binary search between the $Nmin([0,7],4000)$ and $Nmax([0,7],4000)$ depicted in 7.2.

Area and power are obtained thanks to the models presented in Section 5.5.
For a mapping π restricted to an NPU G_i, they are set respectively to $a_i(\pi) = a_{i,RAM}(\pi) + a_{i,NPU}(\pi)$ and, $pw_i(\pi) = pw_{i,RAM}(\pi) + pw_{i,NPU}(\pi)$ where $a_{i,RAM}$ (resp. $pw_{i,RAM}$) corresponds to the area (resp. the power) associated to the SRAM R_i and $a_{i,NPU}$ (resp. $pw_{i,RAM}$) corresponds to the area (resp. the power) associated to the NPU G_i. As stated in Subsubsection 5.5.2.1 and 5.5.3.1, the RAM area $a_{i,RAM}(\pi)$ and power $pw_{i,RAM}(\pi)$

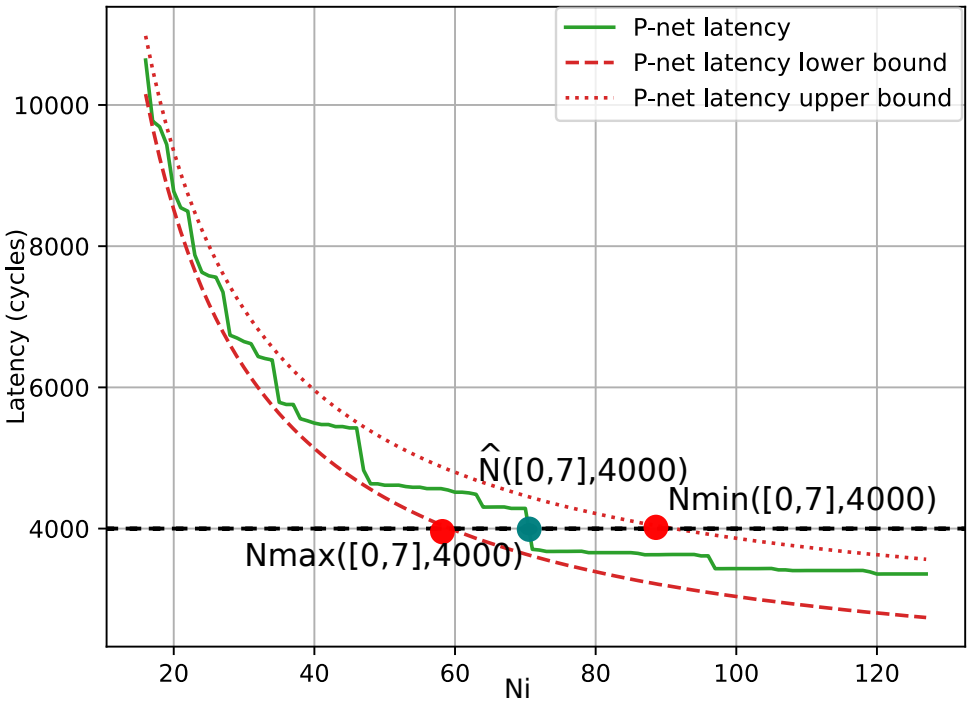

Fig. 7.2 *Minimal N_i for an execution time specification*

depend solely on the RAM capacity $\widehat{K}_i(\pi)$ determined following Theorem 7.2.6. The area, power, and energy of the NPU G_i are given by Equations 5.3, 5.7, and 5.8.

The area and power of the logic blocks (L in Figure 6.7) responsible for connecting the RAMs and NPU, as discussed in Subsection 6.8.1, were not considered due to our limited access to simulation tools that help us to derive closed-formulas. Nonetheless, if such models become accessible in the future, they can be integrated into the RAMs power and area models. However, considering that these blocks are on the same chip, we do not expect excessive power consumption or area in comparison to the other components (NPU and RAMs). It will not affect then the optimized solution.

Finally, it is important to highlight that, in reality, according to the simulations in Figures 5.4 and 5.3, the area and power were not strictly monotonic. However, they were eventually modeled this way due to our understanding of the phenomenon. The non-monotonicity was not fully understood and could potentially be attributed to High-Level Synthesis optimizations that cannot be predicted. Furthermore, in Subsection 7.8.1, we see that this hypothesis can actually be eliminated, but it introduces more computational complexity into the process.

7.7.2 Optimization of Throughput and Latency

We start our experimentation by considering the latency minimization. In this paragraph, the latency is computed for the scenario where *fmaps* are processed simultaneously. For a fixed period, we execute the algorithm outlined in Section 7.6, selecting $\varphi(\pi) = Lat_1(\pi) = n.P$. We then compare this latency value (Lat_2) to the latency achieved by using a single NPU to meet the period constraint (it has $\widehat{N}([0, \ell - 1], P)$ PEs calculated in Subsection 7.7.1).

We observe that smaller values of the period P are unattainable with a single NPU and that a pipeline allows to decrease drastically the period. In what follows, we limit the solutions to fewer than 700 PEs to remain realistic.

Table 7.1 compares the minimum period obtained for a single NPU (note that in this case the latency and the period are equal as the images are processed sequentially) vs. the minimum period for a pipeline (obtained with the algorithm in Section 7.6) with its corresponding minimum latency.

Table 7.1 *Minimal reachable period P and the corresponding latency for a pipeline architecture vs. a single NPU (in cycles)*

		Pipeline Solution	
NN	Single NPU	Min. P	Corresp. Lat_1.
MobileNet	26810	8400	176400
VGG-like	117890	34000	102000
PNet	2720	1470	5880

Throughput can be optimized 3.2 times, 3.5 times, and 1.85 times respectively for MobileNet, VGG-like, and PNet using a pipeline architecture. However, the pipeline's latency is significantly higher; for MobileNet, using 21 NPUs results in a latency 6.6 times higher than the best single NPU latency. In general, using one NPU is optimal for the latency when feasible, but for unattainable periods, the pipeline becomes necessary. The consequence is that the pipeline is a way to heavily reduce the period of the system, but with the increasing of the latency. The conclusion found are the same if the *fmaps* were processed sequentially (Appendix E).

7.7.3 Minimization of Processing Elements Number

We tackle in this section the minimization of the whole number of PEs ($\widehat{N}$). Figure 7.3 presents $\widehat{N}$ depending on the period for the three NNs. The green curves correspond to the optimal pipeline solution obtained using our dynamic program algorithm; the red ones correspond to the solution using a unique NPU ($\widehat{N}([0, \ell - 1], P)$ given by the binary search detailed in 7.7.1).

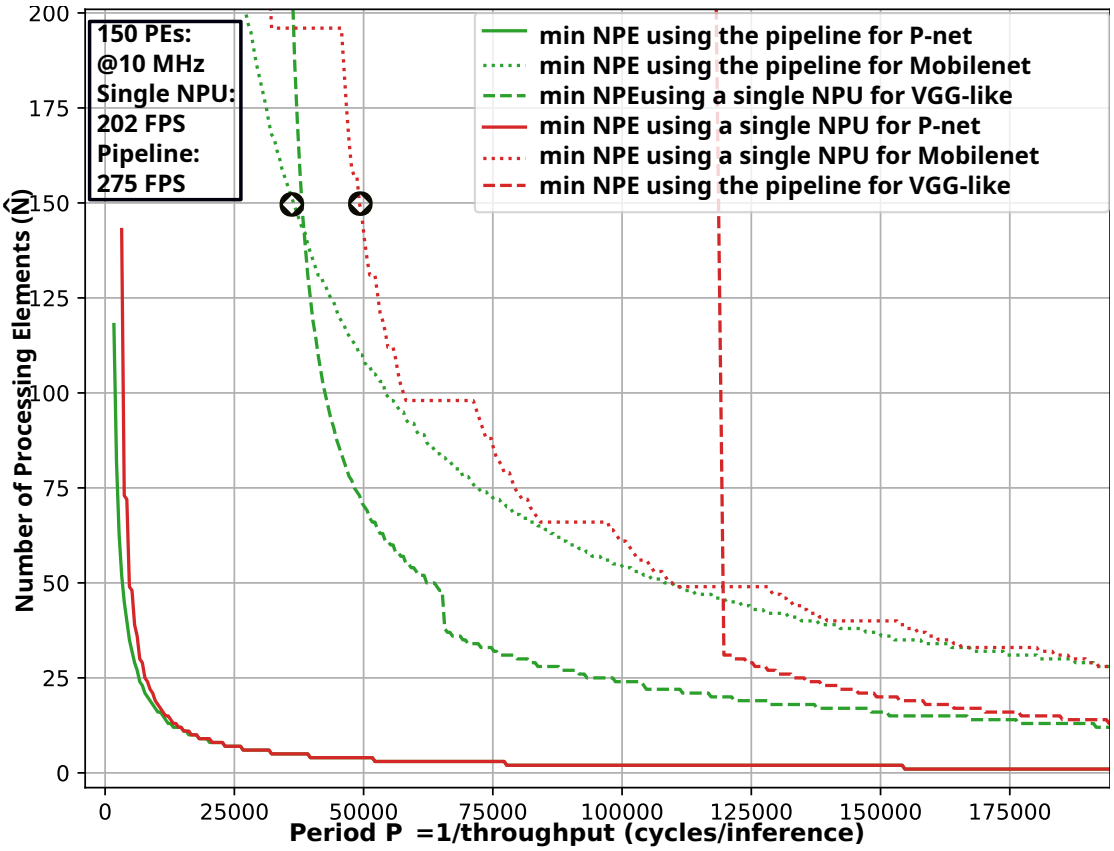

Fig. 7.3 *PEs number under throughput constraints for 3 NNs*

As mentioned earlier, smaller values of the period cannot be achieved considering only one single NPU. Furthermore, we observe that the green curves are always under their corresponding red ones. The consequence is that PEs are underutilized for many layers for the single NPU, while the pipeline optimal solution adjusts as possible the PEs number.

Let us consider that a frame corresponds to the total execution of the NN for one image. For example, we observe that for MobileNet, with 150 PEs operating at 10MHz, a single NPU can process 202 frames per second (FPS). However, the optimal pipeline solution for 150 PEs obtained by our algorithm is given by $n = 5$, $N_i = \{20, 56, 18, 49, 7\}$, and the grouping of the layers $[0]$, $[1, 11]$, $[12, 14]$, $[15, 22]$ and $[23, 26]$; the image's processing rate is then 275FPS for the same frequency, resulting in a significant acceleration of 36%.

7.7.4 Minimization of Area

The optimization criterion in this case is the area. The algorithm was applied to the VGG-like network to optimize either the NPU area $a_{NPU}(\pi)$ or the chip area $a(\pi) = a_{NPU}(\pi) + a_{RAM}(\pi)$ (the sum of the NPUs' areas in the pipeline configuration), which includes the NPUs and FMAPS RAMs. The WEIGHTS RAMs are not included, as

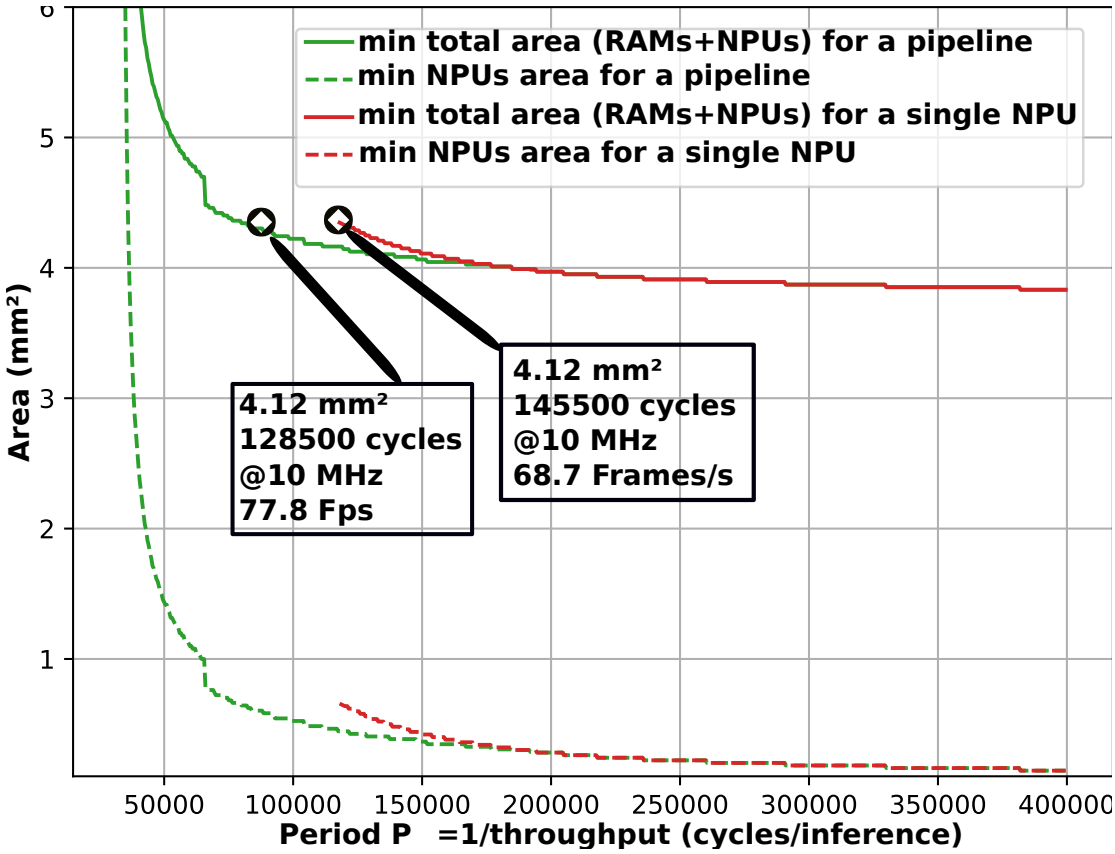

Fig. 7.4 *Total area comparison between one and several pipelined NPUs*

they do not impact the choice of the architecture: the *weights* number is fixed. Note that the two solutions are not necessary the same.

In Figure 7.4, the green curves correspond to the area of the pipelined NPUs solution that satisfies the throughput constraint. The solid line represents the smallest chip area achieved, while the dashed line represents the solution minimizing only the NPUs area. Meanwhile, the red curve corresponds to the area of the single NPU solution. It represents the area of an NPU configured with the minimum number of PEs required to process the NN within P cycles: $\widehat{N}([0, \ell - 1], P)$. The solid line represents the lowest chip area achieved, while the dashed line represents the NPU area. The total area curve has the same shape as the NPU curve, but it is shifted upwards due to the additional size of the RAM (that is larger for the pipeline). The general observation is that the pipeline is interesting for high throughput applications: actually, the pipeline is efficient to enhance the throughput for the same NPUs area. However, the pipeline requires more RAMs than the single NPU solution. Consequently, this leads to a substantial overall area increase. The applicability of the pipeline solution becomes pertinent when the NPU's area attains a sufficient magnitude to become the limiting factor, outweighing the influence of RAMs' area. This scenario aligns with high-throughput applications characterized by a low P. For example, for an area budget of 4.12 mm² and a frequency of 10 MHz, a single NPU achieves 67 FPS, while a pipeline configuration with multiple

NPUs achieves 77.8 FPS (13% higher throughput).

7.7.5 Minimization of Power

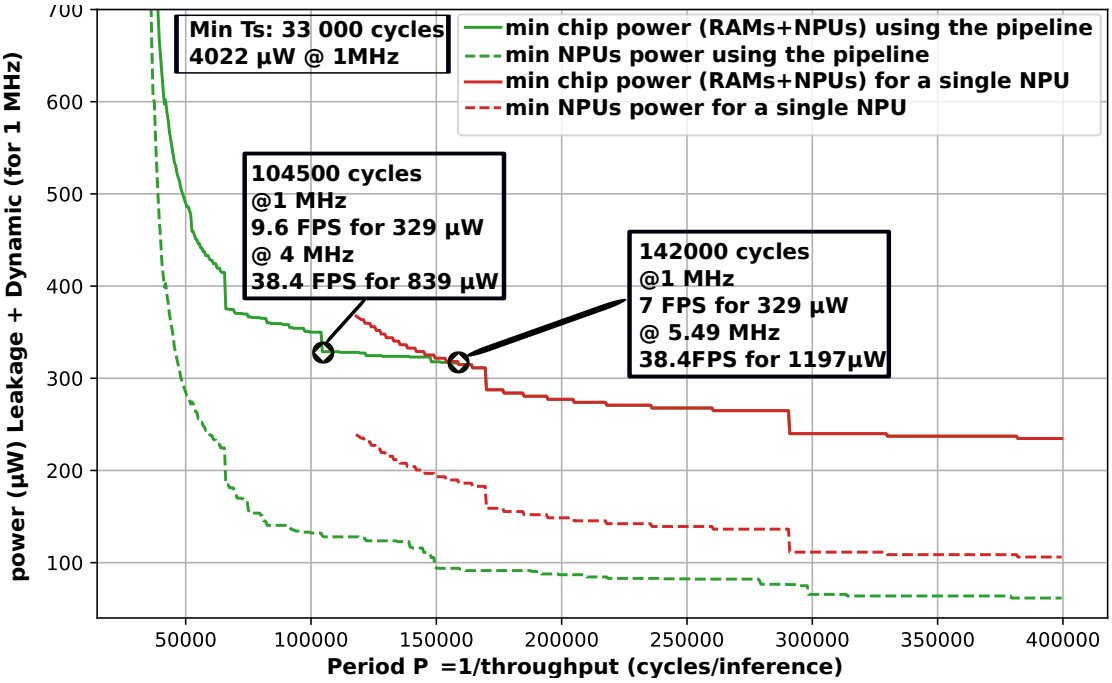

Fig. 7.5 *NPU power comparison between one and several pipelined NPUs*

Let us consider the power minimization considering VGG-like network. Two objective functions were studied: $pw_{NPU}(\pi)$ and $pw(\pi,) = pw_{NPU}(\pi) + pw_{RAM}(\pi)$ to take the SRAMs into consideration (WEIGHTS RAMs are not taken in consideration). Figure 7.5 presents the minimum values pw_{NPU} (dashed lines) and pw (solid lines) depending on the period P at 1MHz for the optimum pipeline (green curves) or a single NPU approach (red curves). The single NPU solution is the power for an NPU having $\widehat{N}([0, \ell - 1], P)$ PEs. The main observation is that the pipeline remains interesting for higher throughput. As an example, the pipeline architecture given by $n = 2$, $N_i = \{16, 6\}$ along with the layers mapping $[0, 7]$, $[8, 10]$ can achieve a processing rate of 9.6FPS at 1MHz and consuming $pw = 329\mu$W; the single NPU configuration can only process 7FPS under the same power budget. As aforementioned, the pipeline method reduces the whole number of PEs, or the power dedicated to NPUs. The downside is that the pipeline requires much more memory than a single NPU. Therefore, the same conclusion as outlined in Subsection 7.7.4 holds true: the pipeline solution is usually interesting when there is a significant enough reduction in the NPU's power to compensate for the increase due to the added RAMs. However, the range of applications where the pipeline architecture is beneficial extends beyond high-throughput applications. Indeed, since dynamic power is linear with respect to frequency (as shown

in [35]), it is possible to lower the frequency (voltage fixed) to decrease the power: when targeting a throughput of 38.4FPS, the optimum pipeline architecture consumes 839μW at 4MHz. The single NPU would need to operate at 5.49MHz to achieve the same throughput, resulting in a power of 1197μW (43% higher than the pipeline). Furthermore, running the circuit at a low frequency allows for low-speed clock constraints, facilitating the use of low-power operators and leading to overall power savings.

7.7.6 Minimization of Energy

Figure 7.6 presents our experiments on the minimization of the energy depending on a fixed period P. The energy is computed considering Lat_2 as a definition of the latency. Similar results are obtained considering Lat_1).

In Fig 7.6, the x-axis represents the target period, and the y-axis shows the minimal energy achieved to meet this constraint. The solid lines represent the solutions minimizing the chip energy, while the dashed lines represent the ones minimizing NPUs one. In green, we have the pipeline solutions and in red the single NPU ones. Overall, we ob-

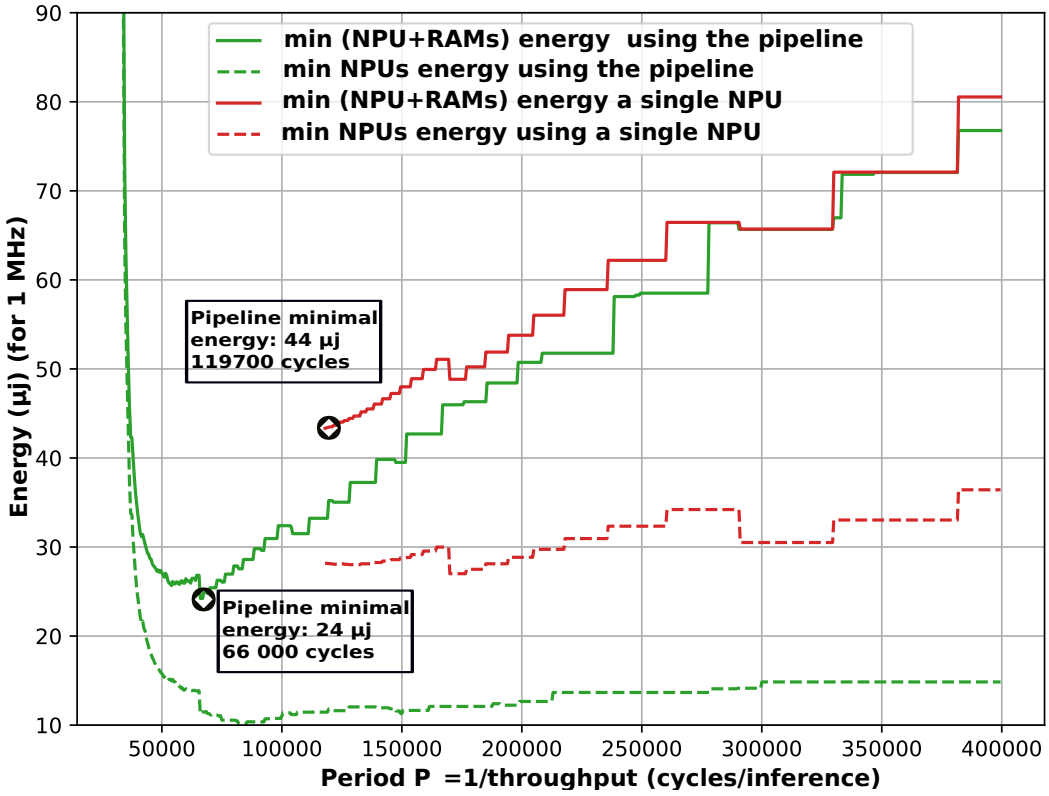

Fig. 7.6 *Energy under throughput constraint for VGG-like*

serve that optimal solutions pipelining multiple NPUs consume less energy than single NPU ones even when the SRAMs are taken into account, particularly when facing high throughput constraints. This is attributed to the fact that the power of a single NPU is high under such constraints. The minimum energy of 24µJ is achieved for $P = 66000$

cycles (the pipeline configuration is $n = 3$, $N_i = \{26, 11, 1\}$ along with the mapping $[0, 1, 2, 3, 4, 5, 6, 7]$, $[8]$, $[9, 10]$ =), which cannot be attained with a single NPU. It is nearly two times lower than the best energy achieved with one NPU ($44\mu J$). Additionally, since the green curve is consistently below the red one, it is possible to optimize energy for a specific throughput. Finally, the optimization can be further improved by reducing the frequency, as described in the previous paragraph.

7.7.7 Minimization of a Convex Linear Combination of KPIs

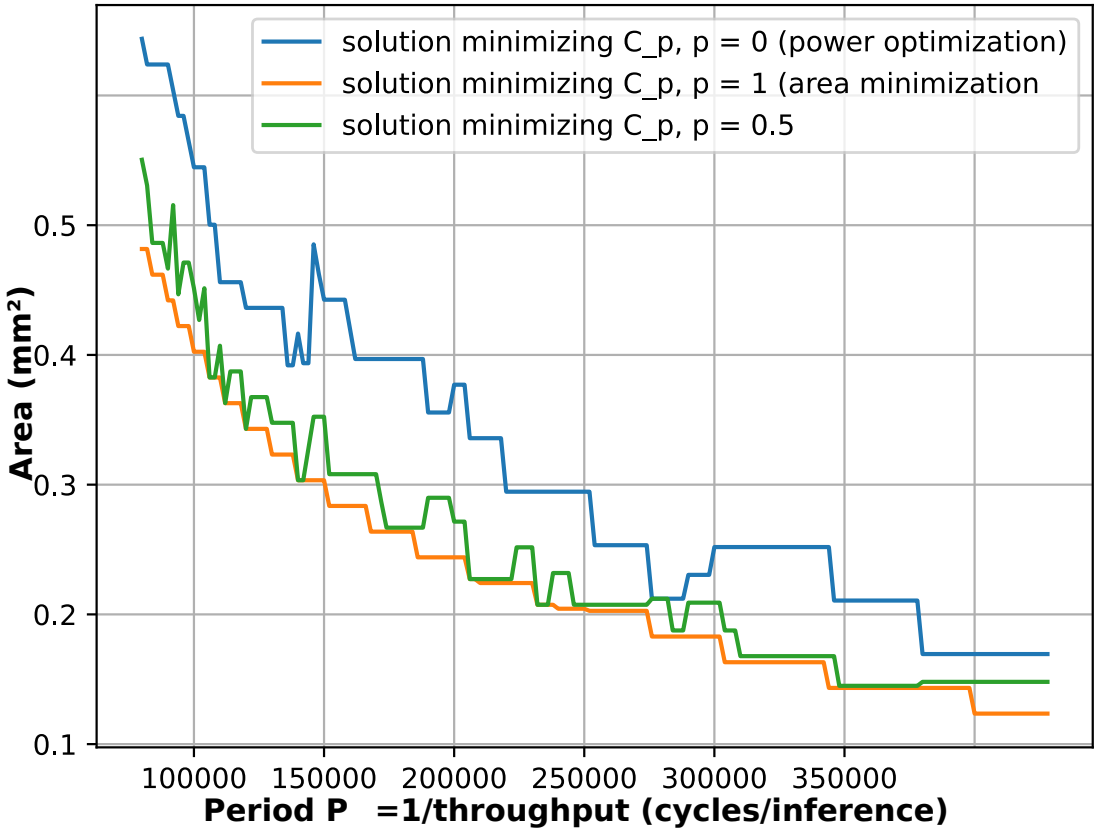

Fig. 7.7 *Area as function of the period for 3 mappings*

It is important to remind that the best solution for optimizing one criterion $\varphi(\pi)$ may not necessarily be the best for optimizing another criterion $\varphi'(\pi)$. In certain situations, it becomes necessary to optimize simultaneously both criteria. One approach could be to use a convex linear combination of KPIs as the criterion. Without loss of generality, we consider area $a(\pi)$ and power $pw(\pi, P)$ of NPUs as criteria. We start by normalizing them within the range $[0, 1]$ by subtracting the minimum value and divide the result by the range (maximum - minimum), we obtain the normalized are $Na(\pi)$ and the normalized power $Npw(\pi, P^\star)$. The criterion to optimize become $\phi(\pi, P) = \lambda \times Na(\pi, P^\star) + (1 - \lambda)) \times Npw(\pi, P^\star)$, where $\lambda \in [0, 1]$ is a weighting factor. The normalization is mandatory as the two criteria does not have the same range of values.

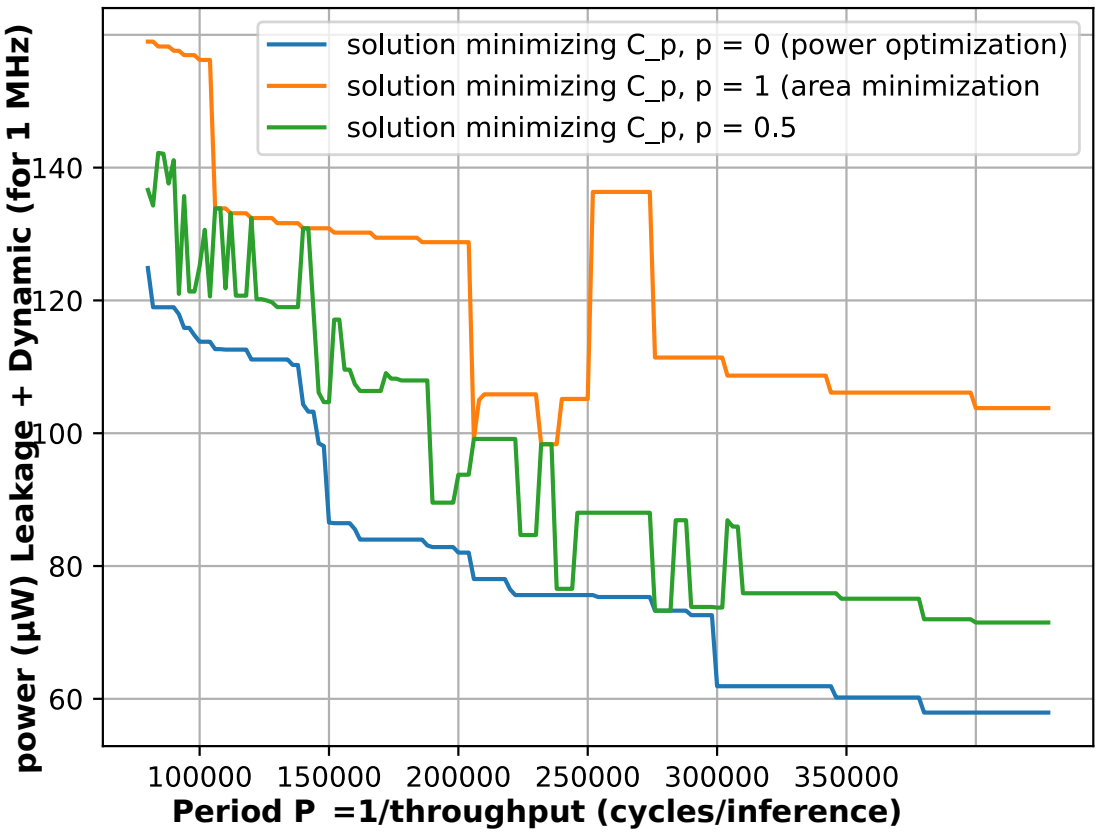

Fig. 7.8 *Power as function of the period for 3 mappings*

Figure 7.7 displays the power, and figure 7.8 displays the area of NPUs for three solutions considering VGG-like network: one optimized for area (orange), one optimized for power (blue), and one with a balanced approach (using $\lambda = 0.5$) that seeks an equitable trade-off between power and area (green). The x-axis represent the targeted period P. Notably, optimizing $\phi(\pi)$ with $\lambda = 0.5$ strikes the best compromise between optimizing power and area. It yields a solution that outperforms the power-focused approach in optimizing area, while still surpassing the area-focused solution in optimizing power. the weight factor λ is adjusted based on the importance assigned to one criterion over the other.

We have shown in this section the application of a generic methodology for optimizing various KPIs using Gemini accelerators, while adhering to throughput constraints. We demonstrated that the pipeline structure enables optimizing the throughput beyond what a single NPU can achieve. Additionally, it offers important optimization for hardware resources, area, power, and energy. The most significant advantage is low-frequency operation, resulting in substantial power gains.

7.8 Extension of the NPUs Pipeline Methodology

In the preceding sections, as well as in Chapter 6, we presented the non-fixed hardware scenario and the fixed-hardware scenario, along with the defined hypotheses, modeling, and solutions. In this section, we will explore potential extensions to broaden the scope and improve performance. In Subsection 7.8.1, we will begin by discussing the elimination of the assumption of monotonicity for the KPIs to optimize, which introduces additional computational complexity. Next, in Subsection 7.8.2, we will explore the potential for extending the pipeline architecture by considering cyclic processing, allowing for data circulation from the last NPU back to the first one. Then, in Subsection 7.8.3, we will investigate the possibility of parallelizing the NPUs to further optimize the execution process.

7.8.1 Extension to Non-monotonic-KPIs with Respect to PEs Number

For the non-fixed hardware scenario, the methodology presented in Section 7.1 can be applied to any KPI φ respecting the constraints detailed in Section 7.3. However, the criterion of monotonicity is in reality not mandatory; although, it is frequently satisfied. Yet, there are cases where it might not hold true. For example, in the case of Gemini, the area and power were not strictly monotonic (see Figures 5.4 and 5.3), although they were eventually modeled as such. Indeed, the linear model and solutions can be adapted for non-monotonic criteria, even though this increases the complexity of the process.

Concerning the model in Section 7.5, the function φ is no more monotonous with respect to the number of PEs, then $\widehat{\varphi}_{[g,h]}(\pi)$ is no more the minimum value achievable by a single NPU processing $[L_g, L_h]$. To address this issue, the new objective to minimize will be: $\varphi(\pi) = \sum_{j=0}^{l-1} \sum_{g=0}^{j} \sum_{h=j}^{l-1} x_{g,h} \times \widetilde{\varphi_{[g,h]}(\pi)}$, where $\widetilde{\varphi_{[g,h]}(\pi)}$ is computed solving: $min_{i \in A} \varphi_i(\pi_{gh})$ with π_{gh} the allocation such as $\forall (j, j') \in [g, h]^2, \pi(j) = \pi(j')$ and $A = \{i \in \mathbb{N} - \{0\}, P_{i,[g,h]} \leq P^\star\}$. In simpler terms, we select $\widetilde{\varphi_{[g,h]}(\pi)}$ as the minimum KPI for the layer set $[L_g, L_h]$, ensuring that the period constraint is met (the minimum PEs number is not necessary the one achieving the minimum as there is no more monotonicity hypothesis).

In the dynamic program presented in 7.6, u will be valuated by $\widetilde{\varphi_{[g,h]}(\pi)}$. The limitation here lies in the fact that the complexity of u is dependent on the specific KPI targeted for optimization. For the case where φ is monotonic with PEs, the computation of u were always at most logarithmic.

7.8.2 Cyclic Pipeline of NPUs

The idea of the pipeline can be extended to a cyclic pipelining. Using the pipeline described in Section 6.4, there are $\frac{\ell.\ell(\ell+1)}{2}$ possible layer sets. By incorporating the possibility to cycle from $L_{\ell-1}$ to L_0, the number of possible layer sets becomes $\frac{\ell.\ell(\ell+1)}{2} \times \ell$. This is because the number of possibilities increases by a factor of ℓ due to the presence of ℓ cycling options (from L_0 to $L_{\ell-1}$). This implies that the dyanimic programming algorithm described in Section 7.6 can be implemented the same way, and the complexity will still be polynomial, bounded by $\mathcal{O}(\ell^5 \log \ell)$. Finally, the implementation effort is minimal, as only the connection between RAM R_{n-1} and NPU G_0 is introduced. This

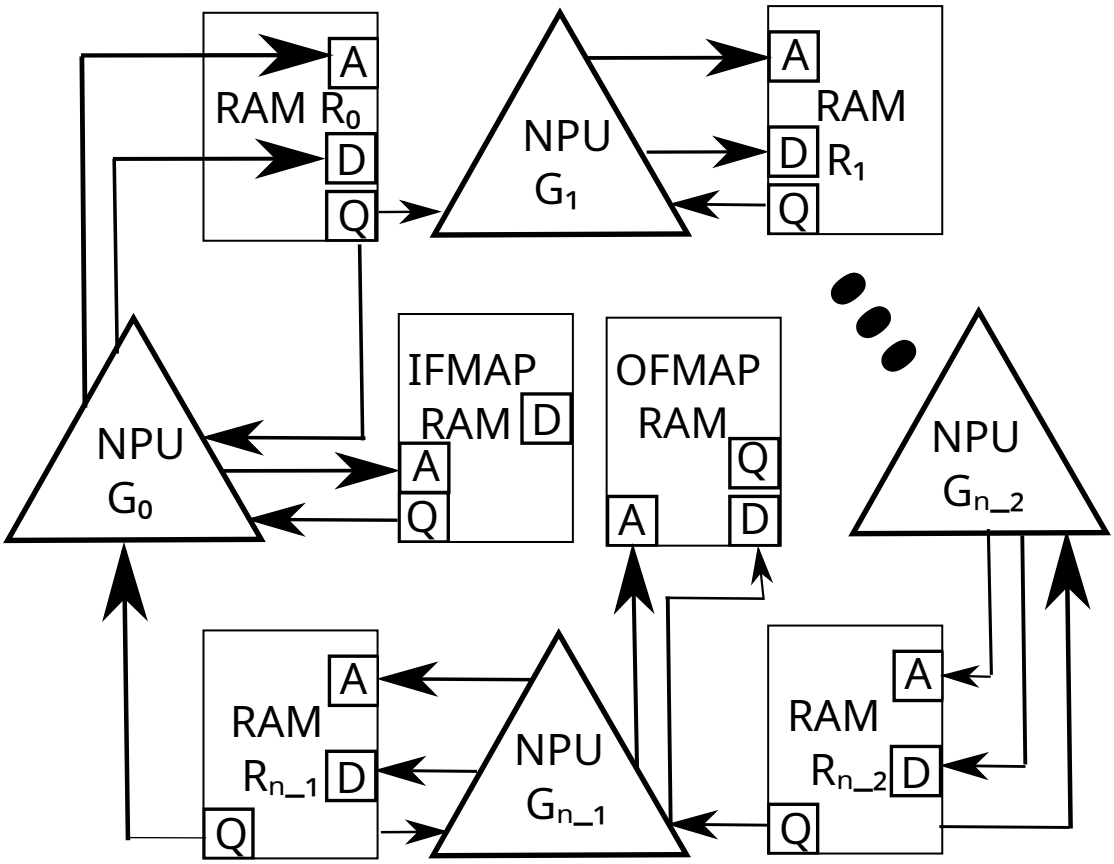

Fig. 7.9 *NPUs in cyclic pipeline*

approach is depicted in Figure 7.9, where the NPU G_0 is communicating with G_{n-1}.

The cycling is beneficial if merging the last layers with the first one could help balance the execution times. However, the drawback of this method is that it introduces an additional latency to the execution time of each image to synchronize the NPUs. For instance, in Figure 7.10, with three NPUs G_0, G_1, and G_2 processing layer sets: L_0, L_3, L_1, and L_2 respectively. Three images can be then processed simultaneously. However, once an *ifmap* from the three is processed (the yellow one, for example, in Figure 7.10), the next *ifmap* (green) can only be loaded when the three first *ifmaps* are completely processed. This is because G_0 is used to process L_0 for the new *ifmap*, but

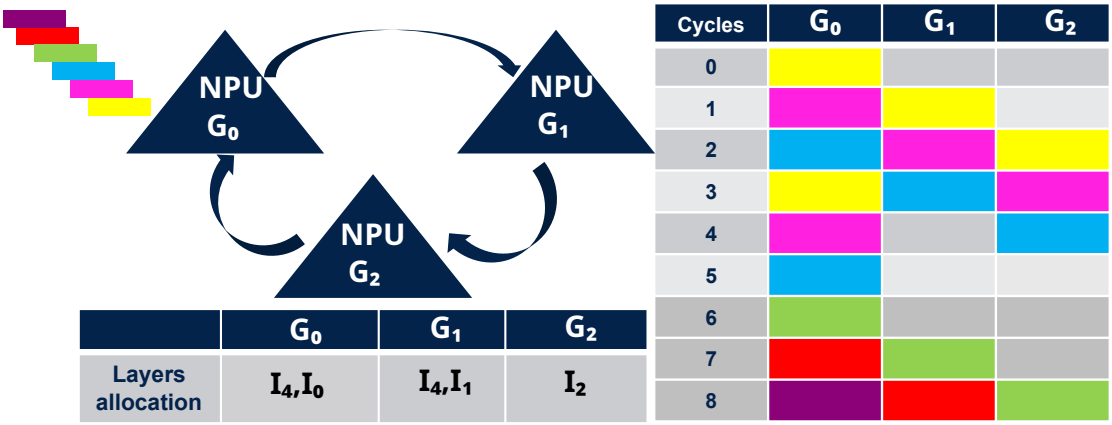

Fig. 7.10 *NPUs in cyclic pipeline example*

also the layer L_3 for the other two *ifmaps* (magenta and blue) from the first load. As a result, it leads to 2 cycles where G_1 and G_2 are IDLE. Unfortunately, the benefice of merging last layers with firsts is not advantageous enough to balance the added latency for NPUs synchronization. When considering this additional latency, the cyclic pipeline was never chosen as the optimal solution for any NN tested.

7.8.3 Parallelizing NPUs

In this subsection, we inquire whether parallelizing the processing of *ifmaps* across NPUs would be judicious.

The advantage of parallelizing the NPUs is processing n *ifmaps* simultaneously using n NPUs. As illustrated in Figure 7.11, in the case of 3 NPUs, the *ifmaps* will be divided into 3 distinct batches. Each of these batches is then written into the *fmaps* RAMs corresponding to its respective NPU. The three batches are concurrently processed by their respective NPUs, although within a batch, the *ifmaps* are processed sequentially. Since the NPUs might not necessarily possess the same N_i values, their execution at each cycle differs. Consequently, *weights* sharing among them is not feasible, except in the scenario where all NPUs have identical N_i values.

The key question is how this method is more efficient compared to using a single NPU equipped with the total number of PEs. Within a single PE, N PEs can process N pixels simultaneously. Thus, if all PEs are fully utilized, sequentially processing n *ifmaps* by employing N pixels per NPU is equivalent to concurrently processing N *ifmaps* with N/n PEs for each NPU. Therefore, parallelization becomes advantageous only when PEs are not optimally utilized for all layer (when there is a latency saturation, as observed in Figure 6.1 for Gemini). This scenario, however, does not exist in the designer problem (non-fixed hardware scenario), as we never consider utilizing more PEs than necessary. However, as elaborated in Section 6.1, the latency and then throughput is bounded when using a single NPU. With this method, the throughput upper bound

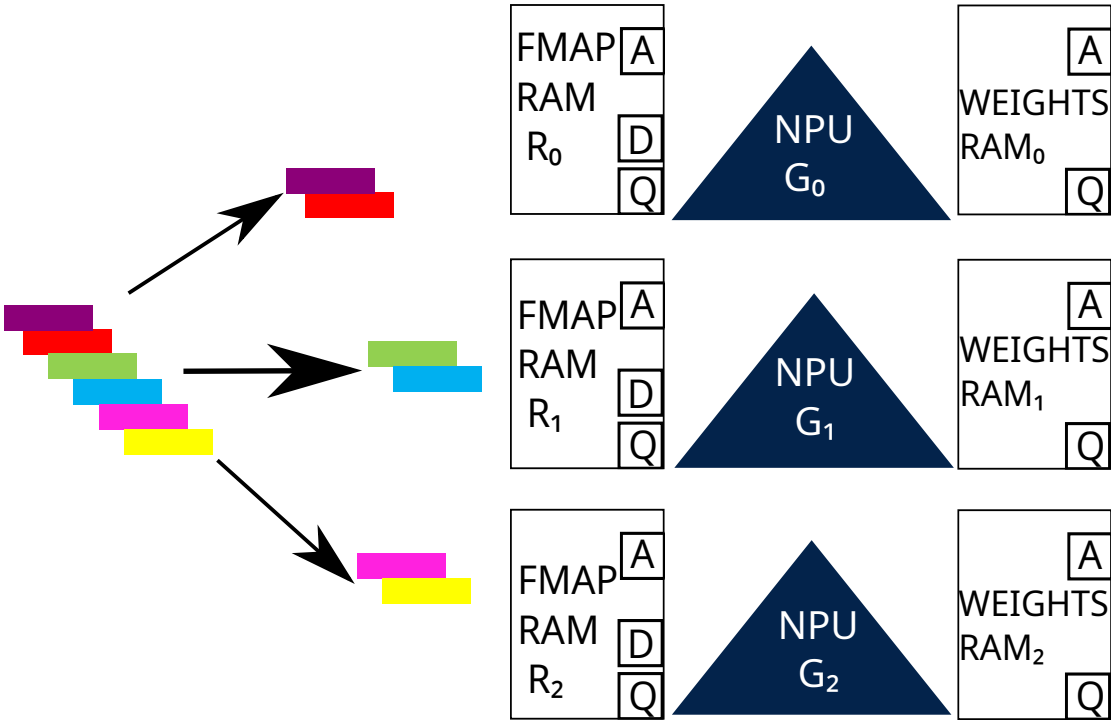

Fig. 7.11 *6 ifmaps processed by 3 NPUs in parallel*

can be improved if adding an extra hardware is not a concern, reaching a configuration of $\frac{1}{n \times N_i fmap}$, where $N_i fmap$ represents the number of *ifmaps* to be processed. This unfeasible solution entails processing each *ifmap* using a different NPU, which, in reality, is impractical for stream applications where a high amount of *ifmaps* need to be processed.

Compared to the pipeline approach, parallelization is less optimal. It shares the same drawback as the single NPU solution, where the number of processors is determined based on all NN layers, while in the pipeline, it is optimized for a specific set of layers. Hence, the pipeline approach will consistently hold an advantage.

Finally, the additional hardware costs of this parallelization solution need to be considered. Compared to a single NPU, we need to account for the expense in area and power of adding an NPU (c_0 in Equation 5.3), as well as having a WEIGHTS RAM for each NPU when they do not share the same PEs number.

7.9 Conclusion

This chapter revisits the pipeline introduced in Chapter 6, which overcomes the limitations of using a single accelerator. It adopts a high-level optimization approach that does not require optimization within the NPUs themselves. The particularity of this chapter lies in its focus on the design level, allowing us to customize the pipeline structure, including the number of accelerators, their parallelization (number of processing

elements), their arrangement, and the sizes of RAMs

In this chapter, we present related works on pipelines, along with similar research on the topic. We also classify the pipeline as a Single Assembly-Line Balancing Problem (SALBP). We then discuss the prerequisites for this optimization, including the ability to have closed formulas for calculating the minimum RAM size required to store a layer and the minimum number of NPU processing elements needed to achieve an inference within the execution time.

The chapter delves then into the Key Performance Indicators (KPIs) that can be optimized. These encompass the ones presented in Chapter 6, but they now include hardware KPIs as the pipeline structure can be modified. There is a multitude of KPIs available for optimization, with the only requirement being adherence to the aggregability constraint and the condition of monotonicity with respect to parallelization. Since throughput is the primary focus and was limited by the single accelerator approach, we introduce a constraint where the optimization objective is to enhance certain KPIs while adhering to a throughput constraint. This strategy is particularly interesting because the KPIs are often antagonistic to throughput.

In this chapter, we also present an integer linear model of the problem and a polynomial dynamic algorithm that precisely solves the problem. This algorithm is illustrated in a study involving Gemini's accelerator, with the objective of demonstrating the advantages of the pipeline over the use of a single NPU with high parallelization levels for optimizing various KPIs. Within this framework, we show that the pipeline can achieve throughputs that were unattainable with a single NPU. Furthermore, for a given throughput, it excels in efficiently reducing area, power consumption, energy usage, and other KPIs that encompass the aforementioned factors. The power reduction aspect is particularly crucial because in certain cases, achieving a high throughput is unnecessary when other system components are limiting, as happens when using accelerators with low-throughput cameras. However, this strategy allows us to operate at lower frequencies, thereby reducing dynamic power consumption.

Finally, we explore the possibility of extending this pipeline strategy. We show that, at the cost of complexity, we can optimize KPIs that do are not necessarily monotone with increased parallelization. We also demonstrate that the idea of cycling the pipeline or using NPUs in parallel is, in reality, not efficient.

The simplicity of the pipeline approach is efficient for optimizing the processing of feed-forward neural networks inference, as it aligns perfectly with the serial structure of NNs. The approach described in this chapter is entirely complementary to the one outlined in the previous chapter, where the pipeline structure is fixed. In fact, a pipeline can be designed for a one specific NN and then reused for another. In these cases, the pipeline is fixed for the second NN, and algorithms from Chapter 6 must be employed.

However, there is potential to directly design a pipeline that is well-suited for several given NNs, specifying an optimized mapping for each NN according to the desired KPIs. This pipeline strategy could also be adapted to recent NNs that are more complex, with features like residual connection layers, for instance.

CHAPTER 8

Conclusion

In the course of this thesis, we ventured into the complexities of neural network hardware acceleration, driven by an industrial project within STMicroelectronics called Gemini, which shares its name with the accelerator under design. This research trajectory encompassed activities ranging from the design and implementation of neural networks on ASICs for edge computing to the evaluation of Gemini's performance, and ultimately, the exploration of advanced optimization techniques using a multi-accelerator pipeline.

This research was driven by the rapid evolution of neural networks, a field that has seen a profound transformation. Today, after this revolution, a multitude of neural network types with various architecture and different applications have emerged. In Chapter 2, we define the scope of our investigation, focusing on the inference of feed-forward neural networks that support conventional layers such as convolutions, depthwise operations, pooling, and fully connected layers. Furthermore, among the myriad neural networks serving various domains, we concentrate on those of moderate size designed for edge computing applications. Notably, we delve into the specifics of MobileNet, VGG-like, and PNet, which serve as benchmark models throughout our research journey.

Once we have identified the target of the hardware, it was crucial to gain an understanding of the architecture used to implement these neural networks (In Chapter 3). We discovered that the architecture imposed for Gemini aligns with the paradigm of near-memory computing accelerators, employing an output-stationary data-flow strategy. This approach optimizes data reuse and enhances parallelization, leading to speed improvements while also efficiently reducing power consumption. This strategy can be complemented by quantization techniques, which further optimize performance. Additionally, we conducted a comprehensive review of various Key Performance Indicators (KPIs) used to characterize accelerators, facilitating their comparative analysis.

In Chapter 4, we delve into the specific requirements that the Gemini architecture must meet, encompassing aspects such as configurability, performance criteria, and validation processes. We also address the design and implementation challenges, resolving them through architectural optimizations and high-level synthesis (HLS) techniques wherever such methods are applicable.

These concerted efforts culminated in the development of a competitive Intellectual Property (IP) that stands up to benchmark references, as affirmed by our KPIs estimations. The practical validation was confirmed through the successful execution of a Gemini version in the initial tests on P18.

In addition to performance considerations, configurability has always been a central focus of the Gemini project. The architecture's flexibility is designed to adapt to various applications with differing trade-offs. However, to fully leverage this valuable feature, it is essential to simplify the process for the end-user when selecting the architecture. Often, their requirements are expressed through the used neural network and the specific performance trade-offs desired.

In this context, we have developed in Chapter 5, a strategy to estimate the KPIs (such as latency, area and power) as a function of two architectural parameters, which are known at a high-level of design abstraction. This strategy is based on predictive models based on a data set of simulations obtained using industrial tools.

Thanks to this performance prediction tool, we gained valuable insights into the evolution of KPIs as function of the architectural parameters. We observed that the initial strategy of scaling accelerators and increasing parallelization was inefficient in achieving scalable performance. This approach had inherent limitations, particularly in its sequential image processing methodology.

After a closer look at the structure of feed-forward neural networks, we recognized that a pipeline of accelerators could offer an interesting high-level optimization complementary to the fine-grained one within individual accelerators. In Chapter 6, we implemented this concept within a fixed pipeline scenario. Here, the number and order of accelerators, along with their specific architectures and RAM sizes, were predetermined. The objective of this approach was to optimize either throughput or latency separately or jointly. In this optic, we developed dynamic programming solutions (as well as integer linear models) that exactly solve the problem for each case. We tested this approach on the Gemini accelerator and found that we could adjust efficiently the neural network layers mappings to optimize the execution KPIs. The key to efficiency lay in our ability to adapt layers or groups of layers to the parallelization of the Neural Processing Units, rather than the entire neural network. This approach could further be improved by fine-tuning the number of processing elements used within each accelerator.

In Chapter 7, we expanded upon the promising pipeline strategy by introducing even more flexibility into the pipeline's design. This approach allowed us to customize the pipeline's structure, including the number of accelerators, their parallelization, arrangement, and RAM sizes. We also introduced additional hardware-related performance metrics that could be optimized within the pipeline. The objective is to identify an

optimal pipeline structure and layer mappings that enhanced various KPIs, all while adhering to throughput constraints. This approach, illustrated on Gemini, enabled the efficient optimization of various aspects, such as the minimization of processing elements within the entire circuit, area, power consumption, energy, and various combinations of these KPIs. Notably, this novel pipeline solution outperformed the previous approach that employed a single accelerator with a high number of processing elements. If there are two key takeaways from this study of the Gemini pipeline, it is the ability to achieve a throughput beyond what could be attained with a single accelerator, which is particularly valuable for applications that demand exceptionally high throughput. Additionally, for applications that do not prioritize high throughput, it is possible to maintain a suitable throughput while operating at low frequencies to reduce power consumption (and energy consumption as a result).

This thesis has made contributions to both research and industry. In the industrial context, during my thesis, I actively participated in the design and implementation of a configurable neural network accelerator tailored for feed-forward neural network inference, along with the development of an efficient KPI evaluation tool. This tool aids customers in finding the appropriate architecture for their specific applications. From a research perspective, while this prediction tool addresses an industrial need, it is based on a strategy that can be adapted to other accelerator architectures. Additionally, the pipeline optimization method we introduced in this work is a high-level approach with the potential for adaptation to various accelerators beyond the Gemini project. These two aspects have led to two publications, one submitted and one to be published [98], as well as the presentation of a poster at GDR SOC2 2023.

As I conclude this thesis, I have come to realize that there are several opportunities for further optimizing the hardware acceleration of neural network inference.

In terms of the accelerator's architecture, although I could not explore it within the scope of my thesis, the In-Memory Computing paradigm shows great promise for edge applications due to its potential for low power consumption and high throughput. Moreover, it allows for finer control over the pixels being processed, potentially eliminating the constraints posed by the current scheduling program that hinders our ability to exploit sparsity effectively.

Another area of improvement within the accelerator involves revisiting the quantization process and finding more efficient hardware implementations for various quantization techniques. It is worth mentioning that my PhD student colleagues at STMicroelectronics have delved into these aspects in their own research.

Regarding coarse-grained optimization, especially within the context of the pipeline, there is room for further research. It would be beneficial to investigate the adaptation of this strategy to support jointly a multitude of neural networks, as well as its application

to other neural network layers, such as residual connections or even recurrent neural networks.

I believe that continuing to optimize hardware at both the fine-grained and coarse-grained levels represents the best approach, as it leverages the advantages of each.

REFERENCES

[1] A. AGNETIS, A. CIANCIMINO, M. L. and PIZZICHELLA, M. [1995], 'Balancing flexible lines for car components assembly', *International Journal of Production Research* **33**(2), 333–350.
URL: *https://doi.org/10.1080/00207549508930152*

[2] Aarno, D. and Engblom, J. [2015], Chapter 7 - dma: A concrete modeling example, *in* D. Aarno and J. Engblom, eds, 'Software and System Development using Virtual Platforms', Morgan Kaufmann, Boston, pp. 211–236.
URL: *https://www.sciencedirect.com/science/article/pii/B978012800725900007X*

[3] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y. and Zheng, X. [2016], Tensorflow: A system for large-scale machine learning, *in* 'Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation', OSDI'16, USENIX Association, USA, p. 265–283.

[4] Abdelouahab, K., Pelcat, M., Sérot, J. and Berry, F. [2018], 'Accelerating CNN inference on fpgas: A survey', *CoRR* **abs/1806.01683**.
URL: *http://arxiv.org/abs/1806.01683*

[5] Abdulkadirov, R., Lyakhov, P. and Nagornov, N. [2023], 'Survey of optimization algorithms in modern neural networks', *Mathematics* **11**(11).
URL: *https://www.mdpi.com/2227-7390/11/11/2466*

[6] Abiodun, O. I., Jantan, A., Omolara, A. E., Dada, K. V., Mohamed, N. A. and Arshad, H. [2018], 'State-of-the-art in artificial neural network applications: A survey', *Heliyon* **4**(11).

[7] Alexandrescu, A., Agavriloaei, I. and Craus, M. [2011], A genetic algorithm for mapping tasks in heterogeneous computing systems, *in* '15th International Conference on System Theory, Control and Computing', pp. 1–6.

[8] Alexandrescu, A. and Craus, M. [2010], Improving mapping heuristics in heterogeneous computing, *in* 'Proceedings ECIT2010 6th European Conference on Intelligent Systems and Technologies', pp. 7–9.

[9] Ali, N., Philippe, J.-M., Tain, B. and Coussy, P. [2021], Exploration and generation of efficient fpga-based deep neural network accelerators, *in* '2021 IEEE Workshop on Signal Processing Systems (SiPS)', pp. 123–128.

[10] Álvarez-Miranda, E. and Pereira, J. [2019], 'On the complexity of assembly line balancing problems', *Computers & Operations Research* **108**, 182–186.

[11] Bacis, M., Natale, G., Del Sozzo, E. and Santambrogio, M. D. [2017], A pipelined and scalable dataflow implementation of convolutional neural networks on fpga, *in* '2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)', pp. 90–97.

[12] Bain, N., Guizzetti, R., Taly, E., Oudrhiri, A., Paille, B., Urard, P. and Pétrot, F. [2023], Quantization modes for neural network inference: Asic implementation trade-offs, *in* '2023 International Joint Conference on Neural Networks (IJCNN)', pp. 01–08.

[13] Balasubramonian, R., Kahng, A. B., Muralimanohar, N., Shafiee, A. and Srinivas, V. [2017], 'Cacti 7: New tools for interconnect exploration in innovative off-chip memories', *ACM Trans. Archit. Code Optim.* **14**(2).
URL: *https://doi.org/10.1145/3085572*

[14] Bao, G., Graeber, M. B. and Wang, X. [2020], Depthwise multiception convolution for reducing network parameters without sacrificing accuracy, *in* '2020 16th International Conference on Control, Automation, Robotics and Vision (ICARCV)', pp. 747–752.

[15] Baybars, [1986], 'A survey of exact algorithms for the simple assembly line balancing problem', *Management Science* **32**(8), 909–932.
URL: *https://doi.org/10.1287/mnsc.32.8.909*

[16] Bohm Agostini, N., Dong, S., Karimi, E., Torrents Lapuerta, M., Cano, J., Abellán, J. L. and Kaeli, D. [2020], Design space exploration of accelerators and end-to-end dnn evaluation with tflite-soc, *in* '2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)', pp. 10–19.

[17] Boysen, N. and Fliedner, M. [2008], 'A versatile algorithm for assembly line balancing', *European Journal of Operational Research* **184**(1), 39–56.
URL: *https://www.sciencedirect.com/science/article/pii/S0377221706011362*

[18] Boysen, N., Fliedner, M. and Scholl, A. [2007], 'A classification of assembly line balancing problems', *European Journal of Operational Research* **183**(2), 674–

693.
URL: *https://www.sciencedirect.com/science/article/pii/S0377221706010435*

[19] Brown, G., Tenace, V. and Gaillardon, P.-E. [2021], Nemo-cnn: An efficient near-memory accelerator for convolutional neural networks, *in* '2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)', pp. 57–60.

[20] Brown, R. [2020], 'Donald o. hebb and the organization of behavior: 17 years in the writing', *Molecular Brain* **13**.

[21] BURNS, L. D. and DAGANZO, C. F. [1987], 'Assembly line job sequencing principles', *International Journal of Production Research* **25**(1), 71–99.
URL: *https://doi.org/10.1080/00207548708919824*

[22] Cai, X., Wang, Y., Ma, X., Han, Y. and Zhang, L. [2022], Deepburning-seg: Generating dnn accelerators of segment-grained pipeline architecture, *in* '2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)', pp. 1396–1413.

[23] Carballo-Hernández, W., Pelcat, M., Bhattacharyya, S. S., Galán, R. C. and Berry, F. [2023], 'Flydeling: Streamlined performance models for hardware acceleration of cnns through system identification', *ACM Trans. Model. Perform. Eval. Comput. Syst.* **8**(3).
URL: *https://doi.org/10.1145/3594870*

[24] Chakradhar, S., Sankaradass, M., Jakkula, V. and Cadambi, H. [2010], A dynamically configurable coprocessor for convolutional neural networks, pp. 247–257.

[25] Chen, Y.-H., Yang, T.-J., Emer, J. and Sze, V. [2019], 'Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices', *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* **9**(2), 292–308.

[26] Cho, D., Tai, Y.-W. and Kweon, I. S. [2018], 'Deep convolutional neural network for natural image matting using initial alpha mattes', *IEEE Transactions on Image Processing* **28**(3), 1054–1067.

[27] Choi, S. C. and Youn, H. Y. [2005], Task mapping algorithm for heterogeneous computing system allowing high throughput and load balancing, *in* V. S. Sunderam, G. D. van Albada, P. M. A. Sloot and J. Dongarra, eds, 'Computational Science – ICCS 2005', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1000–1003.

[28] Chollet, F. [2016], 'Xception: Deep learning with depthwise separable convolutions', *CoRR* **abs/1610.02357**.
URL: *http://arxiv.org/abs/1610.02357*

[29] Choon, L. S., Samsudin, A. and Budiarto, R. [2004], 'Lightweight and cost-effective mpeg video encryption', *Proc. of Information and Communication Technologies: From Theory to Applications* pp. 525–526.

[30] Cisco [2016], Complete visual networking index (vni) forecast, cisco, june 2016, Technical report, Cisco.

[31] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. [2022], *Introduction to algorithms*, MIT press.

[32] Coussy, P., Gajski, D. D., Meredith, M. and Takach, A. [2009], 'An introduction to high-level synthesis', *IEEE Design Test of Computers* **26**(4), 8–17.

[33] Dananjaya, M. [2015], 'Vlsi power in a nutshell'.

[34] Dao, X.-Q. [2023], 'Performance comparison of large language models on vnhsge english dataset: Openai chatgpt, microsoft bing chat, and google bard', *arXiv preprint arXiv:2307.02288* .

[35] Darwish, T. and Bayoumi, M. [2005], 5 - trends in low-power vlsi design, *in* W.-K. CHEN, ed., 'The Electrical Engineering Handbook', Academic Press, Burlington, pp. 263–280.
URL: *https://www.sciencedirect.com/science/article/pii/B9780121709600500220*

[36] Davis, L. [1987], 'Genetic algorithms and simulated annealing'.
URL: *https://www.osti.gov/biblio/5037281*

[37] Deng, l., Li, J., Huang, J.-T., Yao, K., Yu, D., Seide, F., Seltzer, M., Zweig, G., He, X., Williams, J., Gong, Y. and Acero, A. [2013], Recent advances in deep learning for speech research at microsoft, pp. 8604–8608.

[38] Desoli, G., Chawla, N., Boesch, T., Singh, S., Guidetti, E., Ambroggi, F., Majo, T., Zambotti, P., Ayodhyawasi, M., Singh, H. and Aggarwal, N. [2017], 14.1 a 2.9tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems, pp. 238–239.

[39] Du, Z., Fasthuber, R., Chen, T., Ienne, P., Li, L., Luo, T., Feng, X., Chen, Y. and Temam, O. [2015], 'Shidiannao: Shifting vision processing closer to the sensor', *SIGARCH Comput. Archit. News* **43**(3S), 92–104.
URL: *https://doi.org/10.1145/2872887.2750389*

[40] Du, Z., Fasthuber, R., Chen, T., Ienne, P., Luo, T., Feng, X., Chen, Y. and Temam, O. [2015], 'Shidiannao', *ACM SIGARCH Computer Architecture News* **43**, 92–104.

[41] Elmaghraby, S. E. [2003], Operations research, *in* R. A. Meyers, ed., 'Encyclopedia of Physical Science and Technology (Third Edition)', third edition edn, Academic Press, New York, pp. 193–218.
URL: *https://www.sciencedirect.com/science/article/pii/B0122274105005135*

[42] Erdem, A., Silvano, C., Boesch, T., Ornstein, A. C., pal Singh, S. and Desoli, G. S. [2020], 'Runtime design space exploration and mapping of dcnns for the ultra-low-power orlando soc', *ACM Transactions on Architecture and Code Optimization (TACO)* **17**, 1 – 25.

[43] Erel, E. and Sarin, S. C. [1998], 'A survey of the assembly line balancing procedures', *Production Planning & Control* **9**(5), 414–434.
URL: *https://doi.org/10.1080/095372898233902*

[44] FUMIO AKAGI, H. O. and KIKUCHI, S. [1983], 'A method for assembly line balancing with more than one worker in each station', *International Journal of Production Research* **21**(5), 755–770.
URL: *https://doi.org/10.1080/00207548308942409*

[45] Gearhart, J. L., Adair, K. L., Durfee, J. D., Jones, K. A., Martin, N. and Detry, R. J. [2013], 'Comparison of open-source linear programming solvers.'.
URL: *https://www.osti.gov/biblio/1104761*

[46] Geirhos, R., Janssen, D. H. J., Schütt, H. H., Rauber, J., Bethge, M. and Wichmann, F. A. [2018], 'Comparing deep neural networks against humans: object recognition when the signal gets weaker'.

[47] Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W. and Keutzer, K. [2021], 'A survey of quantization methods for efficient neural network inference'.

[48] Girshick, R., Donahue, J., Darrell, T. and Malik, J. [2014], Rich feature hierarchies for accurate object detection and semantic segmentation, *in* '2014 IEEE Conference on Computer Vision and Pattern Recognition', pp. 580–587.

[49] Gokhale, V., Jin, J., Dundar, A., Martini, B. and Culurciello, E. [2014], A 240 g-ops/s mobile coprocessor for deep neural networks, *in* '2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops', pp. 696–701.

[50] Goodfellow, I., Bengio, Y. and Courville, A. [2016], *Deep Learning*, MIT Press, Cambridge, MA.

[51] Hagberg, A., Swart, P. J. and Schult, D. A. [2008], 'Exploring network structure, dynamics, and function using networkx'.
URL: *https://www.osti.gov/biblio/960616*

[52] Haris, J., Gibson, P., Cano, J., Agostini, N. B. and Kaeli, D. R. [2021], 'Secda: Efficient hardware/software co-design of fpga-based dnn accelerators for edge inference', *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* pp. 33–43.
URL: *https://api.semanticscholar.org/CorpusID:238253107*

[53] Hassani, I. K., Pellegrini, T. and Masquelier, T. [2021], 'Dilated convolution with learnable spacings', *CoRR* **abs/2112.03740**.
URL: *https://arxiv.org/abs/2112.03740*

[54] He, K., Zhang, X., Ren, S. and Sun, J. [2015*a*], 'Deep residual learning for image recognition', *CoRR* **abs/1512.03385**.
URL: *http://arxiv.org/abs/1512.03385*

[55] He, K., Zhang, X., Ren, S. and Sun, J. [2015*b*], 'Deep residual learning for image recognition'.

[56] He, Y. and Xiao, L. [2023], 'Structured pruning for deep convolutional neural networks: A survey'.

[57] Held, M., Karp, R. M. and Shareshian, R. [1963], 'Assembly-line balancing—dynamic programming with precedence constraints', *Operations research* **11**(3), 442–459.

[58] *High-Level Synthesis(HLS) Blue Book* [n.d.]. `https://resources.sw.siemens.com/en-US/e-book-high-level-synthesis-hls-blue-book`.

[59] Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N. and Kingsbury, B. [2012], 'Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups', *IEEE Signal Processing Magazine* **29**(6), 82–97.

[60] Horowitz, M. [2014], 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.

[61] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. and Adam, H. [2017], 'Mobilenets: Efficient convolutional neural

networks for mobile vision applications', *CoRR* **abs/1704.04861**.
URL: *http://arxiv.org/abs/1704.04861*

[62] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J. and Keutzer, K. [2016], 'Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size'.

[63] Ivakhnenko, A. G. and Lapa, V. G. [1965], *Cybernetic Predicting Devices*, CCM Information Corporation.

[64] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H. and Kalenichenko, D. [2018], Quantization and training of neural networks for efficient integer-arithmetic-only inference, *in* 'Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)'.

[65] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. and Darrell, T. [2014], Caffe: Convolutional architecture for fast feature embedding, *in* 'Proceedings of the 22nd ACM International Conference on Multimedia', MM '14, Association for Computing Machinery, New York, NY, USA, p. 675–678.
URL: *https://doi.org/10.1145/2647868.2654889*

[66] Jiao, A. [2020], An intelligent chatbot system based on entity extraction using rasa nlu and neural network, *in* 'Journal of physics: conference series', Vol. 1487, IOP Publishing, p. 012014.

[67] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., luc Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A. and Ross, J. [2017], In-datacenter performance analysis of a tensor processing unit.
URL: *https://arxiv.org/pdf/1704.04760.pdf*

[68] Kedia, R., Goel, S., Balakrishnan, M., Paul, K. and Sen, R. [2021], 'Design space exploration of fpga-based system with multiple dnn accelerators', *IEEE Embedded Systems Letters* **13**(3), 114–117.

[69] Kim, B., Lee, S., Trivedi, A. R. and Song, W. J. [2020], 'Energy-efficient acceleration of deep neural networks on realtime-constrained embedded edge devices', *IEEE Access* **8**, 216259–216270.

[70] Krizhevsky, A., Nair, V. and Hinton, G. [2009], 'The cifar-10 dataset', `https://www.cs.toronto.edu/kriz/cifar.html`.

[71] Krizhevsky, A., Sutskever, I. and Hinton, G. E. [2012], Imagenet classification with deep convolutional neural networks, *in* F. Pereira, C. Burges, L. Bottou and K. Weinberger, eds, 'Advances in Neural Information Processing Systems', Vol. 25, Curran Associates, Inc.
URL: *https://proceedings.neurips.cc/paper*$_f iles/paper/2012/file/c399862d3b9d6b76c8436e92$
$Paper.pdf$

[72] Lahti, S., Sjovall, P., Vanne, J. and Hämäläinen, T. [2018], 'Are we there yet? a study on the state of high-level synthesis', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **PP**, 1–1.

[73] Le Cun, Y., Jackel, L., Boser, B., Denker, J., Graf, H., Guyon, I., Henderson, D., Howard, R. and Hubbard, W. [1989], 'Handwritten digit recognition: applications of neural network chips and automatic learning', *IEEE Communications Magazine* **27**(11), 41–46.

[74] LeCun, C. J. C. Y. and Cortes, C. [1998], 'The mnist database of handwritten digits', `http://yann.lecun.com/exdb/mnist/`.

[75] LeCun, Y. [2019], 1.1 deep learning hardware: Past, present, and future, *in* '2019 IEEE International Solid- State Circuits Conference - (ISSCC)', pp. 12–19.

[76] LeCun, Y., Bengio, Y. and Hinton, G. [2015], 'Deep learning', *nature* **521**(7553), 436–444.

[77] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. and Jackel, L. D. [1989], 'Backpropagation applied to handwritten zip code recognition', *Neural Computation* **1**(4), 541–551.

[78] LeCun, Y. and Cortes, C. [2010], 'MNIST handwritten digit database'.
URL: *http://yann.lecun.com/exdb/mnist/*

[79] Lee, J., Kim, C., Kang, S., Shin, D., Kim, S. and Yoo, H.-J. [2019], 'Unpu: An energy-efficient deep neural network accelerator with fully variable weight bit precision', *IEEE Journal of Solid-State Circuits* **54**(1), 173–185.

[80] Li, Z., Zhang, Y., Ding, A., Zhou, H. and Liu, C. [2021], 'Efficient algorithms for task mapping on heterogeneous cpu/gpu platforms for fast completion time', *Journal of Systems Architecture* **114**, 101936.
URL: *https://www.sciencedirect.com/science/article/pii/S1383762120301934*

[81] Liang, X., Hu, P., Zhang, L., Sun, J. and Yin, G. [2019], 'Mcfnet: Multi-layer concatenation fusion network for medical images fusion', *IEEE Sensors Journal* **19**(16), 7107–7119.

[82] Liu, Z., Xiao, X., Li, C., Ma, S. and Rangyu, D. [2022], 'Optimizing convolutional neural networks on multi-core vector accelerator', *Parallel Computing* **112**, 102945.
URL: *https://www.sciencedirect.com/science/article/pii/S0167819122000424*

[83] Liua, Z., P.Luo, Wang, X. and TANG, X. [2010], 'Large-scale celebfaces attributes (celeba) dataset', `https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html`.

[84] Luling, R. and Monien, B. [1992], Load balancing for distributed branch bound algorithms, *in* 'Proceedings Sixth International Parallel Processing Symposium', pp. 543–548.

[85] Maleki, M. A., Kamal, M. and Afzali-Kusha, A. [2022], 'Heterogeneous multicore array-based dnn accelerator'.

[86] Manna, Z. and Waldinger, R. [1987], 'The origin of a binary-search paradigm', *Science of Computer Programming* **9**(1), 37–83.
URL: *https://www.sciencedirect.com/science/article/pii/0167642387900256*

[87] Martin, G. and Smith, G. [2009], 'High-level synthesis: Past, present, and future', *IEEE Design Test of Computers* **26**, 18–25.

[88] Mayer, R. E. [2014], 'Introduction to multimedia learning.'.

[89] Mei, L., Liu, H., Wu, T., Sumbul, H. E., Verhelst, M. and Beigne, E. [2022], A uniform latency model for dnn accelerators with diverse architectures and dataflows, *in* '2022 Design, Automation,Test in Europe Conference, Exhibition (DATE)', pp. 220–225.

[90] Merolla, P., Arthur, J., Alvarez-Icaza, R., Cassidy, A., Sawada, J., Akopyan, F., Jackson, B., Imam, N., Guo, C., Nakamura, Y., Brezzo, B., Vo, I., Esser, S., Appuswamy, R., Taba, B., Amir, A., Flickner, M., Risk, W., Manohar, R. and Modha, D. [2014], 'Artificial brains a million spiking-neuron integrated circuit

with a scalable communication network and interface', *Science (New York, N.Y.)* **345**, 668–673.

[91] Mo, H., Zhu, W., Hu, W., Wang, G., Li, Q., Li, A., Yin, S., Wei, S. and Liu, L. [2021], 9.2 a 28nm 12.1 tops/w dual-mode cnn processor using effective-weight-based convolution and error-compensation-based prediction, *in* '2021 IEEE International Solid-State Circuits Conference (ISSCC)', Vol. 64, IEEE, pp. 146–148.

[92] Moons, B., Uytterhoeven, R., Dehaene, W. and Verhelst, M. [2017], 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)', IEEE, pp. 246–247.

[93] Moons, B. and Verhelst, M. [2016], 'An energy-efficient precision-scalable convnet processor in a 40-nm cmos', *IEEE Journal of Solid-State Circuits* **PP**, 1–12.

[94] Mou, C., Wang, Q. and Zhang, J. [2022], 'Deep generalized unfolding networks for image restoration'.

[95] Navabi, Z. and Massoumi, M. [1991], 'Investigating simulation of hardware at various levels of abstraction and timing back-annotation of dataflow descriptions', *SIMULATION* **57**(5), 321–332.
URL: *https://doi.org/10.1177/003754979105700511*

[96] Ni, Y., Kim, Y., Rosing, T. and Imani, M. [2022], Online performance and power prediction for edge tpu via comprehensive characterization, *in* 'Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe', DATE '22, European Design and Automation Association, Leuven, BEL, p. 612–615.

[97] Odema, M., Bouzidi, H., Ouarnoughi, H., Niar, S. and Al Faruque, M. A. [2023], 'Magnas: A mapping-aware graph neural architecture search framework for heterogeneous mpsoc deployment', *ACM Trans. Embed. Comput. Syst.* **22**(5s).
URL: *https://doi.org/10.1145/3609386*

[98] Oudrhiri, A., Taly, E., Bain, N., Munier-Kordon, A., Guizzetti, R. and Urard, P. [2023], Performance Modeling and Estimation of a Configurable Output Stationary Neural Network Accelerator. working paper or preprint, in press.
URL: *https://hal.science/hal-04168803*

[99] Palmer, J. D. and McAddis, N. [2019], Documentation as a cross-cutting concern of software, *in* 'Proceedings of the 37th ACM International Conference on the Design of Communication', SIGDOC '19, Association for Computing Machinery,

New York, NY, USA.
URL: *https://doi.org/10.1145/3328020.3353949*

[100] Parashar, A., Rhu, M., Mukkara, A., Puglielli, A., Venkatesan, R., Khailany, B., Emer, J., Keckler, S. W. and Dally, W. J. [2017], 'Scnn: An accelerator for compressed-sparse convolutional neural networks'.

[101] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S. [2019], 'Pytorch: An imperative style, high-performance deep learning library', *CoRR* **abs/1912.01703**.
URL: *http://arxiv.org/abs/1912.01703*

[102] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. and Duchesnay, E. [2012], 'Scikit-learn: Machine learning in python', *CoRR* **abs/1201.0490**.
URL: *http://arxiv.org/abs/1201.0490*

[103] Peemen, M., Setio, A., Mesman, B. and Corporaal, H. [2013], Memory-centric accelerator design for convolutional neural networks, pp. 13–19.

[104] Ramachandran, P., Zoph, B. and Le, Q. V. [2017], 'Searching for activation functions', *CoRR* **abs/1710.05941**.
URL: *http://arxiv.org/abs/1710.05941*

[105] Ren, H. [2014], A brief introduction on contemporary high-level synthesis, *in* '2014 IEEE International Conference on IC Design Technology', pp. 1–4.

[106] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C. and Fei-Fei, L. [2015], 'Imagenet large scale visual recognition challenge', *International Journal of Computer Vision* **115**(3), 211–252.
URL: *https://doi.org/10.1007/s11263-015-0816-y*

[107] Sayal, A., Fathima, S., Nibhanupudi, S. T. and Kulkarni, J. P. [2021], 'Compac: Compressed time-domain, pooling-aware convolution cnn engine with reduced data movement for energy-efficient ai computing', *IEEE Journal of Solid-State Circuits* **56**(7), 2205–2220.

[108] Scholl, A. and Becker, C. [2006], 'State-of-the-art exact and heuristic solution procedures for simple assembly line balancing', *European Journal of Operational*

Research **168**(3), 666–693. Balancing Assembly and Transfer lines.
URL: *https://www.sciencedirect.com/science/article/pii/S0377221704004795*

[109] Shafiee, A., Nag, A., Muralimanohar, N., Balasubramonian, R., Strachan, J. P., Hu, M., Williams, R. S. and Srikumar, V. [2016], Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars, *in* '2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)', pp. 14–26.

[110] Shahshahani, M. and Bhatia, D. [2022], Ppa based cnn architecture explorer, *in* '2022 IEEE 13th Latin America Symposium on Circuits and System (LASCAS)', pp. 01–04.

[111] Shanshi Huang, Shimeng Yu, X. P. [2019], 'Neurosim v1'.

[112] Shen, L., Margolies, L., Rothstein, J., Fluder, E., McBride, R. and Sieh, W. [2019], 'Deep learning to improve breast cancer detection on screening mammography', *Scientific Reports* **9**, 1–12.

[113] Shipley, C. and Jodis, S. [2003], Programming languages classification, *in* H. Bidgoli, ed., 'Encyclopedia of Information Systems', Elsevier, New York, pp. 545–552.
URL: *https://www.sciencedirect.com/science/article/pii/B0122272404001386*

[114] Simonyan, K. and Zisserman, A. [2014], Two-stream convolutional networks for action recognition in videos, *in* Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence and K. Weinberger, eds, 'Advances in Neural Information Processing Systems', Vol. 27, Curran Associates, Inc.
URL: *https://proceedings.neurips.cc/paper*$_f iles/paper/2014/file/00ec53c4682d36f5c4359f4$
$Paper.pdf$

[115] Simonyan, K. and Zisserman, A. [2015], 'Very deep convolutional networks for large-scale image recognition', *arXiv preprint arXiv:1409.1556* .

[116] Soutner, D. and Müller, L. [2013], Application of lstm neural networks in language modelling, *in* I. Habernal and V. Matoušek, eds, 'Text, Speech, and Dialogue', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 105–112.

[117] Symons, A., Mei, L., Colleman, S., Houshmand, P., Karl, S. and Verhelst, M. [2022], 'Towards heterogeneous multi-core accelerators exploiting fine-grained scheduling of layer-fused deep neural networks'.

[118] Sze, V., , Chen, Y.-H., Yang, T.-Y. and Emer, J. [2020], *Efficient Processing of Deep Neural Networks*, Synthesis Lectures on Computer Architecture, Morgan and Claypool.

[119] Sze, V., Chen, Y., Yang, T. and Emer, J. S. [2017], 'Efficient processing of deep neural networks: A tutorial and survey', *CoRR* **abs/1703.09039**.
URL: *http://arxiv.org/abs/1703.09039*

[120] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A. [2014], 'Going deeper with convolutions'.

[121] Tang, T., Li, S., Nai, L., Jouppi, N. and Xie, Y. [2021], Neurometer: An integrated power, area, and timing modeling framework for machine learning accelerators industry track paper, *in* '2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)', pp. 841–853.

[122] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. and Polosukhin, I. [2017], 'Attention is all you need', *Advances in neural information processing systems* **30**.

[123] Weber, O., Min, D., Villaret, A., Park, J., Lee, I., Vandenbossche, E., Kim, D., Yun, J., Park, J., Lee, M., Kang, J., Lee, H., Choi, Y., Kim, I., Kim, J., Kedar, D., Janardan, D. K., Haendler, S., Elghouli, S., Puget, S., Bernicot, C., Bernard, E., Wacquant, F., Nimsgern, F., Choi, J., Maeda, S., Lee, J. and Arnaud, F. [2022], 18nm fdsoi enhanced device platform for ulp/ull mcus, *in* '2022 International Electron Devices Meeting (IEDM)', pp. 27.2.1–27.2.4.

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

[125] Wu, Y. N., Sze, V. and Emer, J. S. [2020], An architecture-level energy and area estimator for processing-in-memory accelerator designs, *in* '2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)', pp. 116–118.

[126] Yang, M., Wang, S., Bakita, J., Vu, T., Smith, F. D., Anderson, J. H. and Frahm, J.-M. [2019], Re-thinking cnn frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge, *in* '2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)', pp. 305–317.

[127] Yi, Q., Sun, H. and Fujita, M. [2021], 'FPGA based accelerator for neural networks computation with flexible pipelining', *CoRR* **abs/2112.15443**.
URL: *https://arxiv.org/abs/2112.15443*

[128] Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B. and Cong, J. [2015], Optimizing fpga-based accelerator design for deep convolutional neural networks, pp. 161–170.

[129] Zhang, J. X., Yordanov, B., Gaunt, A., Wang, M. X., Dai, P., Chen, Y.-J., Zhang, K., Fang, J. Z., Dalchau, N., Li, J. et al. [2021], 'A deep learning model for predicting next-generation sequencing depth from dna sequence', *Nature communications* **12**(1), 4387.

[130] Zhang, K., Zhang, Z., Li, Z. and Qiao, Y. [2016], 'Joint face detection and alignment using multitask cascaded convolutional networks', *IEEE Signal Processing Letters* **23**(10), 1499–1503.

[131] Zhao, Y., Xia, X. and Togneri, R. [2019], 'Applications of deep learning to audio generation', *IEEE Circuits and Systems Magazine* **19**(4), 19–38.

[132] Zhao, Y., Yu, Q., Zhou, X., Zhou, X., Li, X. and Wang, C. [2016], Pie: A pipeline energy-efficient accelerator for inference process in deep neural networks, *in* '2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)', pp. 1067–1074.

[133] Zhao, Z., Kwon, H., Kuhar, S., Sheng, W., Mao, Z. and Krishna, T. [2019], mrna: Enabling efficient mapping space exploration for a reconfiguration neural accelerator, *in* '2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)', pp. 282–292.

[134] Zhong, G., Dubey, A., Tan, C. and Mitra, T. [2018], 'Synergy: A HW/SW framework for high throughput cnns on embedded heterogeneous soc', *CoRR* **abs/1804.00706**.
URL: *http://arxiv.org/abs/1804.00706*

[135] Zhou, G., Zhou, J. and Lin, H. [2018], Research on nvidia deep learning accelerator, *in* '2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)', pp. 192–195.

[136] Zhou, H. and Liu, C. [2014], 'Task mapping in heterogeneous embedded systems for fast completion time', *2014 International Conference on Embedded Software (EMSOFT)* pp. 1–10.
URL: *https://api.semanticscholar.org/CorpusID:14574851*

[137] Zhou, P., Ma, Y., Zhou, Q. and Hong, X. [2007], Thermal effects with leakage power considered in 2d/3d floorplanning, pp. 338 – 343.

Appendices

Appendix A

Gemini 1 Testchip P18

During the project of implementing Gemini-1 on the new p18 technology, which underwent testing for two years, several tasks were completed, including:

- Delivery of the gate netlist of Gemini-1 test-chip accelerator: it includes some modifications that has to be done in order to test the new STMicroelectronics technology. The main modification was to implement an auto test that consists of loading an image of a handwritten digit from the MNIST [78] data-set and executes a neural network that recognize the digit and check if the result is correct.
- Use synthesized latches to model RAMS unavailable in this technology.
- Add a design to test the logic (design for testability).
- Simulations:
  - Functional at C++ level.
  - Functional at RTL level.
  - Functional at gates level.
  - Functional at back-annotated gates level after placement and routing.
  - Generate test vectors by automatic test pattern generation tools for the design for testability to test stuck at and transition faults.
- Follow the test of the chip:
  - Functional at different voltages and frequencies.
  - Stuck at faults.
  - Leakage and power consumption.
  - Yields.
  - Identifications of bugs on the circuits and technology issues.

Appendix B

Dynamic program to Optimize the Latency Lat_2 for a Given Throughput Using States Representation in the Fixed Hardware Scenario

The algorithm that we are describing is equivalent to the one presented in Subsection 6.7.2. The inputs and outputs remain unchanged.

$M(i,g,h), (i,g,h) \in [0,n-1] \times [0,\ell-1] \times [0,\ell-1]$ is defined as the minimum latency of the pipeline system from G_0 to G_i assuming that G_i processes the layers L_g to L_h and $\forall i', 0 \leq i' < i$, $G_{i'}$ processes its layers in less than the upper bound period $P^\star$ cycles (i.e $p_{i',\pi} \leq P$). $M(i,g,h)$ is computed as follows:

- $M(i,g,h) = +\infty$ if:
  - $p_{i,[g,h]} > P^\star$
  - $\widehat{K}_{i,[g,h[} > K_i$ if $g \neq h$ and $s_j > K_i$ otherwise.
  - $l - j > i > j$. This condition correspond to mapping properties (presented in Section 6.2.4);
- $M(0,g,h) = p_{0,[g,h]}$;
- $M(i,g,h) = min_{j \in [0,g-1]}(M(i-1,j,g-1)) + p_{i,[g,h]}$ for $i > 0$.

The latency Lat_2 of the system is then given by $\min_{j \in [0,\ell-1]} M_{n,j,\ell-1}$. The mapping π is obtained by backtracking which layer set $[L_g, L_h], (g,h) \in [0,\ell-1]$ was minimizing $M(i,g,h), \forall i \in [0,n-2]$. Finally, the throughput is obtained by $T(\pi) = \min_{i \in [0,n-1]} \frac{1}{p_{i,\pi}}$

Appendix C

MPAR Choice For Gemini Pipeline

As a reminder, Equation 5.2 in Section 5.5.1 shows that the latency, when the NN has convolution layers, is sensitive to $MPAR$. Intuitively, it is evident that the convolution latency remains unaltered if $M \times k \leq MPAR < (k+1)M$, with, k being a constant integer. Consequently, $MPAR$ reaches an optimum when M is a multiple of $MPAR$. Therefore, the selection of $MPAR$ must be made from the divisors of M for each layer. Area and power do not have a particular behavior that could limit the $MPAR$ choice.

Appendix D

Dynamic Program to Optimize φ for a Given Throughput Using States Representation in the Non-fixed Hardware Scenario

The inputs and outputs of the algorithm are identical to the ones described in Section 7.6. For clarity purposes and without loss of generality φ will be considered as the NPU area a_{NPU}. Let us define $M(g,h), (g,h) \in \mathcal{E} = \{[0,\ell-1]^2 g \leq h\}$ the area of all the pipeline system processing the layers L_0 to L_h assuming that $[L_g, L_h]$ are processed on the last NPU. This NPU has $\widehat{N}([g,h], P^\star)$ PEs and the associated RAM has $\widehat{K}([g,h])$ KBs). Additionally, all layers $[L_0, L_g]$ are processed on NPUs that complete their processing within $P^\star$ cycles. $M(g,h)$ can be computed as follows:

- For $g = 0$ and $h \leq \ell - 1$: $M(0,h) = a_{0,NPU}(\pi_{0,h})$ where π_0 is the allocation where $\forall (j,j') \in [0,h]^2, \pi(j) = \pi(j') = 0$. The number of PEs of NPU G_0 is equal to $\widehat{N}([0,h], P^\star)$.
- For $0 < g \leq h - 1$ and $h \leq \ell - 1$:
  $M(g,h) = \min_{j \in [0,g-1]}(M(j, g-1)) + a_{i,NPU}(\pi_{g,h})$ where $\pi_{g,h}$ is the allocation where $\forall (j,j') \in [g,h]^2, \pi(j) = \pi(j') = i$. The number of PEs of the NPU G_i is equal to $\widehat{N}i([g,h], P^\star)$. The index i here is arbitrary, as the number of PEs is not fixed.

The minimum NPUs area of the system processing all the layers $[L_0, L_\ell - 1]$ is given by $\min_{g \in [0,\ell-1]} N(g, \ell - 1)$. The NPUs used, their processing elements number and the layers' allocation are obtained backtracking the minimums at each step. The system by construction produces one inference in less than $P^\star$ cycles.

Appendix E

Optimizing the Latency Lat_2 for a Given Throughput in The Non-fixed Hardware Scenario Using Gemini NPUs

In this paragraph, we focus on optimizing, for a given throughput, the latency in the case where *fmaps* are processed sequentially (as if we were using a single NPU). As reminder, the latency Lat_2 is computed summing the execution time of all NPUs: $Lat_2(\pi) = \sum_{i=0}^{n-1} p_{i,\pi}$. We illustrate this study on the MobileNet network (Figure 2.13). We observe that for high throughput (small load) constraints, one single NPU can not meet this constraint (below 27 000 cycles). However, as soon as the single-NPU solution meets the throughput constraint, it is chosen by the algorithm. This is logical because maximizing latency occurs when all layers are placed on a single NPU (intuited in Subsection 6.3.2) to avoid the cycle cost of adding additional NPUs. Thus, for optimizing the latency, the pipeline is useful only if the single NPU solution cannot achieve its throughput objective. Note that the stair-shaped curves are a result of using ceiling operations (Subsection 7.7.1)

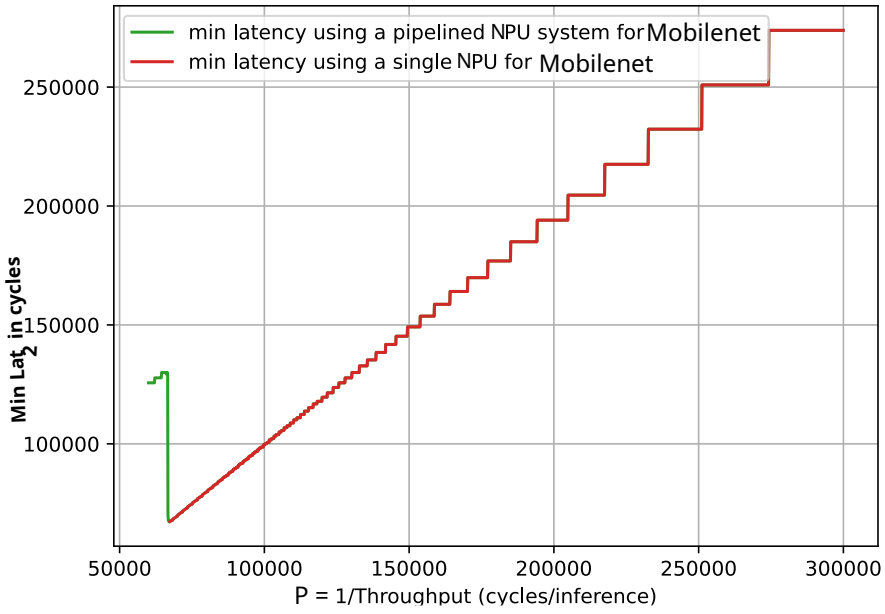

Fig. E.1 *Latency under throughput constraints on Mobilenet*