EPFL

Thèse n° 9884

Hardware-Software Co-design for Improved Resource Utilization in DNN Accelerators

Présentée le 30 janvier 2023

Faculté des sciences et techniques de l'ingénieur
Laboratoire de traitement des signaux 4
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Ahmet Caner YÜZÜGÜLER

Acceptée sur proposition du jury

Prof. A. Roshan Zamir, président du jury
Prof. P. Frossard, directeur de thèse
Prof. L. Benini, rapporteur
Dr C. Malossi, rapporteur
Prof. D. Atienza, rapporteur

École polytechnique fédérale de Lausanne

2023

"Science is the poetry of reality."
— Richard Dawkins

To my beloved family...

Acknowledgements

This thesis is the culmination of a journey that was long and arduous, but ultimately, incredibly rewarding. It would not have been possible without the support and guidance of my colleagues and friends, and for that, I am eternally grateful.

My most sincere thanks go to my advisor, Professor Pascal Frossard, for his ceaseless support and guidance. Besides his vast technical contributions, Pascal's remarkable leadership style has made this thesis possible. Even during the most difficult moments of my Ph.D., he managed to find a way to motivate me to move forward. Every piece of feedback that I received from him was on point, considerate, and carefully crafted in a way that reassured me that I was in the right path. Therefore, I consider myself extremely fortunate to have been Pascal's student, and I will definitely miss working under his supervision and in the superb working environment that he has created in LTS4.

Next, I would like to thank the members of my thesis committee, Professor Amir Zamir, Professor David Atienza Alonso, Professor Luca Benini, and Dr. Cristiano Malossi for their time to review this thesis and for the valuable feedback that they provided. I would also like to thank Professor Babak Falsafi for supervising my research that constitutes the backbone of the third chapter in this thesis. I would like to send a special thanks to Professor Elif Vural, who supervised my first research project in LTS4, and inspired me to pursue a career in research. My deepest gratitude goes to my mentor, Professor Giovanni De Micheli, for his support throughout my Ph.D. I would also like to acknowledge the help of the Hasler Foundation in this thesis, for funding my research and creating an opportunity for me to achieve my goals.

The greatest sources of support throughout my Ph.D. have been my wonderful flatmates and closest friends, Chiara and Nick. Not only did we share a cozy and welcoming home, but we also shared similar goals and faced similar challenges. Together, we navigated the difficulties of the COVID-19 lockdowns and it would have been awfully harder without their presence. The time we spent together will always hold a special place in my memories and I look forward to continuing our friendship long after the completion of our studies.

I am thankful to all of my past and current colleagues at LTS4, including Mireille, Tolis, Beril, Isabela, Ádám, Clément, Guille, Ortal, Yamin, Harshitha, Javier, William, Jelena, and Caroline. With them, I had many fruitful discussions, delightful lunch breaks, and wonderful winter retreats. I would like to extend my special thanks to Nikos, who greatly contributed to

the fourth and fifth chapters of this thesis. A special thanks also goes to Abdellah, who kindly helped me to translate the abstract of this thesis to French.

I am deeply grateful to my former colleagues from PARSA, Yunho, Mario, Arash, Sid, Ognjen, Simla, and Canberk. They have been an integral part of my Ph.D. and their help and support have been invaluable. I will always treasure the memories we shared together. My special thanks go to Yunho, Mario, and Canberk, who have made huge contributions to the third chapter of this thesis. Their expertise and contributions have been critical to the success of my research.

I feel lucky to have a fantastic circle of Turkish friends in Lausanne, Kağan, Arman, Didem, Doruk, and Sinem, with whom I shared countless memories and experiences while enjoying the comfort of speaking in my mother tongue.

Last but not least, I must express my deepest gratitude to my beloved family. My brother, Can, and my sister-in-law, Ayça, have been constant sources of support for me over the last years. My beloved newborn nephew, Ege, brought immeasurable joy and excitement to my days spent writing this thesis. I wish him a long and fulfilling life, full of success and happiness. Finally, my deepest appreciation goes to my parents, Mutlu and Kudret, who have dedicated their entire lives to ensuring our happiness and well-being. Their unwavering support has been the driving force behind my Ph.D. journey, and my heart is filled with an overwhelming sense of gratitude for everything they have done for me.

Lausanne, 13 January 2023 A. C. Yüzügüler

Abstract

Deep neural networks (DNN) have become an essential tool to tackle challenging tasks in many fields of computer science. However, their high computational complexity limits their applicability. Specialized DNN accelerators have been developed to accommodate the high computational complexity of DNN inference, but the mismatch between the accelerators and DNN models prevents unlocking their full potential.

In this thesis, we address the mismatch between accelerators and DNN models from both hardware and software perspectives. First, we investigate one of the most widely used architectures in DNN accelerators, i.e., systolic arrays, and identify the leading cause of underperformance in DNN inference, namely dimension mismatches between arrays and DNN layers. We analyze the characteristics of today's popular DNN models, perform an extensive design-space exploration and propose a novel scale-out systolic array architecture that maximizes the effective throughput (i.e., FLOPS per second) for a given set of target DNN workloads.

Then, we go beyond the limits of what can be achieved with hardware optimization and focus on optimizing DNN architectures to improve the resource utilization at the target accelerators. To that end, we study differentiable neural architecture search frameworks, which automate the creation of DNN architectures using efficient gradient-descent optimizers. We introduce a novel computational model for the utilization of systolic arrays and propose a novel utilization-aware neural architecture search framework. The proposed framework is capable of creating DNN models with improved resource utilization on target accelerators, which permits to perform DNN inference more efficiently and faster.

The existing neural architecture search framework searches for channel dimensions of a DNN model in a fixed search space, which requires to manually design complex search spaces. However, designing a search space is a nontrivial task that requires heuristics and domain expertise, which undermines the effectiveness and practicality of neural architecture search. To eliminate the necessity to predefine search spaces, we propose a flexible channel masking method, which dynamically adjusts the search space based on the progress of architecture search. We introduce the differentiable neural architecture search framework that uses the

flexible channel masking method, which obviates the need for manually designing a search space. We demonstrate through extensive experiments that the proposed framework significantly reduces the search time and memory requirements compared to the existing neural architecture search framework with fixed search spaces.

Overall, this thesis proposes hardware and software co-design techniques to improve the performance of DNN inference. We demonstrate that the proposed scale-out systolic array architecture combined with DNNs optimized using the proposed neural architecture search frameworks exhibits significantly higher resource utilization. Consequently, the proposed methods enable faster and more efficient DNN inference, improving the effectiveness of DNN applications on resource-constraint platforms.

Keywords: deep neural networks, systolic arrays, neural architecture search

Résumé

Les réseaux de neurones profonds (DNN) sont devenus des outils essentiels pour aborder les problèmes exigeants et complexes dans plusieurs domaines de la science de l'information. Cependant, la complexité computationnelle de ces modèles profonds limite leur champs d'application. Des accélérateurs spécialisés pour les DNN ont été développés pour répondre à ce problème de complexité de calcul des DNN, mais l'inadéquation entre les accélérateurs et les DNN nous empêche de tirer profit de l'entier du potentiel des accélérateurs.

Dans cette thèse, nous étudions deux perspectives de cette inadéquation, entre les accélérateurs et les modèles du DNN, qui peuvent se manifester sous forme de problèmes hardware ou software. Premièrement, nous investiguons une des architectures les plus utilisées parmi les accélérateurs de DNN, i.e, systolic arrays, et nous identifions la cause fondamentale de la sous-performance durant l'inférence des DNN, qui est l'inadéquation entre les dimensions des arrays et celles des couches des DNN. Nous analysons les caractéristiques des modèles DNN les plus populaires, effectuons une exploration extensive du design-space et proposons une méthode appelée "scale-out systolic array" qui maximise le débit effectif (i.e., FLOPS per second) pour un certain ensemble de charge de travail d'un DNN.

Deuxièmement, nous dépassons les limites de ce que nous pouvons obtenir par une optimisation des hardwares, en focalisant sur l'optimisation des architectures des DNN qui améliorent l'utilisation des ressources d'un accélérateur hardware. A cet effet, nous étudions des méthodes de recherche qui automatisent la création des architectures de DNNs en utilisant une optimisation efficace par descente de gradient. Nous introduisons un nouveau modèle de calcul pour l'utilisation des "systolic arrays" et proposons un nouveau cadre intitulé "utilization-aware neural architecture search". Le cadre proposé est capable de créer des modèles de DNNs avec des utilisations de ressources améliorées pour des accélérateurs cibles, ce qui permet d'effectuer une inférence plus efficace et rapide.

Le cadre existant des "neural architecture search" cherche à trouver les dimensions des channels d'un modèles DNNs dans un espace de recherche fixé, ce qui nécessite un choix manuel d'un espace de recherche complexe. Néanmoins, le choix d'un espace de recherche

représente une tâche non triviale qui exige des heuristiques et une expertise, ce qui affaiblit l'efficacité et la praticité du neural architecture search. Pour éliminer la nécessité de prédéfinir des espaces de recherche, nous proposons une méthode flexible de masquage de channel, qui ajuste dynamiquement l'espace de recherche en se basant sur l'evaluation de la function de recherche. Nous introduisons le cadre nommé ; "differentiable neural architecture search", qui utilise la méthode flexible du masquage, qui évite le besoin d'un choix manuel d'espaces de recherche. Nous démontrons à travers différentes expériences et en comparant avec différents modèles que le cadre proposé diminue le temps de recherche et aussi les exigences en termes de mémoire de calcul.

En guise de conclusion, cette thèse propose des techniques de conception conjointe de hardware et software qui améliorent les performances de l'inférence des DNNs. Nous démontrons qu'optimiser les DNN en utilisant le cadre proposé "differentiable neural architecture search" et en le combinant avec la méthode scale-out systolic array, permet une utilisation très élevée des ressources. Par conséquent, la méthode suggérée permet une inférence efficace et rapide des DNN, ce qui améliore l'efficacité de l'application des DNNs sur les plateformes à ressources limitées.

Mots clefs : Réseaux de neurones profonds (DNN), systolic arrays, neural architecture search

Contents

List of Figures

List of Tables

1 Introduction

Deep neural networks (DNN) have unlocked an unprecedented potential in solving a wide range of challenging problems in various fields of science and engineering. The success of DNNs is mostly attained due to their capability to automatically extract meaningful features from large amounts of data whereas classical algorithms rather rely on human heuristics and engineering efforts to come up with useful features. With the ever-growing amounts of data collected every day, DNNs will undoubtedly remain for the foreseeable future as an indispensable tool for scientists and engineers to learn effective representations of information in various application domains.

While the principles of deep learning date back to mid-20th century, the recent increase in the popularity of DNNs and their widespread adoption in numerous domains coincide with the advents in parallel computing in the modern era. With their high computational complexity, DNN applications have initially resorted to hardware platforms such as field programmable gate arrays (FPGA) [Farabet et al., 2011] and graphic proccesing units (GPU) [Krizhevsky et al., 2012]. However, the immense volume of DNN workloads in datacenters and mobile devices has encouraged researchers to develop specialized hardware architectures for DNNs, which has led to the emergence of highly efficient and powerful DNN accelerators [Jouppi et al., 2017; Hock, 2019; Liao et al., 2019].

1.1 Multi-pod Systolic Architectures

One of the most widely used specialized hardware architectures in DNN accelerators is *systolic arrays*. In systolic arrays, processing elements are typically organized in a two-dimensional grid, where the adjacent processing elements can efficiently share data with each other through direct links. In each cycle, processing elements retrieve data from a set of adjacent neighbors, perform a multiply-and-accumulate operation, and pass the data and intermediate results to another set of adjacent neighbors [I]. While systolic arrays can achieve exceptional power

[I] The name "systolic" comes from the rhythmical pattern of data movement due to its resemblance to a heartbeat.

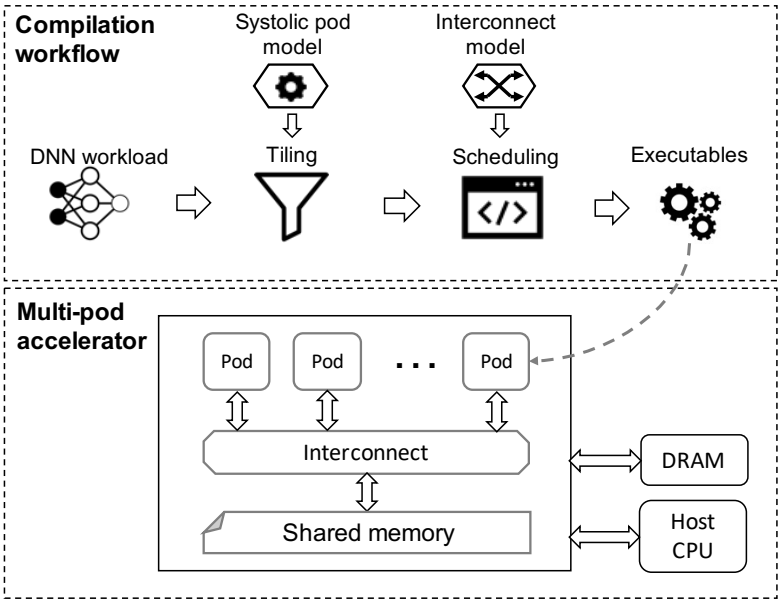


Figure 1.1: General overview of a multi-pod systolic architecture.

efficiency thank to this unique pattern of data movement, they heavily suffer from underutilization due to mismatches between the arrays and DNNs, which prevents the effective usage of available computational resources. For instance, the average resource utilization while processing a DNN in the latest version of a TPU, which is a systolic-array based accelerator, is only 33% [Jouppi et al., 2021], underlining the severity of underutilization in DNN accelerators.

The leading source of underutilization in systolic arrays originates from mismatches between arrays and DNN models [Jouppi et al., 2017; Kung et al., 2019]. When model dimensions (e.g., number of channels and filters) is smaller than the array dimensions, the excess rows and/or columns in the systolic array become idle, resulting in underutilization. A trivial solution to this problem would be to reduce the array dimensions of systolic arrays. However, data reuse in systolic arrays is tightly correlated with the array size. Thus, while small systolic arrays may offer a remedy to the underutilization problem, they would suffer from low data reuse and consequently from poor power efficiency. This intricate relation between power efficiency and utilization in systolic arrays poses a challenging optimization problem that must be carefully addressed while designing accelerators with systolic arrays.

Because the underutilization problem places an upper bound on the size of systolic arrays, architectures with multiple systolic arrays have emerged [Kung et al., 2019], which we simply refer to as *multi-pod* architectures throughout this thesis. These architectures either partition a single DNN workload and run different partitions in parallel pods, or they rely on task-level parallelism and run multiple DNN workloads in parallel pods [Jouppi et al., 2021; Baek et al., 2020].

Figure 1.1 depicts the overall diagram of a multi-pod inference accelerator, where each *pod*

includes a systolic array and necessary buffers and peripherals. The pods are connected to an on-chip shared memory through an interconnect network. DNN models and intermediate data are typically stored in on-chip memory for temporal reuse. The accelerators typically have interfaces to off-chip DRAM to store data when the on-chip memory size is not sufficient and a host CPU to receive instructions. In order to map workloads to multiple systolic arrays, DNN layers are first partitioned into tile operations of sizes that match a pod's array dimensions. Then, a scheduler optimizes the mapping of tile operations to pods to maximize parallelism and throughput.

Achieving scalability and high utilization in multi-pod accelerators requires design optimizations in various aspects. First of all, systolic array dimensions should be carefully selected to find a sweet-spot between power efficiency and utilization for the target DNN workloads. Second, the pods should be connected to each other through an interconnect topology that allows sufficient bandwidth with high efficiency. Third, the compiler should be capable of tiling, mapping and scheduling DNN workloads in a way to make use of maximum number of pods in parallel. These design requirements and constraints pose a significant challenge while designing multi-pod accelerators. In this thesis, we address these challenges and propose techniques to improve the resource utilization of multi-pod systolic architectures.

1.2 Hardware-aware Neural Architecture Search

Despite the improvements in hardware architectures, the performance of DNN accelerators varies based on the computational characteristics of the DNN workloads. Therefore, DNNs must also be designed or optimized to improve their compatibility with the target compute platforms. Many aspects of DNNs such as layer types and dimensions have considerable impact on their computational characteristics. However, designing DNNs while taking their computational characteristics into account is a challenging task for developers.

To automate the task of designing DNNs, researchers have proposed *neural architecture search*, which aims to replace design heuristics with optimization algorithms. Such neural architecture search frameworks are often *hardware-aware* and optimizes the DNNs not only for the accuracy at the given task but also for their computational cost at target platforms. The early versions of neural architecture search frameworks have resorted to evolutionary algorithms or reinforcement learning, which require a substantial computational cost to solve this multi-objective optimization problem. However, recent improvements such as *differentiable* neural architecture search frameworks have significantly reduced the cost of design optimization, which becomes practical for a wide range of applications.

Figure 1.2 depicts the main components of a hardware-aware differentiable neural architecture search framework. We can divide a neural architecture search framework into three phases, namely initialization, optimization, architecture evaluation. The initialization phase includes designing a search space, which contains the set of variables that defines possible candidate architectures. Based on these variables, one obtains an overparameterized DNN model, which

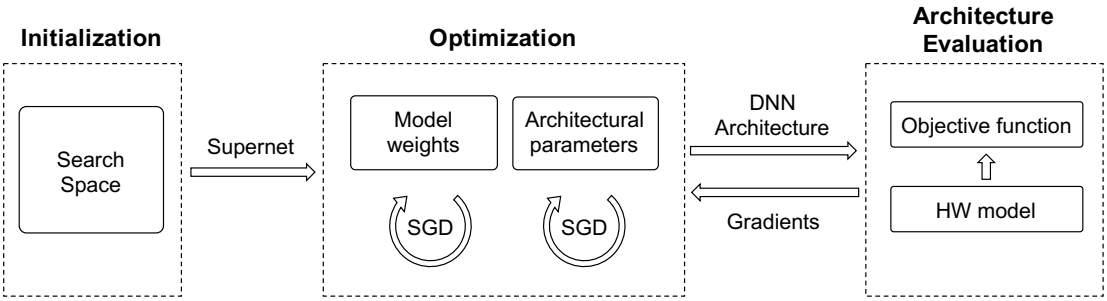


Figure 1.2: General overview of a hardware-aware differentiable neural architecture search.

is also known as *supernet*. The supernet contains both trainable *model weights* that define its functionality as well as a set of *architecture parameters* that simulate different architectural choices. During the optimization stage, the supernet's model weights and architectural parameters are jointly optimized using gradient descent. The gradients for these optimization steps are obtained by calculating a loss value using a batch of training data. Finally, in case of a hardware-aware neural architecture search, hardware-related terms are included in the objective function (e.g., latency), which are typically estimated using a hardware model.

Estimating the hardware-related metrics accurately and efficiently is, however, critical to obtain DNNs that perform well on target platforms. While simplistic hardware models such as FLOPS per second or roofline [Williams et al., 2009] might give rough estimates about the latency or throughput of selected DNNs, they do not capture the nonlinear runtime characteristics of systolic architectures. This leads to discrepancies between estimated and ground-truth values, resulting in DNNs that do not perform well on the target platforms. To overcome the discrepancy between estimated and ground-truth values, researchers have proposed to use *black-box* models, in which measurements from physical devices are stored in lookup tables and retrieved during the architecture search. However, such black-box models are not differentiable; thus, they can not be effectively used in differentiable neural architecture search frameworks that require gradient calculations. Overall, the existing methods to estimate hardware-related metrics are unfortunately either inaccurate for systolic architectures or incompatible with the efficient neural architecture search frameworks.

Besides the challenges in estimating hardware-related metrics, another obstacle to develop an effective and efficient hardware-aware neural architecture search relates to the quality of their search space. Because the search space defines all possible DNN configurations, having a poor search space design may lead to finding suboptimal DNNs. A trivial solution to this problem might be to use an excessively large search space to ensure that it covers the optimal DNN configurations, but this would introduce large amounts of redundant computation, increasing the computational cost of the search. As a result, the existing frameworks rely on domain experts to carefully design an effective search space, which defeats the purpose of automating the design of DNNs. Due to these limitations inherent to their search space, existing frameworks are not capable of finding optimal DNNs that perform well at target

platforms. In this thesis, we address the challenges in estimating hardware-related metrics for systolic architectures and limitations inherent to the search space of neural architecture search frameworks.

1.3 Thesis Outline

This thesis addresses the underutilization problem in DNN inference accelerators and proposes solutions from both hardware and software perspectives. We first study the fundamental design trade-offs in systolic arrays and perform a design space exploration for server-grade accelerators targeting convolutional and moderately-sized Transformer models. Then, we study three key design aspects in multi-pod systolic architectures and identify the optimal array granularity, interconnect topology, and tiling strategies. Based on our findings from our design space exploration and analysis on these three key design pillars, we propose a novel scale-out systolic array architecture. We demonstrate that the proposed architecture achieves higher resource utilization and effective throughput in DNN inference workloads compared to the baseline designs.

Then, we turn our focus towards the optimization of DNN models to improve their resource utilization on target inference platforms. Using our insights from the scale-out systolic array architecture, we develop an analytical model for resource utilization on systolic architectures and propose a smooth approximation that makes it suitable for effective optimization methods such as differentiable architecture search. We then develop the novel utilization-aware differentiable neural architecture search framework, which allows finding DNNs that exhibit high resource utilization on target inference platforms. Our experiments on popular vision tasks such image classification show that the DNNs found by the proposed framework achieve higher accuracy and/or shorter inference latency thanks to more effective usage of computational resources at target platforms.

To improve the efficiency and practicality of the proposed utilization-aware neural architecture search framework, we finally address the limitations inherent to the search space of the neural architecture search. We propose a flexible search space, which allows finding the optimal channel dimensions for DNNs without the need for manually designing a search space, and a novel dynamic channel allocation mechanism that enables modifying the dimensions of a supernet efficiently in neural architecture search. We show that the proposed flexible search space and the dynamic channel allocation mechanism obviate the need for manually designing search spaces for channel dimensions and improves the efficiency of neural architecture search compared to the existing frameworks with fixed search spaces.

In short, this thesis makes the following contributions:

- We analyze the workload characteristics of widely used CNN and Transformer models and perform a design space exploration to find the optimal array dimensions in systolic

architectures.

- We study various interconnect topologies and identify the ideal topology to connect large numbers of systolic pods in DNN accelerators.
- We study tiling and scheduling methods for DNN workloads and propose a novel tiling strategy for multi-pod systolic architectures.
- We propose the novel scale-out systolic array architecture and show that it outperforms the existing architectures in DNN inference workloads.
- We develop an analytical model for resource utilization in systolic arrays and propose a smooth approximation to make this model differentiable and amenable to effective optimization.
- We propose a novel utilization-aware differentiable neural architecture search framework that improves the resource utilization of DNNs on target accelerators.
- We introduce a flexible search space for channel dimensions in differentiable neural architecture search and we propose a novel dynamic channel allocation mechanism to improve the efficiency of differentiable neural architecture search.

The rest of this thesis is organized as follows: We first introduce the prior work on DNN accelerators and neural architecture search in Chapter 2. Then, we explain the key design considerations for multi-pod systolic architectures and propose the scale-out systolic array architecture in Chapter 3. e introduce our analytical model for resource utilization and propose the utilization-aware differentiable neural architecture search framework in Chapter 4. Then, we elaborate on the flexible search space for channel dimensions and the dynamic channel allocation mechanism in Chapter 5. Finally, we provide a summary of the contributions that we make in this thesis and discuss potential future directions in Chapter 6.

2 Related Work

To accommodate the high computational complexity of DNN inference, a myriad of hardware and software techniques have been proposed. In this chapter, we first give background information and elaborate on the prior work on the specialized hardware for DNN inference, and then we discuss the prior work on software optimizations and co-design techniques with a focus on hardware-aware neural architecture search.

2.1 DNN Accelerators

DNN workloads have been deployed in a wide range of compute platforms from general-purpose CPUs to specialized accelerators. We can categorize these platforms based on their processing units into four groups: CPU, GPU, FPGA, and ASIC. General-purpose CPUs require low development cost thanks to their programmability; thus, they are in use of DNN inference for certain datacenter applications such as Facebook [Hazelwood et al., 2018]. However, CPUs exhibit high latency and limited throughput and power efficiency due to their long execution pipeline, deep memory hierarchy and complex control flow, which encourages the development and deployment of more specialized platforms for DNN workloads.

Thanks to their massive parallel execution capabilities, GPUs today are widely used in processing DNN workloads. Recent GPUs are equipped with specialized tensor cores [Choquette et al., 2021] and support custom data encoding formats such as BFloat16 [Kalamkar et al., 2019] to further improve their performance and power efficiency on DNN workloads. While their exceptional throughput with such customizations have made GPUs indispensable for DNN training, their long execution pipeline still incurs long latencies as in CPUs, which undermines their applicability in DNN inference for latency-critical online services.

To further improve the efficiency and performance of DNN inference, some services use the FPGA nodes in cloud services [Fowers et al., 2018]. Thanks to their reconfigurability capability, FPGAs enable building execution pipelines that are highly specialized to efficiently process DNN inference [Xuechao Wei et al., 2017; Alwani et al., 2016; Farabet et al., 2011]. Moreover,

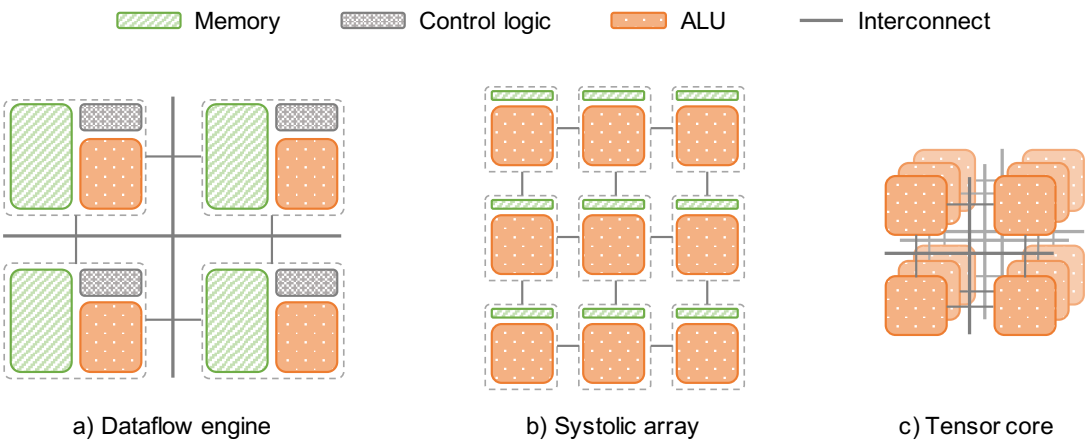


Figure 2.1: Three categories of spatial architectures used in DNN accelerators.

the customized execution pipelines in FPGAs can benefit from more aggressive quantization schemes [Zhang et al., 2018] and efficiently exploit the sparsity [Nurvitadhi et al., 2017] of DNN models. However, the additional logic for reconfigurability incurs significant overhead in FPGAs, placing an upper bound on their maximum clock frequency, arithmetic density and overall performance. Therefore, FGPAs do not offer an effective solution to accommodate the high computational cost of DNN inference.

To maximize throughput and efficiency in DNN workloads, many applications have invested in ASIC solutions. Despite their high cost of development and lack of reconfigurability, ASICs provide 1-2 orders of magnitude higher throughput compared to CPU, GPU, and FPGAs [Jouppi et al., 2017]. These accelerators typically use spatial architectures to simplify data movement between processing elements. Based on the type of spatial architecture that they use, we can group these ASIC solutions into three categories: **Dataflow engines** (e.g., Eyeriss [Chen et al., 2016], Wave DPU [Nicol, 2017]), **Tensor cores** (e.g., NVIDIA A100 [Raihan et al., 2019], Huawei DaVinci [Liao et al., 2019]), and **Systolic arrays** (e.g., Google TPU [Jouppi et al., 2017], Baidu Kunlun [Ouyang et al., 2020]). More details are given below about these spatial architectures, which are illustrated in Figure 2.1.

Dataflow engines typically consist of a grid of processing elements that are connected through a two-dimensional mesh topology, as shown in Figure 2.1a. These architectures aim to maximize both spatial and temporal data reuse by implementing certain mapping and scheduling patterns that are called *dataflow* [Chen et al., 2017]. Thanks to their ability to support various mapping and scheduling patterns, dataflow engines can adapt varying workload characteristics to improve utilization [Lu et al., 2017]. To implement such dataflow schemes, however, dataflow engines typically require a scratchpad memory and control logic in each processing element. Unfortunately, prior work reports that the scratchpad memory and control logic in processing elements incur significant overhead in power consumption and silicon area (up to 90% [Chen et al., 2016]), which places an upper bound on their efficiency and throughput.

In contrast to dataflow engines, tensor cores employ simplistic processing elements without any scratchpad memory or control logic [Raihan et al., 2019], as shown in Figure 2.1c. Instead of implementing configurable dataflow schemes, they typically act as matrix multiplication engines with a fixed dataflow. To benefit from the three-dimensional spatial data reuse inherent to the matrix multiplication operations, the processing elements in tensor cores are often organized in a cube topology [NVIDIA, 2020]. While their simplistic design of the processing elements and their cube topology aim to maximize the throughput in matrix multiplication operations, they require large numbers of memory operations and high bandwidth between processing elements and on-chip memory due to the lack of temporal data reuse, which results in high power consumption and reduced efficiency.

Systolic arrays, which are depicted in Figure 2.1b, achieve superior power efficiency compared to tensor cores as they exploit both spatial and temporal data reuse in DNNs [Kung, 1982]. While performing a matrix multiplication operation, they store the elements of one of the operands in the registers of processing elements and reuse them throughout the operation, reducing the memory access and bandwidth requirements between processing elements and on-chip memory. Thanks to these advantages, systolic arrays have been used in various commercial accelerators [Jouppi et al., 2017; Ouyang et al., 2020].

Because systolic arrays reuse one of the operands temporally while performing a matrix multiplication, their spatial data reuse is limited to two-dimensional data. As a result, unlike the tensor cores that are often designed as a three-dimensional cube, the systolic arrays are typically organized in two-dimensional grid topologies. This fundamental limitation in the topology of systolic arrays has significant implications on their resource utilization. For the same number of processing elements, a systolic array organized in a two-dimensional grid has much larger dimensions than a three-dimensional tensor core. As a result of their large grid dimensions, underutilization due to dimension mismatches becomes a significant problem in systolic arrays.

The severity of the underutilization problem in systolic arrays has attracted considerable research efforts and led to various modifications in commercial accelerators. While the first generation TPUs consist of large and monolithic systolic arrays with an array size of 256×256 [Jouppi et al., 2017], its successors adopted a multi-pod design with smaller systolic arrays of 128×128 [Jouppi et al., 2021]. Likewise, researchers have proposed architectures that consist of many fine-grained systolic arrays with array sizes as small as 8×8[Kung et al., 2019], which greatly sacrifices the power efficiency. However, the question of how to find the optimal array dimensions for systolic arrays for a given set of target DNNs remains unknown. Therefore, in this thesis, we first focus on finding the optimal array granularity in multi-pod systolic array architectures.

The interconnection between the cores of a DNN accelerator pods also plays a critical role in its performance. While prior work has extensively studied the interconnection topologies in the context of dataflow engines, the findings of these studies do not apply to multi-pod systolic

architectures due to former's unique computational characteristics and data movement patterns. Many of the prior work used Mesh [Chen et al., 2017, 2014b; Gao et al., 2019; Shao et al., 2019] or H-tree [Kung et al., 2019; Stevens et al., 2019] topologies to connect the processing elements in DNN accelerators. However, these topologies lack sufficient bisection bandwidth to support a large number of pods in multi-pod systolic architectures. Qin et al. proposed to use a Benes topology due to its high bisection bandwidth [Qin et al., 2020] but this topology suffers from a long round-trip latency, which leads to reduced utilization. Trivial solutions such as Crossbar or Bus are unfortunately not suitable for multi-pod architectures due to their poor scalability. As a result, finding the optimal interconnect topology in multi-pod systolic architectures is also an open research question.

Besides the research efforts in array granularity and interconnect topology, prior work has also explored concurrency as a possible solution to the underutilization problem in systolic architectures. AI-MultiTasking [Baek et al., 2020] has proposed a scheduling strategy to mitigate the memory stalls in multi-pod systolic arrays. PREMA [Choi & Rhu, 2020] has resorted to multi-tenancy to improve the array utilization. Likewise, Equinox [Drumond et al., 2021] has proposed to piggyback DNN training tasks on inference accelerators to claim idle cycles. However, these proposals offer improvements in architectures only with a few pods and do not address the challenges in tiling and scheduling inherent to the architectures with large numbers of pods. Therefore, in this thesis, we also focus on tiling and scheduling DNN inference on multi-pod accelerators that scale up to hundreds of pods, which is critical to unlock the full potential of these accelerators.

To get the best of both systolic arrays and tensor cores, prior work has proposed to integrate these two types of architectures on a single-chip [Guo et al., 2020a]. To enable such integration, Guo et al. proposes micro-architectural modifications in the systolic arrays to make their memory access coalesced. However, these micro-architectural modifications require longer links between processing elements, hindering the arrays' scalability. In this thesis, we adopt a much simpler approach to make the systolic arrays' memory access coalesced without any modifications required in the arrays. We simply devise *skew buffers* in the systolic pods between on-chip memory banks and systolic arrays to transform coalesced memory accesses into the unique data pattern required by the systolic arrays. Therefore, this thesis proposes a simpler and more efficient solution to the limitations inherent to the uncoalesced memory access patterns of systolic arrays than Guo et al.

2.2 Neural Architecture Search

2.2.1 DNN optimization

The development of methods to build DNN architectures that achieve higher accuracies in the given tasks while maintaining low hardware footprint (such as model size and latency [Tan et al., 2019; Wu et al., 2019]) has received considerable attention from researchers. While early

work has introduced design heuristics such as residual connections [He et al., 2016], batch normalization [Ioffe & Szegedy, 2015] and bottleneck layers [Sandler et al., 2018], building DNNs with these techniques requires human expertise and domain knowledge.

To minimize the need for human supervision while building DNNs, researchers have first resorted to model compression and network pruning techniques [Luo et al., 2017; He et al., 2017; Yu & Huang, 2019; Dong & Yang, 2019b]. Likewise, Jha et al. proposed to optimize DNN architectures for the data reuse to improve resource utilization by eliminating the memory bottlenecks [Jha et al., 2020]. Unfortunately, these methods still rely on human supervision for the design of the initial DNN architectures as starting points, which hinders their effectiveness and prevents them from realizing the purpose of automating DNN design. Moreover, addressing only the memory bottlenecks is not sufficient to achieve high resource utilization in systolic array accelerators, which requires a more complex modelling of the computational characteristics of these accelerators.

To that end, researchers have focused on neural architecture search, which aims to fully automate the design process of DNNs without any need for human supervision. There are three components of neural architecture search frameworks that are critical for their effectiveness and efficiency: optimization strategy, architecture evaluation, and search space. In this section, we discuss the prior work on these three critical components of neural architecture search.

2.2.2 Optimization strategy

Early work on neural architecture search adopted reinforcement learning [Pham et al., 2018; Tan et al., 2019; Zoph & Le, 2017; Zoph et al., 2018], evolutionary algorithms [Marchisio et al., 2020; Real et al., 2019], and Bayesian optimization [Bergstra et al., 2011]. Because these methods operate on a discrete search space and need to perform many trials while searching for an optimal architecture in an exponentially-increasing hyperparameter space, they require thousands of GPU-hours to find optimal DNN architectures, which greatly limits their applicability and even raises concerns about their impact on environment [Patterson et al., 2021]. To mitigate the prohibitive cost of architecture search, techniques such as weight-sharing [Pham et al., 2018] and one-shot search [Bender et al., 2018] have been proposed. Likewise, prior work has also used sparse optimization techniques such as compressed sensing to efficiently search discrete search spaces [Hazan et al., 2018; Cho et al., 2019]. While these techniques reduce the cost of each trial by allowing to reuse trained parameters, they still require many trials to find the optimal DNN architectures.

To further reduce the computational cost, recent work has proposed differentiable neural architecture search [Cai et al., 2019; Chang et al., 2019; Liu et al., 2018; Nayman et al., 2019; Xie et al., 2019; Xu et al., 2020], which is a weight-sharing method with a gradient-descent optimizer. In these methods, a continuous relaxation is applied to the categorical decisions using a set of trainable weights (i.e., architectural parameters). Because differentiable NAS methods use the information from gradients with respect to the architectural parameters

during training, they achieve faster convergence than their non-differentiable counterparts. Thanks to reusing trained parameters and faster convergence, differentiable methods achieve a reduction of 2-3 orders of magnitude in the computational cost of neural architecture search [Dong & Yang, 2019a; Stamoulis et al., 2019].

This remarkable reduction in computational cost has encouraged researchers to use differentiable neural architecture search in various ways: Liu et al. proposed to use this technique to search microarchitecture cells (i.e., basic building blocks) [Liu et al., 2019]. Wu et al. proposed a differentiable hardware-aware neural architecture search [Wu et al., 2019], and Wan et al. used a differentiable neural architecture search to find optimal spatial and channel dimensions for DNNs [Wan et al., 2020]. Because of their superior efficiency and widespread adoption in various application domains, in this thesis, we also focus on differentiable neural architecture search.

2.2.3 Architecture evaluation

The objective functions of hardware-aware neural architecture search frameworks typically contain multiple terms such as accuracy on a given task, inference latency [Wu et al., 2019] and energy consumption [Dai et al., 2019]. While the accuracy metrics can be straightforwardly obtained by calculating the cross-entropy loss on a training dataset, the existing hardware-aware neural architecture search frameworks differ in the way that they estimate the hardware-related metrics. Some of the prior work evaluates the DNN directly on physical devices during the search and use real-time measurements on the loss function [Tan et al., 2019; Yang et al., 2018]. While this approach may allow obtaining hardware metrics accurately, it requires access to physical devices during the search, which hinders its practicality and reproducability.

To eliminate the need for accessing physical devices, more recent work proposed to store the measurements from physical devices on lookup tables and read these values during the search to calculate the loss function [Dai et al., 2019; Stamoulis et al., 2019; Wan et al., 2020; Wu et al., 2019]. However, this approach is also not practical due to two reasons. First, the number of required measurements to represent a search space grows combinatorially with the number of architectural parameters and hardware configurations. Second, these lookup tables correspond to *black-box* models, which are non-differentiable and therefore do not allow gradient calculations. As a result, these models can not effectively be used in gradient descent optimizers.

To obtain a differentiable latency model, some of prior work exploits the fact that a DNN's total latency is equal to the sum of individual layers' latency [Wu et al., 2019]. While this approach allows using gradient descent optimizers to make layer-wise decisions such as which layers to keep or discard, it still assumes black-box models to characterize the latency of each layer, prohibiting the usage of gradient descent to optimize layer parameters such as operator type and channel dimensions. Therefore, the proposed differentiable models are effective to optimize only certain aspects of DNNs and do not offer a complete solution.

To be fully compatible with differentiable frameworks, prior work proposed to estimate the hardware-related metrics using surrogate models such as linear regression [Xiong et al., 2021] or neural networks [Choi et al., 2021]. Unfortunately, these models require large numbers of samples for training and do not generalize well to the out-of-distribution samples; therefore, this approach does not offer an effective solution. Other prior works proposed to use analytical hardware models, which estimates the hardware performance metrics using a cycle-accurate model [Marchisio et al., 2020] or a roofline model [Gupta & Akin, 2020; Li et al., 2021a]. Unfortunately, these models do not represent the runtime characteristics of systolic architectures accurately, leading to significant discrepancies between the estimated and actual values of runtime measurements.

In short, the prior approaches to estimate hardware-related metrics in neural architecture search are either not practical, differentiable, or accurate for systolic architectures. Therefore, this thesis also focuses on developing efficient and effective hardware models for systolic architectures that are compatible with differentiable neural architecture search frameworks.

2.2.4 Search space

The design of the search space plays a critical role in the outcome of the neural architecture search; thus, researchers have put considerable effort into developing strategies to design more efficient and extensive search spaces. To simplify the search complexity, Zoph et al. [Zoph et al., 2018] proposed to search for basic building blocks (*cell*), which can be stacked up to form a DNN architecture. Although this approach can construct intricate cell structures with relatively low search space complexity and is widely adopted by others [Liu et al., 2018; Pham et al., 2018; Liu et al., 2019; Real et al., 2019], using identical cells in all stages of DNN architectures has a negative impact on its accuracy and efficiency [Wu et al., 2019]. Therefore, Tan et al. proposed the hierarchical search space, which allows searching unique cells on different stages of a DNN architecture with varying macro-architectural properties such as the number of layers, spatial and channel dimensions.

Among the macro-architectural properties, channel dimensions are especially of interest due to their significant impact on the computational characteristics of DNN architectures. To that end, prior work has proposed numerous methods to search for optimal channel dimensions [He et al., 2018; Ashok et al., 2018; Cai et al., 2018; Tan et al., 2019]. Among these methods, DMaskingNAS [Wan et al., 2020] have become widely popular thanks to its efficiency. However, this method relies on a fixed search space for the channel dimensions that must be carefully tuned for target resource budget, which undermines its practicality.

A few prior work have addressed the issues inherent to such fixed search spaces. Liu et al. [Liu et al., 2018] proposed the *progressive* neural architecture search, which gradually increases the complexity of the search space during search. Similarly, Ci et al. [Ci et al., 2021] proposed the *neural search space evolution* technique, which enables adding new operations to the search space as the architecture search progresses. However, both of these techniques address

only the search space for cell structures, ignoring other critical properties of DNNs such as channel dimensions. As a result, channel dimension search without the restrictions of a fixed search space remains as an open research question. Therefore, in this thesis, we also focus on developing an efficient neural architecture search for channel dimensions with a flexible search space.

3 Scale-out Systolic Arrays

Systolic arrays have become the architecture of choice for DNN accelerators as they offer superior power efficiency, high arithmeric density, and scalable design. While many variants of systolic arrays have been used in various data processing applications [Kung, 1982], two-dimensional systolic arrays are adopted for DNN accelerators because of their efficiency in performing matrix multiplication (GEMM) operations, which constitute the backbone of DNN workloads. The GEMM operations compiled from DNN layers are typically in the form of $XW + P_{in} = P_{out}$, where X is the input of the layer, W is the trainable parameters of the layer, P_{out} is the output of the layer, and P_{in} is the inital value of P_{out}. Throughout this thesis, we refer to X, W, P_{in}, and P_{out} as the *activations*, *weights*, *input sums*, and *output sums*, respectively.

Figure 3.1 depicts a two-dimensional systolic array, in which the processing elements are placed in a grid of r rows and c columns. Each processing element includes a multiply-and-accumulate (MAC) arithmetic logic unit and a small number of registers to temporarily store the input and output data. The processing elements are connected to their neighbors along rows and columns through uni-directional point-to-point links. While multiple equivalent dataflows exist for two-dimensional systolic arrays, in this thesis, we focus on a weight stationary dataflow due to its widespread adoption in the literature and commerically available accelerators [Jouppi et al., 2017; Kung et al., 2019].

To perform a GEMM operation, a systolic array first fetches the weight matrix row by row and stores them in processing elements. Then, in every cycle, the processing elements in the left-most column fetch activations from the memory banks. Likewise, the processing elements at the top row fetch the input partial sums from the memory banks, perform a multiply-and-accumulate operation, and pass the resulting partial sum to the row below. The operation continues with activations flowing from left to right, partial sums flowing from

Part of this chapter has been accepted to be published in
"Scale-out Systolic Arrays", ACM Transactions on Architecture and Code Optimization, 2022.

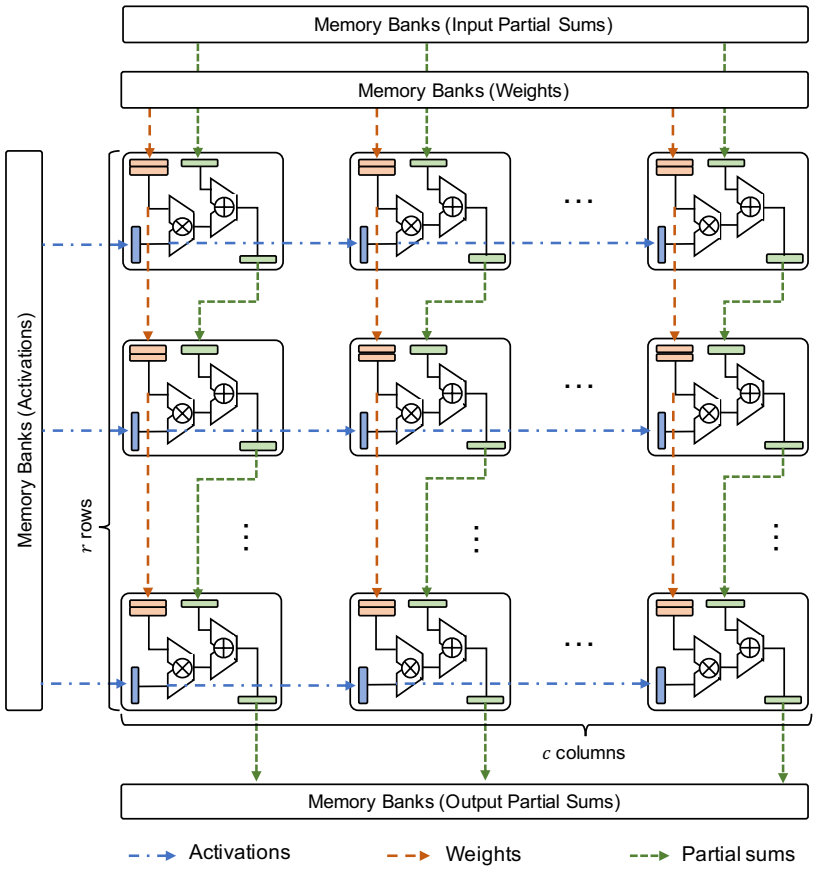


Figure 3.1: A weight-stationary systolic array with r rows and c columns. Activations are assumed to traverse along the rows from left to right, weights and partial sums traverse along the columns from top to bottom.

top to bottom, and weights staying stationary. The processing elements at the bottom row produce the final results and write them back to the memory banks.

While a single unit of systolic array can effectively perform a GEMM operation, modern accelerators often deploy multiple of them in a single die (i.e., *multi-pod* designs) to allow data and task-level parallelism [Kung et al., 2019; Baek et al., 2020; Jouppi et al., 2021]. A multi-pod accelerator's effective throughput is a function of the overall utilization of processing elements both within and across pods. Therefore, maintaining a high utilization not only maximizes the gain from provisioned silicon resources but also improves the overall performance of an accelerator. Figure 3.2 illustrates three main causes of the underutilization in a multi-pod accelerator: (1) dimension mismatch [Samajdar et al., 2020] between array and workload resulting in underutilization within a pod, (2) poor connectivity resulting in underutilization across pods, and (3) sub-optimal tiling resulting in underutilization both within and across pods.

Utilization within a pod highly depends on the pod's systolic array granularity and the layer

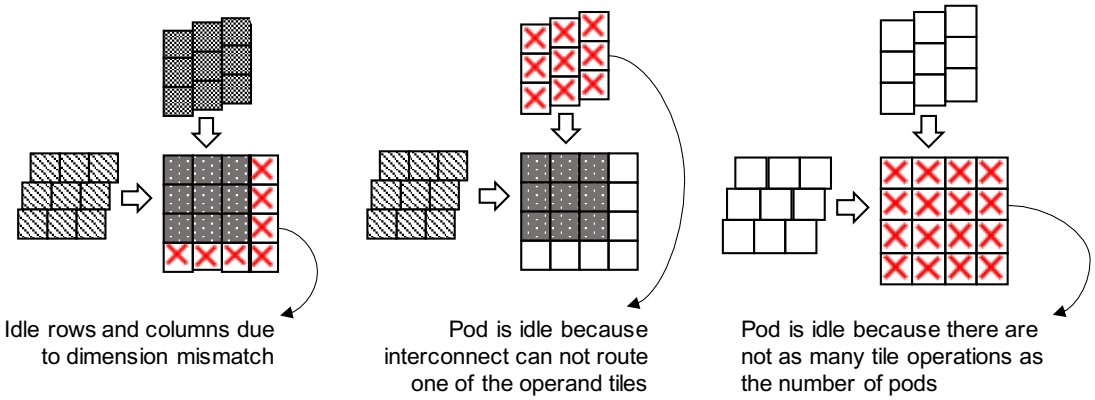


Figure 3.2: Three main factors of underutilization in systolic arrays.

dimensions of a DNN workload. Prior multi-pod accelerators [Jouppi et al., 2017; Google, 2017; Baek et al., 2020] opt for larger array dimensions to reduce access to on-chip memory, provisioning higher power for processing elements. Unfortunately, larger arrays also increase the likelihood that the workload's layer dimensions are smaller than the number of array's rows or columns, resulting in idle processing elements and wasted throughput/Watt. In contrast, minimizing the array dimensions per pod reduces the mismatch between the workloads and the array, resulting in improved utilization. Smaller systolic arrays, however, increase the power required for on-chip memory access, undermining the overall throughput/Watt. In this chapter, we show that the optimal array size for DNN workloads is an order of magnitude smaller than that widely adopted by academia and industry and identify the optimal array size for widely used DNN workloads for computer vision and natural language processing applications.

The interconnect also plays a key role in utilization and effective throughput in multi-pod accelerators [Kung et al., 2019]. To achieve a scalable multi-pod accelerator architecture with high utilization, the interconnect should allow transferring data tiles between systolic pods and memory banks with minimal contention at high efficiency. Therefore, the selected network topology should satisfy several design requirements such as high bisection bandwidth, high combinatorial power, low latency, and good scalability. As these requirements are often contradictory to each other, the network topologies should be carefully evaluated from the perspective of scale-out architectures to ensure high performance and scalability of the accelerators. In this chapter, we perform a quantitative analysis of network topologies in the context of multi-pod accelerators and identify the optimal network topology to connect large numbers of pods efficiently.

Finally, the tile size for each pod fundamentally impacts utilization in multi-pod accelerators and should be tuned with care. We observe that conventional approaches to tiling for systolic arrays [Baek et al., 2020; Choi & Rhu, 2020] fall well short of generating a sufficient number of tile operations to populate a large number of pods, resulting in idle arrays during execution.

On the one hand, choosing large tiles limits the overall number of tile operations and results in idle pods. On the other hand, choosing a small tiling size may introduce underutilization within pods due to internal buffering times. Therefore, we study the impact of tiling size on the utilization in multi-pod accelerators and propose a new tiling strategy that improves the utilization in accelerators with large numbers of pods.

The rest of this chapter is organized as follows: We first investigate the design considerations for the multi-pod accelerators and identify the optimal design choices for target DNN workloads. Then, we introduce the Scale-out Systolic Arrays architecture and elaborate on its implementation details. Finally, we give the details of our experiments and discuss its results.

3.1 Key Pillars of Multi-pod Accelerator Design

Designing a multi-pod accelerator that can efficiently scale to hundreds of pods is a challenging task. To address the challenges in multi-pod accelerator design and develop an effective scale-out architecture, we identify the key design aspects and study them in this section. First, we show that the optimal systolic array size is a function of data dimensions in target DNN workloads and perform a design space exploration of the optimal array sizes for various scenarios. Second, we analyze the charachteristics of the possible interconnection topologies between the pods and memory banks and discuss the advantages of the expanded Butterfly network for the multi-pod DNN accelerators. Finally, we elaborate on tiling and scheduling of DNN workloads on multi-pod architectures and discuss their impact on utilization in DNN accelerators.

3.1.1 Optimal Systolic Array Size

The dimensions of a systolic array have a significant impact on its power efficiency and resource utilization. When the systolic array is fully utilized, all processing elements in a systolic array perform a MAC operation while only those at the edges perform memory operations. More specifically, in each cycle, a systolic array with dimensions of $r \times c$ reads r activation, c weights, c input partial sums from, and writes c to the on-chip memory, adding up $r + 3c$ memory operations per cycle, while it performs $r \times c$ MAC operations. Therefore, the number of memory accesses increases linearly with the array dimensions, while the number of MAC operations grows quadratically. As such, with the increasing array dimensions, a systolic array performs more MAC operations for every byte of data fetched from the memory.

The ability to perform more MAC operations per data, or simply higher data reuse, has significant implications on the power efficiency of systolic arrays. The systolic arrays in inference accelerators typically have low-complexity, integer-point arithmetic logic units in their processing elements due to the reduced precision requirements of DNN inference, which consumes significantly less power (about an order of magnitude [Chen et al., 2016]) than fetching data from on- or off-chip memory. As a result, the power consumption of DNN accelerators is often

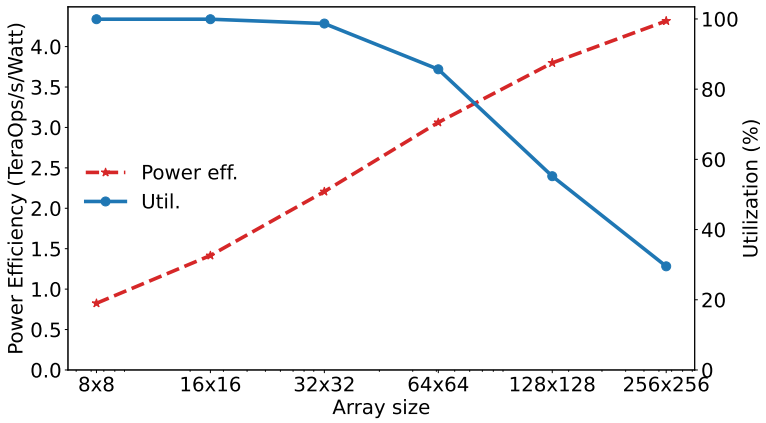


Figure 3.3: Trade-off between power efficiency and utilization in systolic arrays with respect to the array dimensions.

dominated by the memory operations. Because memory operations constitute the majority of the power consumption, higher data reuse thanks to the larger array dimensions directly translates into higher power efficiency, encouraging monolithic designs with larger systolic arrays in DNN accelerators such as Google's TPU [Jouppi et al., 2017].

Despite their high power efficiency, systolic arrays with large array dimensions suffer from underutilization. As we increase the array dimensions, the number of rows and columns in a systolic array starts to exceed matrix dimensions of DNN layers. When the weight matrix dimensions are smaller than the array dimensions, the excess rows and columns become idle, resulting in underutilization. Moreover, in systolic array designs with double buffering [Ross, 2017], the entire array stalls between operations if the matrix multiplication takes fewer cycles than the weight buffering time. Because the execution time of a GEMM operation is approximately equal to the first dimension of the activation matrix and the weight buffering time is proportional to the number of rows in the array, choosing a number of rows that is larger than the dimensions of the activation matrix also results in underutilization. In short, systolic arrays with large numbers of rows and columns are suffer from underutilization due to dimension mismatches and weight buffering times.

To study the trade-off between power efficiency and utilization in systolic arrays, we developed a computational model for systolic arrays. The proposed computational model estimates the power efficiency and utilization of a systolic array by calculating the required amount of on-chip memory access and MAC operations and simulating the execution of a DNN model on systolic arrays with various array dimensions. We validated the correctness of our power and utilization estimations against the post-synthesis simulations of our RTL design. Figure 3.3 shows the results of our power and utilization estimates for InceptionNet [Szegedy et al., 2016] with an image size of 299×299 and a batch size of 1. We observe that, for small array dimensions, systolic arrays exhibit low power efficiency due to low data reuse whereas they

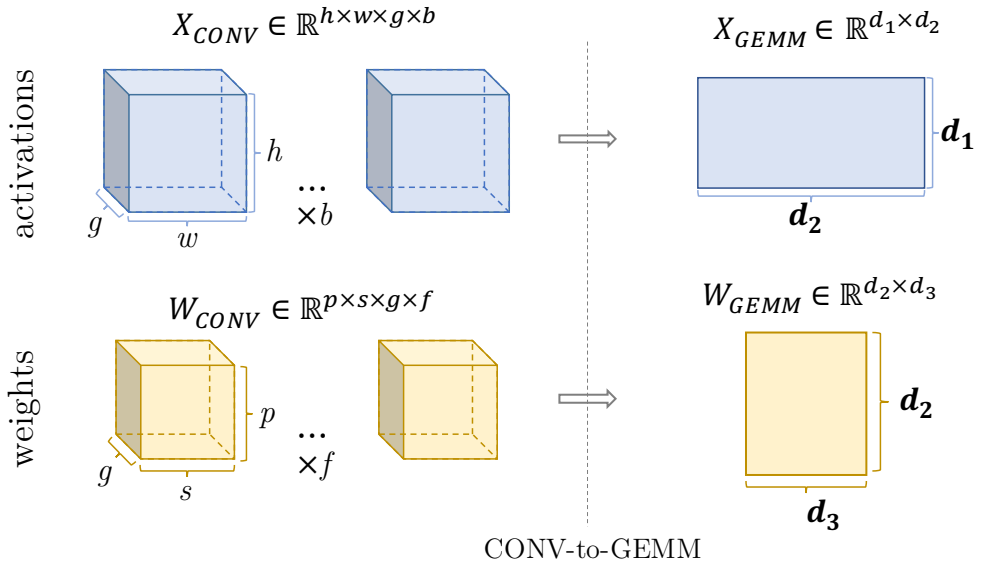


Figure 3.4: Illustration of CONV-to-GEMM conversion. Four-dimensional activation and weight tensors of a convolution operation are reshaped into two-dimensional matrices.

run the model's inference with high utilization as they are less likely to suffer from dimension mismatches. As we increase the array dimensions, we see that the power efficiency increases thanks to higher data reuse. However, we observe that the utilization starts sharply falling for the array dimensions larger than 32×32 due to higher mismatch between the model's layers and array dimensions. Therefore, these two metrics that are critical for the performance of the systolic arrays, namely power efficiency and utilization, are inversely proportional to each other in terms of array dimensions and finding an optimal design point requires an in-depth study of the workload characteristics and a detailed design space exploration.

To gain insights into the distinguishing characteristics of various DNN workloads, we analyze the operand dimensions of the GEMM operations in popular DNN workloads. To do so, we must first understand how such workloads are compiled into GEMM operations.

Convolutional layers correspond to four-dimensional tensor operations; thus, they are first transformed into two-dimensional matrix multiplications with a process called CONV-to-GEMM conversion [Jordà et al., 2019]. Figure 3.4 illustrates the CONV-to-GEMM conversion process, where the four-dimensional activation (X_{CONV}) and weight (W_{CONV}) tensors are rearranged to obtain two-dimensional X_{GEMM} and W_{GEMM} matrices. In this figure, $h \times w$ and $p \times s$ denote the window size of activations and weights and g, f, and b denote the number of input channels, filters, and batch size, respectively. As a result of this CONV-to-GEMM transformation, the dimensions of the X_{GEMM} and W_{GEMM} matrices are also transformed and become equal to: $d_1 = hwb$, $d_2 = psg$, and $d_3 = f$. In contrast to the convolutional layers, the fully-connected and self-attention layers are linear transformations, thus they can be directly represented as GEMM operations. Therefore, in fully-connected and self-attention layers,

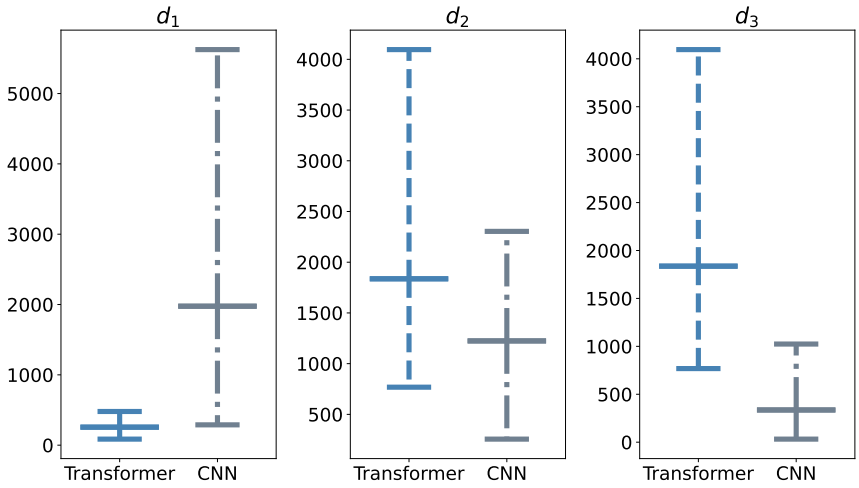


Figure 3.5: Range of the matrix dimensions for the GEMM operations in BERT and CNN models. d_1 denotes the first dimension of activation matrix (X_{GEMM}) and d_2 and d_3 denote the first and second dimension of weight matrix (W_{GEMM}). Horizontal lines show 10th percentile, average, and 90th percentiles.

d_1, d_2, and d_3 are simply equal to the batch size, number of features, and number of filters, respectively.

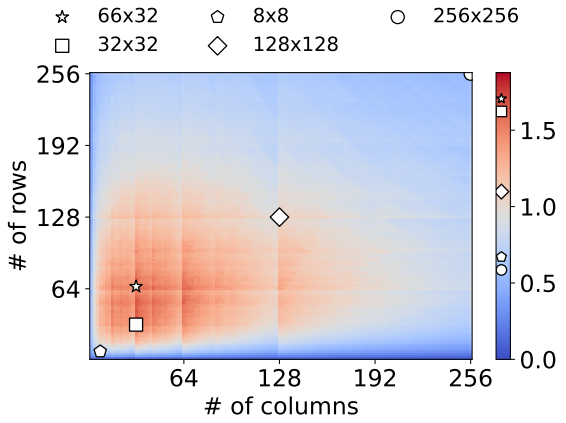
Figure 3.5 shows the range of the values for d_1, d_2, and d_3 in CNN models (i.e., Resnet, InceptionNet, and Densenet with an image size of 299) and Transformer models (i.e., BERT-medium, BERT-base, and BERT-large with a sentence length of 128) models. We calculated the values of d_1, d_2, and d_3 as a result of the CONV-to-GEMM conversion process explained above. Although the dimensions of the DNN layers vary both across and within models, we make the following observations. First, GEMM operations in CNN models have significantly larger values for d_1 (15× on average) than Transformer models. This large difference is mainly due to the convolutional reuse, i.e., filters in convolutional layers stride across input images. As a result, CNN models run at higher utilization on systolic arrays with a larger number of rows than columns. Our second observation is that GEMM operations in Transformer models have significantly larger values for d_3 (about 6×) than convolutional layers, which indicates that Transformer models are better suited than CNNs for systolic arrays with more columns than rows. These observations indicate that the computational requirements of these two popular DNN workloads are contradictory to each other, complicating the design of a generic DNN accelerator that performs well on both type of workloads.

To find the optimal array sizes for these two types of workloads, we perform a design space exploration. For this purpose, we use an optimization metric called *effective throughput per Watt*, which is a function of both utilization and power efficiency. Due to the utilization component of the effective throughput/Watt metric, the optimal array size is sensitive to the selection of target workloads. Therefore, we conduct a design space exploration for three

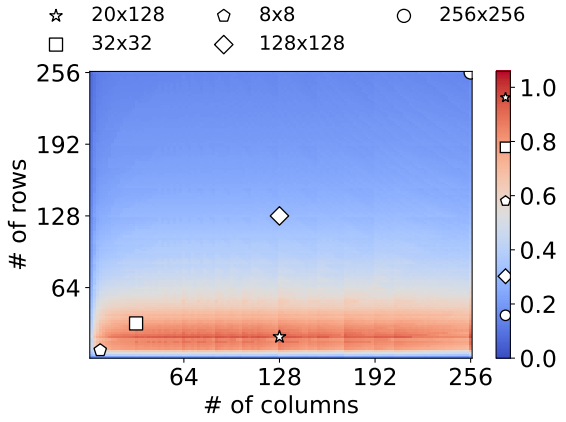
cases: we first study the CNN and Transformer models separately, and then a mixture of both types. To compare our proposed design choice, we pick four baseline array sizes that represent the majority of designs in industry and academia: 512×512, which represents the monolithic systolic array; 256×256, which represents Google's TPU v1 [Jouppi et al., 2017], 128×128, which represents TPU v4 [Jouppi et al., 2021] and AI-MT [Baek et al., 2020], and 8×8, which represents the Maestro [Kung et al., 2019]. For this design space exploration, we use our computational model for systolic arrays to obtain the power efficiency and utilization values, and then calculate the effective throughput as peak throughput multiplied by utilization. Our design space exploration is isopower because DNN inference accelerators are typically bound by power consumption because of their dense arithmetic formats [Sze et al., 2017].

Figure 3.6a shows the design space exploration for CNN models, namely Inception-v3, ResNet, DenseNet with input image sizes of 224×224, 256×256, and 299×299. Because the number of filters in CNN models is typically limited compared to the number of filter reuse and the number of features as shown in Figure 3.5, we observe that optimal design points have a large number of rows and a small number of columns. In fact, the design point with the highest effective throughput for CNN models is 66×32, which is about $1.3\times$ better than any other baselines. In contrast, Figure 3.6b shows the design space for Transformer models with sequence lengths of 10, 20, 40, 60, 80, 100, 200, 300, 400, 500. Because the number of filter reuse in Transformer models is typically limited compared to the number of filters and number of features as shown in Figure 3.5, we observe that the optimal design points have a large number of columns and a small number of rows. More specifically, the design point with the highest effective throughput for Transformer models is 20×128, which is also about $1.7\times$ better than any other baselines.

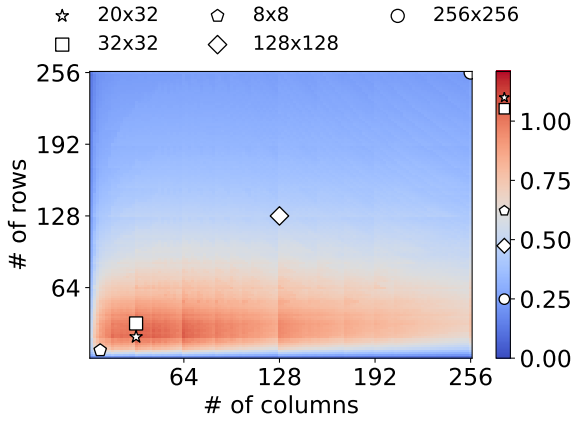
Our design space exploration for CNN and Transformer models show that accelerators that are specialized for either CNNs or Transformers have contrasting optimal array dimensions. An accelerator that is specialized for one type of model would perform poorly for the other type. Therefore, we perform a design space exploration that takes both types of models into account. Figure 3.6c shows the design space for a mixture of CNN and Transformer models with equal weights. In this case, data dimensions of both model types have an impact on the shape of the design space: the areas with large array dimensions exhibit low effective throughput/Watt due to underutilization. Likewise, areas with very small dimensions also have low effective throughput/Watt due to poor power efficiency. We identify that only a small part of the design exhibits high effective throughput/Watt, where the number of rows and columns are in the order of ten to hundreds. In fact, the highest effective throughput/Watt is obtained with array dimensions of 20×32, which has an effective throughput/Watt of 1.34, 0.93, and 1.1 TeraOps/s/Watt for CNNs, Transformers, and their mixture, respectively. To facilitate implementation and connectivity to memory (e.g., alignment to cache block size), we round up the number of rows from 20 to 32 and choose 32×32 as the array size in our design.



(a) Effective throughput (TeraOps/s) per Watt for CNN models.



(b) Effective throughput (TeraOps/s) per Watt for Transformer models.



(c) Effective throughput (TeraOps/s) per Watt for the mixture of CNN and Transformer models.

Figure 3.6: Design space exploration for systolic arrays. Colormap represents the effective throughput (TeraOps/s) per Watt.

3.1.2 Interconnection Network

For the interconnect between systolic arrays and memory banks, we prioritize four design requirements. First, the interconnect should provide sufficient bisection bandwidth to allow continuous data read and write for all systolic pods simultaneously. Second, the interconnect should facilitate high combinatorial power (i.e., the ratio of realizable input-output permutations [Beneš, 1964]) to connect large numbers of pods with memory banks freely. Third, the interconnect should have a latency shorter than the execution of tile operations so that the interconnect latency can be hidden by computation. Finally, the interconnect should scale well up to hundreds of systolic pods in terms of power consumption and silicon area.

2D mesh [Chen et al., 2017, 2014b; Gao et al., 2019; Shao et al., 2019] and H-trees [Kung et al., 2019; Stevens et al., 2019] are popularly used in many DNN accelerators thanks to their relatively low hardware cost. However, neither of them can provide enough bisection bandwidth for large numbers of systolic pods. To improve their bisection bandwidth, one can replicate an H-tree interconnect N-times (scaled-up H-tree [Kung et al., 2019]), which results in an unfeasible hardware cost with a complexity of N^2. Although Crossbar interconnect [Zhu et al., 2020] offers high bisection bandwidth, it also has a quadratically increasing hardware cost with respect to the number of pods. As such, we conclude that H-tree and Crossbar are not suitable for multi-pod DNN accelerators due to their excessive hardware cost.

Multistage interconnect networks emerge as a promising solution for the energy-efficient multi-pod DNN accelerators as they exhibit sufficient bisection bandwidth, relatively low hardware cost with a complexity of $N \log N$ and low latency. Prior DNN accelerators[Qin et al., 2020] have proposed to use Benes network[Beneš, 1964], which is a non-blocking multistage interconnection that consists of $(2 \log N - 1)$ stages. Compared to the H-tree and Crossbar, the Benes network has a feasible hardware cost of $N \log N$ while providing a sufficient bisection bandwidth of N. However, the Benes network suffers from a considerable latency that is proportional to its large number of stages, $2 \log N - 1$. While the Benes network can route all possible input-output permutations without contention, it offers only a limited multi-casting capability. Due to this limitation, we consider the augmented version of the Benes network with a copy network [Liew & Lee, 2010] instead of its standard implementation, which enables the full multi-casting capability at the expense of longer latency. Consequently, none of these interconnects mentioned above satisfy the design requirements of an accelerator with a large number of pods.

The Butterfly network, which is also a multistage interconnect, offers high bisection bandwidth with low latency and scalable hardware cost. A standard Butterfly network provides only limited multicasting and combinatorial power; however, this limitation can be alleviated by employing multiple of them in parallel [Liew & Lee, 2010], where the number of parallel butterflys is referred to as *expansion factor*. Figure 3.7 depicts an example Butterfly network with eight source and destination ports, and with an expansion factor of two. Thanks to the redundant switches and links between source and destinations facilitated by the expansion,

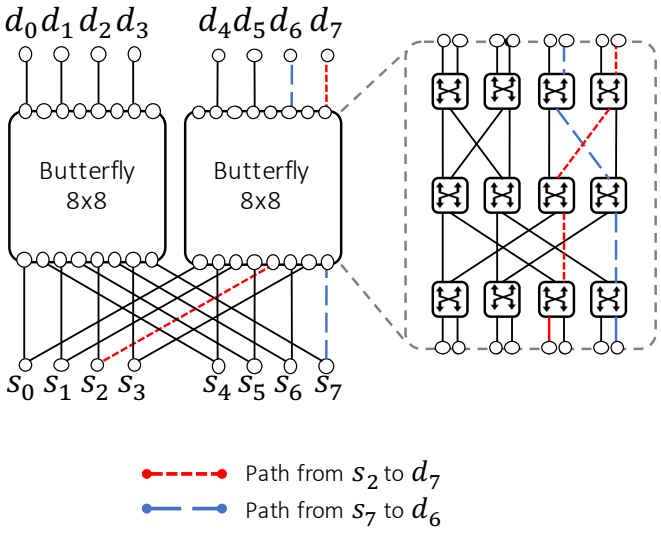


Figure 3.7: An 8 × 8 Butterfly network with an expansion factor of 2. Routings from s_2 to d_7 and from s_7 to d_6 are shown with blue and red lines, respectively.

we achieve higher combinatorial power than the standard Butterfly network; i.e., more permutations are feasible without network contention. For instance, the paths from s_3 to d_2 and s_6 to d_3 can be routed simultaneously as shown in Figure 3.7, whereas this permutation would not be possible in a standard Butterfly implementation. Moreover, because the network is expanded vertically rather than horizontally, its latency remains low, allowing to overlap data movement with computation for a larger number of pods. In the rest of this thesis, we will refer to the expanded Butterfly network as Butterfly-k, where k is the expansion factor.

Table 3.1: Interconnect performance metrics generated by our cycle-accurate simulator averaged across all the workloads.

Type	Busy Pods [%]	Cycles per Tile Op.	mW/byte
Butterfly-1	66.81	19.72	0.23
Butterfly-2	72.41	20.17	0.52
Butterfly-4	72.26	20.27	1.15
Butterfly-8	72.43	20.48	2.53
Crossbar	72.38	19.73	7.36
Benes	72.38	30.00	0.92

To evaluate various interconnect types from the perspective of the design requirements, we identified three performance metrics, which are listed in Table 3.1. First, the percentage of busy pods is defined as the average ratio of busy pods to the total number of pods. We expect that the interconnects with higher combinatorial power achieve a higher percentage of busy pods due to reduced contention. Second, the number of cycles per tile operation describes how long it takes for a systolic pod to complete a tile operation. As we overlap the

interconnect accesses with the systolic pod computation, the interconnect latency is typically hidden by the execution time of tile operations. However, if the interconnect latency is too long, it may become exposed, thereby increasing the number of cycles per tile operation metric. Finally, Watt/byte of the interconnect characterizes the power consumption. We seek a smaller Watt/byte value for reducing the overall power consumption.

Table 3.1 presents the previously mentioned performance metrics for various types of interconnects. First, we observe that the standard Butterfly network has a reduced percentage of busy pods by 6% due to its insufficient combinatorial power. However, expanding it with a factor of two is sufficient to increase its percentage of busy pods to that of interconnects with full combinatorial power, such as Crossbar and Benes. Second, the interconnect types with low latencies (i.e., Butterfly and Crossbar) results in the minimum number of cycles per tile operation because their latencies are hidden by the computation. However, Benes network, which has a substantially longer latency incurs a significant overhead (around 50% more) in the number of cycles per tile operation. Finally, Crossbar, whose power consumption increases quadratically with the number of arrays, requires 8× and 32× more watts per byte than Benes or standard Butterfly networks for 256 pods. To conclude, the standard Butterfly, Benes, and Crossbar interconnects are not suitable for multi-pod accelerators due to their insufficient combinatorial power, long latency, and high watts per bytes metrics, whereas Butterfly network with an expansion of two is the optimal design choice thanks to its sufficiently high combinatorial power, relatively short latency, and low watts per bytes.

3.1.3 Tiling & Scheduling

Tiling and scheduling play an essential role in maintaining high utilization across large numbers of pods. In this subsection, we first propose a fixed-length tiling strategy with an emphasis on optimal tiling dimensions that maximize utilization across a large number of pods. Then, we explain our scheduler implementation, which maps tile operations onto systolic pods while handling the interconnect constraints, tile dependencies, and bank conflicts.

Performing a GEMM operation on systolic arrays often requires partitioning data into tiles due to dimension mismatches. The resulting tile operations can be performed on a single array sequentially, or they can be distributed among multiple arrays and performed in parallel. In weight stationary systolic arrays, the weight matrix W is spatially laid out onto the systolic arrays; thus, W must be partitioned into tiles of $r \times c$ to match the array dimensions, where r and c denote the number of array rows and columns, respectively. Because the first dimension of W must match the second dimension of the activation matrix X in a matrix multiplication, X's second dimension is also required to be partitioned with a size of r. Moreover, we can further partition X's first dimension to obtain more tile operations. Because the execution time of a matrix multiplication is approximately equal to X's first dimension, the resulting tile operations have shorter execution times.

We observe that the data tiling strategies proposed by prior work either do not partition

X's first dimension [Baek et al., 2020], or choose a partition size that is much larger than array dimensions [Choi & Rhu, 2020]. We argue that not partitioning X's first dimension or choosing a large partition size does not exploit the available data-level parallelism in matrix multiplication operations to the fullest extent, limiting the number of pods that can run in parallel. Partitioning the X matrix into smaller tiles produces more tile operations that can run in parallel. However, decreasing the partition size below a certain threshold value results in underutilization within pods. This threshold is the number of rows in an array (r) because the execution time for tile operations becomes shorter than r cycles, which exposes the weight buffering time. Therefore, we propose to partition the activation matrix X into tiles of $r \times r$, which produces as many parallel tile operations as possible without undermining utilization within pods.

Our compiler first performs a tiling stage during the compile time, which determines the start and end of the tiles addresses from activation and weight matrices. During the runtime, the proposed architecture loads the tiles onto on-chip memory banks in the proposed tiling format and performs GEMM operations using the loaded tiles. The resulting partial sum tiles are either consumed directly from on-chip memory by the subsequent GEMM operations, or they are written back to the off-chip memory.

3.2 Scale-out Systolic Arrays

In the previous section, we found that the optimal array size to maximize effective throughput/Watt is 32×32, which requires a thermal design power (TDP) of about one Watt. Today's server form factors allow TDPs of up to several hundreds of Watts, which implies that a single accelerator can contain hundreds of optimally sized systolic arrays. Therefore, in this section, we propose the scale-out systolic array (SOSA) architecture, which efficiently employs multiple systolic arrays in parallel to accelerate DNN workloads.

Figure 3.8 shows the overall diagram of the SOSA accelerator. Each systolic pod encapsulates a systolic array with the peripherals required to perform GEMM operations. The main controller fetches instructions from a dedicated cache, issues them to the corresponding pods, and synchronizes the pods to perform their operations in lockstep. Activation, weight, and partial sum tiles are stored in dedicated on-chip memory banks to reduce the interconnect width between memory banks and systolic pods. A SIMD post-processor performs element-wise operations on the partial sums and writes its results back to the activation or partial sum banks.

3.2.1 Systolic Pod Microarchitecture

Our systolic pod design brings CONV-to-GEMM converter and skew/deskew buffers near systolic arrays to minimize interconnect traffic. As shown on the right-hand side of Figure 3.8, a systolic pod consists of a systolic array, a CONV-to-GEMM converter, skew/deskew buffers,

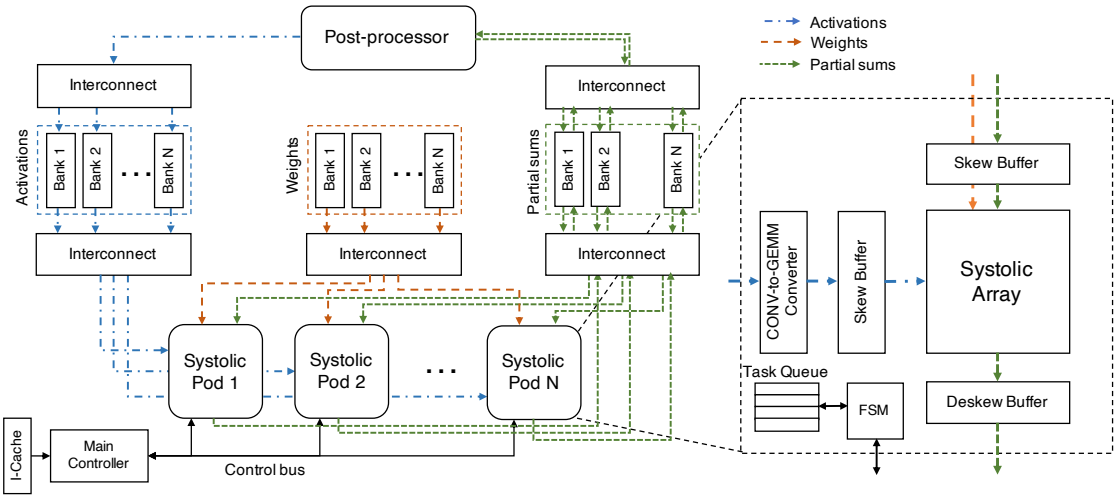


Figure 3.8: Overview of the proposed architecture, with the internals of the Systolic Pod shown on the right-hand side.

and a local controller (FSM) with a task queue that stores instructions. The CONV-to-GEMM converter [Liu et al., 2020b] is a hardware block that converts activation data from a four-dimensional convolutional to two-dimensional matrix format to prevent redundant on-chip memory accesses. Likewise, skew and deskew buffers apply a skew to activation and input partial sums, and remove the skew from output partial sums to improve the memory efficiency. In this design, we encapsulate the systolic arrays with the CONV-to-GEMM converters and skew/deskew buffers to improve memory efficiency and reduce network traffic.

In standard systolic array implementations, activations and partial sums propagate at a rate of one column and one row per cycle. These slow propagation rates incur long pipeline latencies and require large skew/deskew buffers. To mitigate these problems, prior work [Kung, 1982; Liu et al., 2020a] proposed to multicast activations across multiple rows at each cycle and use adder trees to accumulate multiple partial sums at each cycle. In our systolic pod design, we also use activation multicasting and partial sum fan-in methods to reduce pipeline bubbles and buffer sizes. In each cycle, we multicast activation values to U consecutive processing elements along the rows. Likewise, we propagate partial sums with an offset of V processing elements along the columns and use adder trees accumulate V partial sums. The selection of the design parameters U and V is critical for the performance of the systolic pods. On the one hand, setting the parameters U and V as one (corresponds to the standard systolic arrays) leads to the best timing thanks to the short paths between registers. However, it incurs large pipeline latencies that result in idle processing elements between tasks. On the other hand, choosing large U and V parameters hinders timing characteristics due to longer paths between registers, but it improves utilization thanks to reduced pipeline latencies. For the optimal array size that we found in our design space exploration (32×32), we choose the parameters U and V as 16 and 16, respectively.

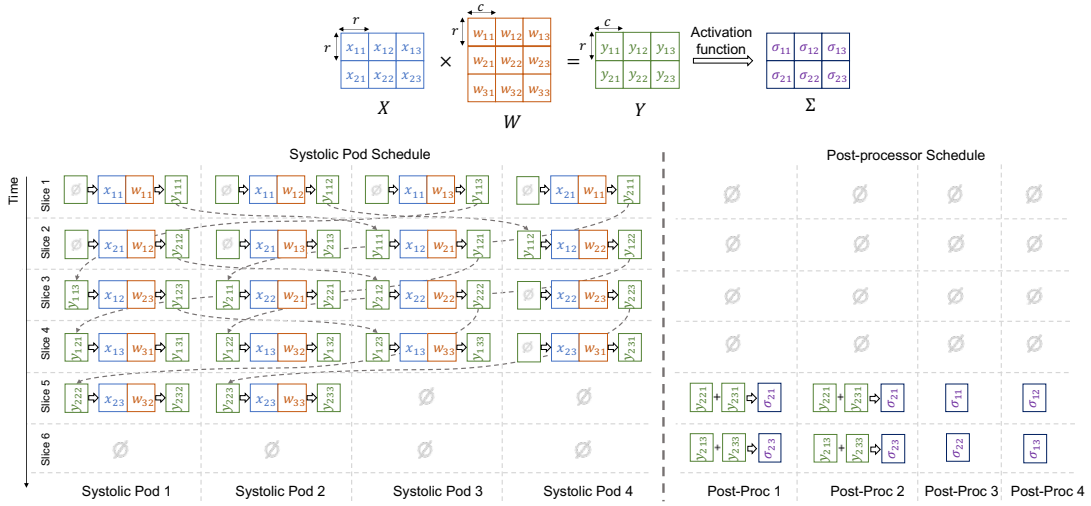


Figure 3.9: Tiling and scheduling example of a matrix multiplication of $X \times W \rightarrow Y$, followed by an activation function $Y \rightarrow \Sigma$. The example shows the scheduling for four systolic pods with array sizes of $r \times c$, and four post-processors.

3.2.2 Offline Scheduling Algorithm

Based on the fixed partition size that we described in Section 3.1.3, we propose an offline scheduling algorithm for multi-pod systems. Compared to online (dynamic) scheduling algorithms, the proposed algorithm enables performing more aggressive code optimizations (e.g., reordering and remapping the operations), leading to higher resource utilization and faster execution. Because our data tiling scheme produces tile operations with identical execution times, we design a scheduler with fixed time slices of r cycles. The scheduler takes a list of tile operations generated by the tiling algorithm and attempts to map and schedule them on the slots of available systolic pods at the earliest available time slice. The scheduler has three constraints while checking a tile operation's availability for a slot. First, there must be no read-after-write data dependence between scheduled tile operations. Second, a memory bank can not be accessed by another pod as we assume single-ported banks. Third, the interconnect must be able to route all pod-bank permutations for a given time slice. Starting from the first tile operation, the scheduler searches for available slots in time slices that satisfy all three conditions.

Algorithm 1 shows the pseudocode of how the scheduler schedules a tile operation (g). The scheduler first finds the earliest possible time slice (l) by checking the dependencies among tile operations. Then, it finds the idle systolic pods and memory banks in the time slice found in the previous step. Next, it exhaustively searches all combinations of available pods and memory banks and checks whether a routing between pods and banks is possible. If it finds a valid routing for all X, W, and P interconnects, it schedules the tile operation in the time slice l. If it fails to find any valid routing after all combinations are exhausted, it repeats the same steps for the next time slice, $l+1$. The scheduler repeats this process until all tile operations

Input: g: Tile operation
$l \leftarrow$ find layer's beginning round
while g *is not scheduled* **do**
    $S_{idle} \leftarrow$ find idle systolic pods in round l ;
    $X_{idle} \leftarrow$ find available X banks in round l ;
    $W_{idle} \leftarrow$ find available W banks in round l ;
    $P_{idle} \leftarrow$ find available P banks in round l ;
    foreach s *in* S_{idle} **do**
        foreach x *in* X_{idle} **do**
            if $x \rightarrow s$ *is* ***not*** *free* **then**
                continue;
            end
            foreach w *in* W_{idle} **do**
                if $w \rightarrow s$ *is* ***not*** *free* **then**
                    continue;
                end
                foreach p *in* P_{idle} **do**
                    if $s \rightarrow p$ *is* ***not*** *free* **then**
                        continue;
                    else
                        assign g to s in round l;
                        assign tiles to bank x, w, and p;
                        return ;
                    end
                end
            end
        end
    end
    $r \leftarrow r + 1$;
end

Algorithm 1: The proposed mapping and scheduling algorithm. $a \rightarrow b$ represents the interconnect path from a to b

are scheduled.

Figure 3.9 shows the result of our tiling and scheduling algorithm on a small-scale example. Each column in the figure represents a systolic pod or post-processor and each row represents a time slice. In each time slice, a systolic pod performs a tile operation $x_{ij} \times w_{jk} + y_{imk} = y_{ijk}$, where x_{ij} and w_{jk} are tiles from X and W, respectively, y_{imk} is an optional input partial sum, and y_{ijk} is the output partial sum. The output partial sums (y_{ijk}) are then aggregated to obtain the final output tiles: $y_{ik} = \sum_j y_{ijk}$. Finally, post-processors applies an activation function on final output tiles (y_{ik}) to obtain output activations, σ_{ik}.

Due to the aggregation operations between partial output tiles, there are data dependencies between the tile operations. There are two ways of performing these tile aggregations. The

output of a tile multiplication can be mapped onto a later multiplication as the input partial sum (shown with dashed lines in Figure 3.9), or the aggregation of two tiles can be performed in the idle slots of post-processors. Post-processors work in pairs to perform tile aggregations to match the throughput of systolic pods. We use an exhaustive search to map aggregation operations on systolic arrays and post processors.

3.3 Experiments

3.3.1 Methodology

We synthesized the proposed systolic pods using the TSMC 28nm process technology and Synopsis Design Compiler, then measured that the energy consumption per MAC operation is 0.4 pJ at a clock frequency of 1GHz. We encoded the weight and activation values as 8-bit and partial sums as 16-bit integers, the same as prior work [Drumond, 2020; Jouppi et al., 2017; Baek et al., 2020]. We modeled the on-chip memory banks using Cacti-P [Li et al., 2011]. As we use an N-to-N interconnect, we employ the same number of SRAM banks as the number of systolic pods. We chose the SRAM bank size as 256 KB, which is the smallest bank size that can store the working set of all of the benchmarks. We calculated that the energy per byte for accessing memory banks is 2.7 pJ/Byte using Cacti-P [Li et al., 2011]. For off-chip memory access, we assume 32GB HBM3 memory with a bandwidth of 1.2 TB/s.

To evaluate the proposed method, we selected a number of benchmarks from the two most widely used application domains of DNNs, namely computer vision and natural language processing. Because the majority of DNN models (all top ten models in the Imagenet competition) for computer vision tasks are convolutional, we choose a number of widely used, state-of-the-art CNN models, namely Inception-v3 [Szegedy et al., 2016], ResNet50, ResNet101, ResNet152 [He et al., 2016], DenseNet121, DenseNet169, and DenseNet201 [Huang et al., 2017]. We use the pre-trained models provided by Keras [Chollet, 2015] for CNNs with an input image size of $299 \times 299 \times 3$. Transformer models are ubiquitous in the NLP domain (eight out of the top ten in the WMT-14 English-German dataset use a form of Transformer models). Therefore, we select three BERT models [Devlin et al., 2019], namely BERT-medium, BERT-base, and BERT-large [Google, 2020]. For BERT models, we select the median value of the sequence lengths from the benchmark [Tencent, 2020], which is equal to 100.

3.3.2 Results

In this section, we first show that the proposed array size (32×32) offers the highest effective throughput among all other design points. Then, we show and discuss the impact of multi-batching and multi-tenancy in the effective throughput and its scalability. After that, we evaluate various interconnect types presented in Section 3.2 and demonstrate that the Butterfly network is the most optimal choice to connect large numbers of systolic pods. Next, we evaluate the proposed tiling scheme and quantitatively show the improvement achieved by

choosing the optimal tiling size. Later, we analyze the SRAM bank size's impact on the off-chip DRAM usage and effective throughput, and identify the optimal SRAM bank size for the proposed design. Finally, we share the details of our RTL implementation and synthesis results to show the validity of our insights and assumptions. For all design points, we consider a TDP of 400 Watts as in [NVIDIA, 2020] and calculate the number of parallel systolic arrays as the largest power-of-two number that results in a peak power consumption smaller than the TDP.

3.3.3 Array Granularity

Systolic array designs in academia and industry covers a wide range of array sizes (e.g., 8×8 in Maestro [Kung et al., 2019], 128×128 in TPUv2, v3, v4 [Google, 2017] and AI-MT [Baek et al., 2020], and 256×256 in TPUv1 [Jouppi et al., 2017]. To cover the entire design space, we run our experiments with array sizes from 16×16 to 512×512 with steps of power-of-two numbers. Moreover, to compare SOSA against the monolithic designs (e.g., TPUv1 [Jouppi et al., 2017]), we have the Monolithic baseline, in which available processing elements are organized in the form of a single systolic array. For the TDP of 400 Watts, the monolithic baseline corresponds to an array size of 512×512, while it varies from 400×400 to 1024×1024 for the experiments shown in Figure 3.11.

Table 3.2 summarizes the performance results of SOSA with varying array granularities. Because large systolic arrays have higher power efficiency than smaller ones, Monolithic baseline achieves the highest theoretical peak throughput (1.85 PetaOps/s) among all other array granularities. However, due to underutilization in large systolic arrays, Monolithic baseline exhibits only 10.3 % of its peak throughput, which corresponds to an effective throughput of 191.3 TeraOps/s. In contrast, array granularity of 16×16 has the lowest peak throughput due to its low power efficiency. As a result, even though it achieves the highest utilization, its effective throughput is only 198.9 TeraOps/s because of its limited peak throughput.

Table 3.2: Performance of SOSA with various array sizes. The effective throughput is the harmonic mean of CNN and Transformer models.

Systolic Array Size	# of Pods	Peak Power [Watts]	Peak Throughput @400W [TeraOps/s]	Util. [%]	Effective Throughput @400W [TeraOps/s]
512×512	1	113.2	1853	10.3	191.3
256×256	8	245.0	1712	14.0	183.0
128×128	32	283.1	1481	13.8	205.0
64×64	128	362.2	1158	17.4	200.9
16×16	512	210.6	498.0	55.5	198.9
20×32	256	211.1	620.8	40.0	344.5
32×32	**256**	**260.2**	**806.0**	**39.4**	**317.4**

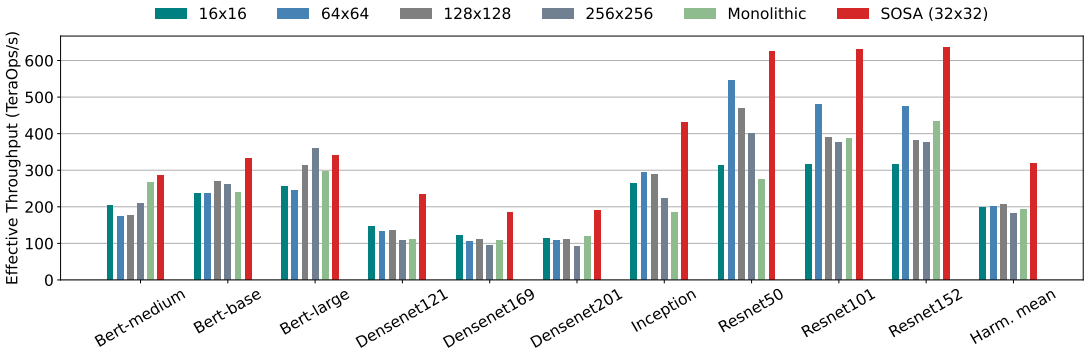


Figure 3.10: Effective throughput of SOSA with various array sizes and Monolithic baseline for various DNN benchmarks. All values are normalized to 400 Watts.

As discussed in the design space exploration of Section 3.1.1, the array size of 20×32 finds a balance between power efficiency and utilization and achieves the maximum effective throughput, which is 344.5 TeraOps/s. However, the number of rows in this specific array size is not a power of two, which introduces the following implementation difficulties and inefficiencies: First, memory bus and SRAM banks typically have a width value that is power-of-two; thus, aligning data size with the bus and cache block width by choosing a number of rows that is a power of two in an array improves memory efficiency. Second, systolic arrays typically utilise adder trees to reduce pipeline latencies: their efficiency is maximised when the array dimensions are powers of two. Thus, we round up the number of rows from 20 to 32 and use the array size of 32×32.

The results given in Table 3.2 also show the effectiveness of the proposed design space exploration method introduced in Section 3.1.1. Although the computational model used in the design space exploration is a first-order approximation and orders of magnitude faster than the cycle-accurate simulations, it can identify the optimal array size for the given DNN workloads. Therefore, we conclude that the proposed design space exploration method can be employed to analyze the performance of multi-pod systolic arrays.

Figure 3.10 shows the breakdown of effective throughputs for individual DNN models. We observe, in nine out of ten benchmarks, SOSA with the array size of 32×32 outperforms other designs by up to a factor of ~ 1.6. The only benchmark that 32×32 does not exhibit the highest throughput is BERT-large, for which the array size of 256×256 outperforms 32×32 by a factor of 1.06. We argue that the reason why 256×256 outperforms other designs for BERT-large is because its array dimensions are well-aligned with the data dimensions in BERT-large. Nevertheless, these results show that SOSA with the array size of 32×32 is the optimal design point for a wide range of state-of-the-art DNN workloads, with an average effective throughput higher than prior designs by a factor of $1.55\times$.

In the previous evaluation, we assumed the number of systolic pods as 256. However, AI accelerators' power and area budgets may vary depending on the system requirements; thus,

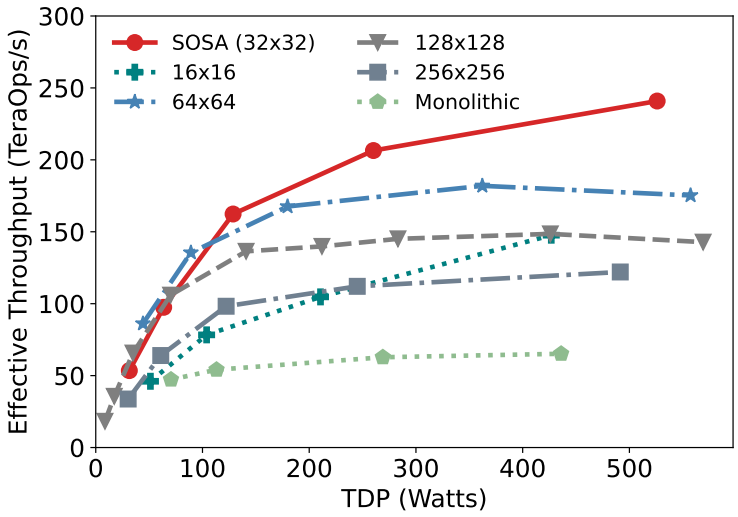


Figure 3.11: Effective throughput of SOSA and Monolithic baseline for various TDP values. For Monolithic baseline, we assume a single systolic array and vary its dimensions between 400×400 and 1024×1024; whereas for SOSA designs, we use keep the array size constant and vary the number of parallel pods.

we evaluate the proposed architecture's sensitivity to the number of pods. Figure 3.11 shows the effective throughput for various numbers of pods. We observe that, for TDP values larger than 90 Watts, SOSA with the array size of 32×32 outperforms all other designs by a factor up to 1.5×, when the number of arrays is scaled up to 512. We also observe that the increase in the effective throughput starts to saturate as we increase the number of arrays more than 128. This is because of the fact that we target a batch size of one for all workloads to mimic an online setting, which easily falls short of tile operations that run in parallel in large numbers of arrays. This limitation can be easily avoided by increasing the data size, either by increasing the batch size or running multiple workloads in parallel.

To show the impact of choosing larger batch sizes and running multiple workloads in parallel, we measured the effective throughput of a Resnet-152 and BERT-medium models with varying batch sizes, which is shown in Figure 3.12. We observe that, because the proposed design already achieves an effective throughput close to its peak throughput for the Resnet model, increasing the batch size does not lead to a significant improvement in its effective throughput. In contrast, because the proposed design is underutilized while running the BERT model due to insufficient number of tile operations, increasing the batch size results in a significant improvement in effective throughput. Likewise, running multiple workloads in parallel also increases the number of tile operations. As such, running the Resnet and BERT models in parallel with a batch size of one achieves an effective throughput of 397 TeraOps/s, which is 1.44× higher than running them sequentially.

To understand how the optimal array size changes in multi-workload and multi-batch scenar-

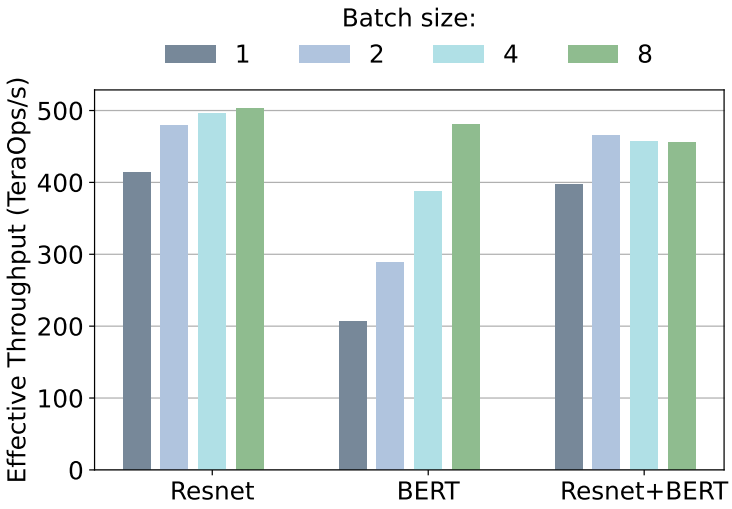


Figure 3.12: Effective throughput of SOSA for varying batch sizes for Resnet only, BERT only, and both Resnet and BERT in parallel.

ios, we perform an ablation study where we vary the batch size and the number of parallel workloads, and calculate the optimal array size using our design space exploration. Table 3.3 shows the optimal array sizes for various batch sizes and numbers of workloads. We observe that increasing the number of parallel workloads or the batch size results in an increase in the optimal array size due to larger model dimensions and higher degree of available parallelism. Therefore, we conclude that the designs that target single-batch and single-workload settings (e.g., real-time applications) should pick a small array size (i.e., 20×32), whereas the designs that target multi-batch and multi-workload settings (e.g., datacenter applications) benefit from picking larger array sizes (e.g., TPU).

3.3.4 Interconnect

To demonstrate the interconnect's impact on a multi-pod system, we measured the effective throughput and calculated the TDP for various numbers of pods and for various types of interconnects, which is shown in Figure 3.13. Crossbar achieves the highest effective throughput

Table 3.3: Optimal array size for varying batch size and number of parallel workloads.

		No. of workloads		
		1	2	4
Batch size	1	20×32	32×64	40×32
	2	40×32	40×32	64×32
	4	40×32	64×32	80×32
	8	80×32	80×32	80×32
	16	80×32	96×64	128×64

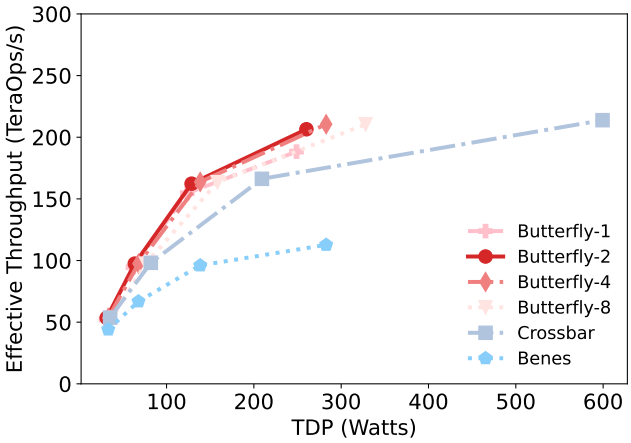


Figure 3.13: Effective throughput versus TDP for various interconnect types. Points represent the number of pods, which are equal to 32, 64, 128, and 256.

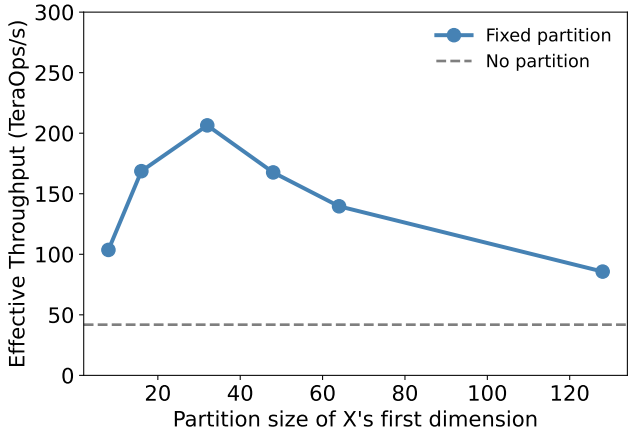


Figure 3.14: Effective throughput versus data partition size for activation matrices.

(213.8 TeraOps/s) thanks to its high connectivity and low latency but it requires around 2.3× more peak power than any other interconnect due to its quadratically increasing hardware cost. Although Benes network offers high connectivity, it performs poorly for the increasing number of pods mainly due to the fact that their long latency becomes exposed, reducing its effective throughput. This experiment confirms that Butterfly is the optimal interconnect for multi-pod systems, which achieves an effective throughput only 4% less than Crossbar but at a much lower TDP.

Figure 3.13 also evaluates the impact of the expansion factor for the Butterfly network. For increasing expansion factors, Butterfly networks are capable of routing more input and output permutations, reducing network contention and improving effective throughput. However, the hardware cost of the interconnect increases with the expansion factor, which reduces its

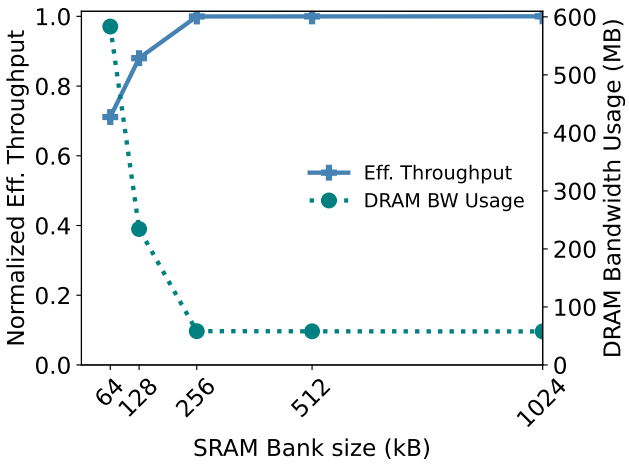


Figure 3.15: Effective throughput (normalized to maximum value) and off-chip DRAM usage for varying on-chip SRAM bank sizes.

effectiveness. Our experiment shows that the expansion factors larger than two achieve only a marginal increment in effective throughput (less than 2%) while the hardware cost of the interconnect doubles with every expansion. Thus, we conclude that Butterfly network with an expansion factor of two is the optimal choice of interconnect, with an effective throughput of 206.5 TeraOps/s at a TDP of 260 Watts.

3.3.5 Tiling

To demonstrate the effect of our proposed tiling strategy quantitatively, we measured the effective throughput for CNN and BERT benchmarks using various partition sizes for activation matrix's first dimension. Figure 3.14 shows the sensitivity in effective throughput with respect to partition size. When the activation matrix's first dimension is not partitioned or the partition size is larger than the number of rows in the systolic array, the effective throughput is suboptimal because the number of tile operation that can run in parallel is not sufficient to keep large numbers of arrays active. When the partition size is smaller than the number of rows, the effective throughput also decreases because the interconnect and buffering latencies become exposed. Therefore, we identify the optimal partition size for the activation matrix as $r \times r$, which maximizes the overall effective throughput in a multi-pod system.

3.3.6 Memory

The effective throughput of the proposed architecture is also sensitive to the on-chip SRAM capacity because an SRAM capacity that is smaller than the workload's active working set may lead to latency and energy overhead due to frequent off-chip DRAM accesses, whereas an unnecessarily large SRAM capacity increases energy consumption and silicon area cost.

To show the effective throughput's sensitivity to on-chip SRAM capacity, we vary the bank size from 64kB to 1MB, and measured the effective throughput and DRAM bandwidth usage, which are shown in Figure 3.15. In this experiment, we used the workload with the largest active working set among all benchmarks, namely Resnet152, with a batch size of eight. We observe that, choosing an SRAM bank size smaller than 256kB results in tile eviction and SRAM misses, which leads to both increased DRAM bandwidth usage, which consequently reduces the effective throughput. As a result, we pick a bank size of 256kB in our proposed design.

3.3.7 RTL Synthesis

To verify our assumptions on hardware parameters, we synthesized the proposed design in Synopsys Design Compiler using TSMC 28nm library. Table 3.4 summarizes our synthesis results. We observe that on-chip SRAM memory constitutes the majority of the power consumption (45.81%) and silicon area (75.37%). The results also demonstrate that the proposed interconnect, namely the Butterfly network with an expansion factor of two, is only 15.06% of the total power consumption and 4.18% of the total silicon area. Unsurprisingly, systolic arrays constitute a large portion of a pod's total power consumption (97.58%) and silicon area (97.82%), whereas the rest is taken by the control logic and buffers. Leakage power constitutes only 1.5% of the total power of systolic pods.

Table 3.4: Power and area breakdown of the proposed architecture for 256 systolic pods. The design is synthesized in Synopsys Design Compiler using the TSMC 28nm library for up to 16 systolic pods and the results are extrapolated for 256 systolic pods.

		Power [%]	Area [%]
SRAM		45.81	75.37
Post-processor		0.56	0.25
Interconnect		15.06	4.18
Systolic Pod	Systolic Array	37.64	19.76
	Job Queue	0.30	0.18
	Act. Buffer	0.07	0.01
	Conv. Buffer	0.19	0.06
	Input Psum Buffer	0.09	0.03
	Output Psum Buffer	0.09	0.03
	Others	0.19	0.13

3.3.8 Comparison to Prior Inference Accelerators

Table 3.5 summarizes the properties of prior inference accelerators as well as the proposed SOSA architecture. We can classify the prior accelerators into two groups based on their TDPs, namely mobile and server accelerators. The mobile accelerators such as Eyerissv2 [Chen et al., 2019], NVDLA [NVIDIA, 2018], and DianNao [Chen et al., 2014b] prioritize low power consumption due to the strict resource constraints of mobile devices; and consequently, they

Table 3.5: Summary of the prior inference accelerators and SOSA. †Results of Resnet50 with a batch size of one. *Implemented on an Intel Arria 10 GX 1150 FPGA. ‡For a fair comparison, we scaled SOSA down to 64 systolic pods to have equal number of PEs with TPUv4.

		Array size	No. of arrays	No. of PEs	Data format	Clk freq. (GHz)	Tech. node	Die area (mm^2)	TDP (Watts)	Peak throughput (TFLOPS/s)	Eff. throughput† (TFLOPS/s)
Mobile	Eyerissv2 [Chen et al., 2019]	3 × 4	16	192	INT8	0.2	65nm	24	<1	0.15	–
	NVDLA [NVIDIA, 2018]	–	–	2k	INT8	1.0	16nm	3.3	0.76	2.0	2.0
	DianNao [Chen et al., 2014a]	–	–	496	INT16	0.98	65nm	3	0.48	0.45	–
Server	Brainwave [Fowers et al., 2018]	–	–	1.5k	BF16	0.3	20nm*	–	60	9.8	4.5
	Simba [Shao et al., 2019]	16 × 8	36	4.6k	INT8	1.8	16nm	216	–	128	16.4
	NVIDIA P4 [Cherlopalle et al., 2019]	–	–	2.5k	INT8/FP16/ FP32	1.53	16nm	314	75	22	13.9
	NVIDIA T4 [Cherlopalle et al., 2019]	4 × 4 × 4	320	20k	INT4/INT8/ FP16/FP32	1.59	12nm	545	70	130	31
	TPUv4 [Jouppi et al., 2021]	128 × 128	4	65k	INT8/BF16	1.05	7nm	<400	175	138	55.8
	SOSA	32 × 32	64‡	65k	INT8	1.0	28nm	160	65	131	110

can exhibit only limited throughput. In contrast, the server accelerators enjoy much higher power envelopes; thus, they can reach up to hundreds of TeraOps/s. However, due to low utilization ratios, the effective throughput of these inference accelerators corresponds only to a fraction of their theoretical peak throughput. In fact, Google' TPUv4 [Jouppi et al., 2021] exhibits the highest effective throughput (55.8 TeraOps/s) among all the server-grade inference accelerators while maintaining a utilization ratio of 40%. As a result of the design optimizations proposed in this thesis, SOSA exhibits a significantly higher resource utilization (84%) than any other server-grade accelerators, achieving an effective throughput that is almost 2× higher than TPUv4.

3.4 Conclusion

In this chapter, we studied the power efficiency and utilization in systolic arrays and identified three key design aspects, namely array granularity, interconnect, and tiling. Our analysis on DNN workloads and design space exploration allowed us to identify the optimal array granularity as 32 × 32 for an accelerator that targets widely used convolutional and transformer DNN models. We also studied various interconnect topologies for their suitability to connect large numbers of systolic arrays on a single chip and identified that the Butterfly topology is the ideal design choice for multi-pod systolic architectures. Moreover, we showed the impact of the tiling strategy on the utilization of a multi-pod systolic architecture and proposed a novel tiling scheme to maximize the utilization across multiple systolic pods.

Based on the insights that we gained on the three key design aspects of systolic arrays, we introduced the scale-out systolic array architecture. We demonstrated that the proposed architecture offers 1.5× higher effective throughput and improves the resource utilization by up to 3× compared to the baseline that is equivalent to widely used TPU designs.

4 Utilization-Aware Neural Architecture Search

In the previous chapter, we addressed the problem of underutilization in systolic arrays due to dimension mismatches between array and DNN dimensions. To improve the resource utilization, we performed a design space exploration to identify the optimal array granularity and proposed a novel hardware architecture. However, the problem of underutilization due to dimension mismatches can also be seen as an optimization problem for DNN architecture. In this chapter, we address the underutilization problem in systolic arrays from the perspective of DNN architectures.

Finding optimal DNN architectures that achieve high accuracy at the given task with minimal computational requirements has been a challenging task for developers. To offload the task of finding optimal DNN architectures from developers to efficient optimization algorithms, prior work has proposed neural architecture search. With the recent advancements in their optimization algorithms and search space, neural architecture search has become an essential tool in designing DNN architecture that can outperform the hand-crafted ones in complex tasks such as image classification or object detection.

The design process of DNN architectures often requires solving a multi-criteria optimization problem that includes the accuracy and computational cost as optimization objectives. To solve this multi-criteria optimization problem, hardware-aware neural architecture search algorithms have been proposed, which are generally formulated using a loss function composed of the accuracy and hardware-related metrics such as latency and energy consumption. While the accuracy metric in the loss function can be easily calculated using a test dataset during the architecture search, the efficiency and accuracy of the hardware-related metrics play an important role in the effectiveness of the hardware-aware neural architecture search algo-

Part of this chapter has been published in
"U-Boost NAS: Utilization-Boosted Differentiable Neural Architecture Search", In European Conference on Computer Vision (ECCV), 2022. [Yüzügüler et al., 2022]

rithms. While prior work has proposed various techniques to estimate the hardware-related metrics for DNN architectures during the search, none of them addresses the leading source of underutilization in systolic arrays, namely dimension mismatches. As a result, the DNN architectures found by the existing hardware-aware neural architecture search algorithms can not exploit the available computational resources the array-based DNN accelerators to the full extent, leading to sub-optimal inference performance.

To design DNN architectures that minimize dimension mismatches on array-based accelerators, in this chapter, we propose a novel framework, namely utilization-boosted neural architecture search (U-Boost). Using our insights about systolic arrays from the previous chapter, we first develop an analytical model for the resource utilization in array-based accelerators. Then, we propose a smooth approximation for the utilization model to integrate it with the efficient differentiable neural architecture search algorithms. Moreover, we extend the loss function of the existing hardware-aware neural architecture search frameworks with a utilization term and search for DNN architectures that maximize the effective usage of the hardware resources at the target inference platforms. Our experiments on popular computer vision tasks as well as our extensive hardware simulations show that the proposed U-Boost framework significantly improves the resource utilization at target platforms, leading to shorter inference time and/or higher accuracy.

The rest of this chapter is organized as follows: We first introduce our analytical model for resource utilization in array-based accelerators. Then, we give details about the proposed neural architecture search framework. Finally, we elaborate on our experimental details and discuss their results.

4.1 Analytical Model for Resource Utilization in Systolic Arrays

Prior hardware-aware neural architecture search frameworks often use inference latency as a hardware performance metric in their loss functions. While this approach can easily reduce the inference latency by limiting the number of layers in DNN architectures, the resulting DNN architectures do not necessarily correspond to those with high resource utilization at target inference platforms. Therefore, we adopt a different approach and use both latency and utilization as hardware performance metrics in the loss function. Because the underutilization problem is especially evident in systolic arrays due to mismatches between DNN architectures and array dimensions, in this section, we develop an analytical model for the resource utilization in systolic arrays.

As we already discussed in the previous chapter, systolic arrays consist of a two-dimensional grid of processing elements, as shown in Figure 3.1. Let us assume a systolic array of $r \times c$ with a weight stationary dataflow. In this dataflow, the weights are spatially mapped onto the array, activations are streamed along the rows, and partial sums are accumulated along the columns [Jouppi et al., 2017]. In this dataflow, any mismatch between the array and weight dimensions results in idle rows and columns of processing elements, which reduces

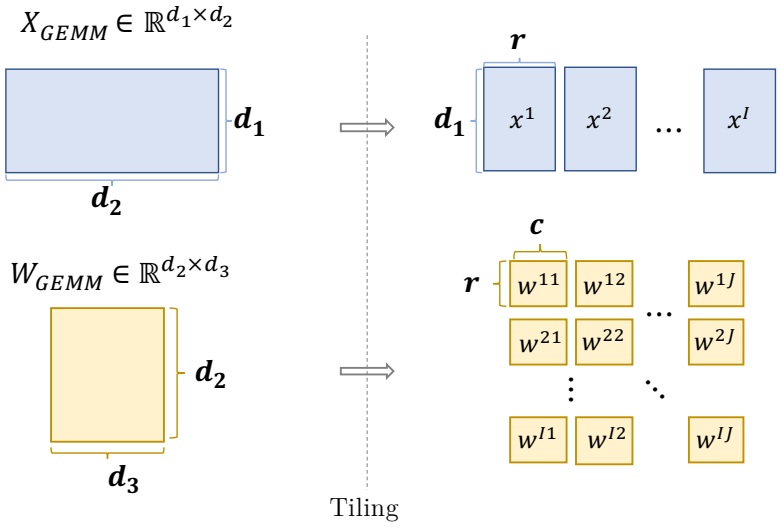


Figure 4.1: Tiling of a GEMM operation onto a systolic array.

the array utilization. Although we consider the weight stationary dataflow in this thesis due to its widespread adoption, the same phenomenon occurs in other dataflows as well; thus, our method is applicable to other dataflows without loss of generality.

Let us consider the GEMM operation $Y_{GEMM} = X_{GEMM} W_{GEMM}$, where the operands are $X_{GEMM} \in \mathbb{R}^{d_1 \times d_2}$ and $W_{GEMM} \in \mathbb{R}^{d_2 \times d_3}$. In case $d_2 > r$ or $d_3 > c$, X_{GEMM} and W_{GEMM} must be divided into smaller tiles for this GEMM operation to be mapped onto the systolic array, as shown in Figure 4.1. In this tiling process, X_{GEMM} is divided into I tiles of $x \in \mathbb{R}^{d_1 \times r}$ and W_{GEMM} is divided into $I \times J$ tiles of $w \in \mathbb{R}^{r \times c}$ as follows:

$$X_{GEMM} = \begin{bmatrix} x^1 & \dots & x^I \end{bmatrix}, \quad W_{GEMM} = \begin{bmatrix} w^{11} & \dots & w^{1J} \\ \vdots & \ddots & \vdots \\ w^{I1} & \dots & w^{IJ} \end{bmatrix}$$

With this tiling scheme, the output Y_{GEMM} can be obtained as:

$$Y_{GEMM} = \begin{bmatrix} y^1 & \dots & y^J \end{bmatrix}$$

where $y^j \in \mathbb{R}^{d_1 \times c}$ is equal to:

$$y^j = \sum_{i=1}^{I} x^i w^{ij}$$

As such, the output can be calculated in $I \times J$ tile operations and I and J are equal to:

$$I = \left\lceil \frac{d_2}{r} \right\rceil, \quad J = \left\lceil \frac{d_3}{c} \right\rceil$$

where the ceil function is defined as $\lceil x \rceil = \min\{n \in \mathbb{Z} : n \geq x\}$.

While performing a tile operation $x^i w^{ij}$, a systolic array fetches one row of x^i in each cycle. Therefore, each tile operation takes as many cycles as the number of rows in x^i, namely d_1. Multiplying the cycles per tile operation by the number of tile operations, we obtain the total execution runtime (latency) in terms of the number of cycles as follows:

$$\texttt{RUNTIME} = d_1 \left\lceil \frac{d_2}{r} \right\rceil \left\lceil \frac{d_3}{c} \right\rceil \tag{4.1}$$

The effective throughput (i.e., operations per unit time) is equal to the ratio of the number of operations to the execution time. Given that the number of multiply-and-accumulate operations needed to perform the matrix multiplication of $X_{GEMM} W_{GEMM}$ is equal to $d_1 \times d_2 \times d_3$ and using the RUNTIME given in Equation 4.1, we calculate the effective throughput in terms of MAC per cycle as:

$$\texttt{EFF. THROUGHPUT} = \frac{d_2 d_3}{\left\lceil \frac{d_2}{r} \right\rceil \left\lceil \frac{d_3}{c} \right\rceil}$$

The utilization of processing elements can be simply calculated as the ratio of the effective throughput to the peak throughput. Given that the theoretical peak throughput of a systolic array is $r \times c$ MAC operations per cycle, we finally obtain the utilization of a systolic array as:

$$\texttt{UTIL} = \frac{d_2 d_3}{r c \left\lceil \frac{d_2}{r} \right\rceil \left\lceil \frac{d_3}{c} \right\rceil} \tag{4.2}$$

Table 4.1: Utilizations and runtimes for various types of DNN layers.

Block Type	Runtime	Utilization
Convolution	$\left\lceil \frac{k_1 k_2 g}{r} \right\rceil \left\lceil \frac{f}{c} \right\rceil hwb$	$\frac{k_1 k_2 g f}{rc \left\lceil \frac{k_1 k_2 g}{r} \right\rceil \left\lceil \frac{f}{c} \right\rceil}$
Depthwise Convolution	$g \left\lceil \frac{k_1 k_2}{r} \right\rceil fhwb$	$\frac{k_1 k_2}{\left\lceil \frac{k_1 k_2}{r} \right\rceil f}$
Fully connected	$\left\lceil \frac{g}{r} \right\rceil \left\lceil \frac{f}{c} \right\rceil b$	$\frac{gf}{rc \left\lceil \frac{g}{r} \right\rceil \left\lceil \frac{f}{c} \right\rceil}$

The expression derived above gives us insights about how dimension mismatches affect the utilization in systolic arrays. Consider the case where the operand dimensions exactly match the array dimensions: $d_2 = r$ and $d_3 = c$. Then, Equation 4.2 simplifies to a utilization of 1.0, which indicates that the systolic array runs at full capacity. However, if the operand dimensions are slightly increased, for instance $d_2 = r + 1$, the ceil function reveals a significant drop in utilization since $\left\lceil \frac{d_2}{r} \right\rceil = \left\lceil \frac{r+1}{r} \right\rceil = 2$, resulting in a utilization of about 0.5. In other words, a slight modification in operand dimensions may lead to a significant change in hardware utilization in systolic arrays.

The utilization term derived in Equation 4.2 is a function of the operand dimensions of a GEMM operation. Table 4.1 summarizes the expressions for the runtime and utilization terms for various types of DNN layers in terms of layer dimensions, where k_1 and k_2 are kernel size, f is the number of filters, g is the number of input channels, h and w are the input image size, and b is the batch size. The expressions derived for different type of DNN layers also indicates that hardware utilization in systolic arrays is also highly sensitive to the layer type. For instance, depthwise convolutional layers [Sandler et al., 2018], which are widely used in mobile applications, have only a single filter ($f = 1$) and perform convolution operations channel-by-channel. As a result, depthwise convolutional layers require matrix multiplications with dimensions equal to the $hwb \times k_1 k_2$ and $k_1 k_2 \times 1$, which is much smaller than the standard convolutional layers. The small matrix dimensions inherent to depthwise convolution often lead to a hardware utilization as low as 1% [Cho, 2021; Gupta & Akin, 2020], which reduces their inference performance in systolic. Therefore, selecting layer types with high utilization is also crucial to obtain DNN architectures that perform well in systolic arrays.

To validate the proposed utilization model and to demonstrate the impact of channel dimensions on hardware utilization, we performed dense and convolutional DNN inference with varying numbers of output channels on a Cloud TPU v2 and measured the runtime and utilization values using Google Cloud's XLA `op_profiler` tool. Figure 4.2 shows the result of our experiment as well as estimated values with the proposed and roofline [Williams et al., 2009] models. Because Cloud TPUv2 have an array size of 128×128, we observe significant drops in

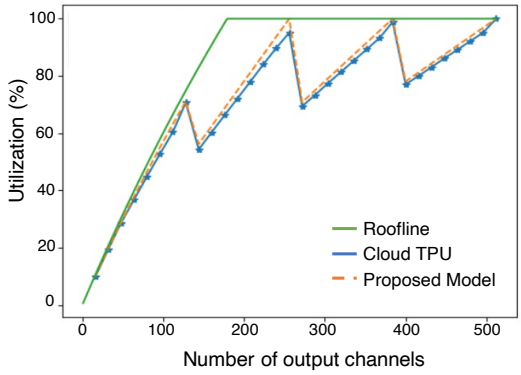


Figure 4.2: Measured utilization on Cloud TPUv2 versus predicted utilization with roofline and the proposed model.

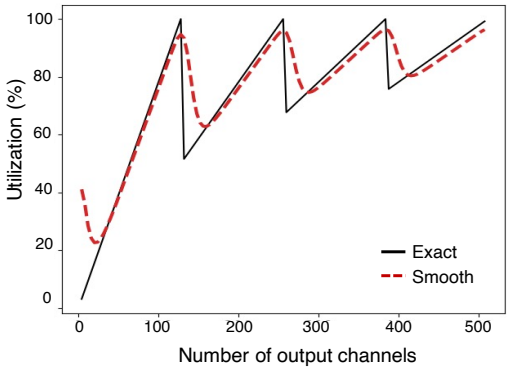


Figure 4.3: Proposed utilization model with exact ceil function and its smooth approximation using the generalised logistic function.

utilization when the channel dimensions exceed multiples of 128. The roofline model, which accounts only for memory bottleneck, does not capture these drops in utilization, leading to a discrepancy up to 40% between measured and estimated values. The proposed utilization model, however, accounts for the dimension mismatches and is therefore able to estimate the actual utilization value with an error of only up to 2%.

4.2 Proposed NAS Framework

Using the proposed utilization model, we introduce a utilization-aware differentiable NAS framework. In this Section, we first explain how we approximate the proposed utilization model, then we formulate our multi-objective loss function, and finally, we describe the NAS algorithm used to search optimal DNN architectures.

4.2.1 Approximation of the utilization function

The ceil function in Equation 4.1 is not differentiable and can only be used as a collection of point estimates. This limits the effectiveness of the neural architecture search and allows only for evolutionary or reinforcement learning methods, which require orders of magnitude more computational resources compared to differentiable methods. For this reason, we use the generalised logistic function to obtain a smooth approximation of ceil function:

$$\texttt{CEIL}_{smooth}(x) = \sum_i \left[1 + \frac{\exp\left(-B(x - w_i)\right)}{C}\right]^{-1/v}$$

where w_i are intervals between zero and a fixed value; C, B, and v are constants that adjust the

smoothness of the approximation. We empirically selected $C = 0.2$, $B = 20$, and $\nu = 0.5$, which leads to a smooth and accurate approximation of the original ceil function. Figure 4.3 shows a comparison between the true utilization, denoted as *hard*, and its smooth counterpart. We verify that both hard and smooth utilization models yield peak utilization values at the same channel dimensions. Therefore, we replace the original utilization model with its smooth approximation in the proposed NAS framework.

4.2.2 Multi-objective loss function

Let $\mathscr{F}$ be the hypothesis class of neural networks that characterizes the search space. The candidate neural network $\alpha \in \mathscr{F}$ implements the function $f_\alpha : \mathscr{X} \rightarrow \mathscr{Y}$ where $\mathscr{X}$ and $\mathscr{Y}$ are the domains of the input and the output for our dataset $\mathscr{D}$, respectively. Let $(\boldsymbol{x}, y) \in \mathscr{X} \times \mathscr{Y}$ be a sample. Then the loss function consists of three terms:

$$\mathscr{L}(\boldsymbol{x}, y, \alpha) = \mathscr{L}_{classification}(f_\alpha(\boldsymbol{x}), y) + \lambda \cdot \mathscr{L}_{latency}(\alpha) - \beta \cdot \mathscr{L}_{utilization}(\alpha) \tag{4.3}$$

where $\lambda > 0$ and $\beta > 0$ determine the tradeoff between the accuracy, latency and utilization. The classification loss corresponds to cross-entropy, while the latency and utilization terms have been discussed in the previous section.

4.2.3 NAS algorithm

The search algorithm employs a hierarchical search similar to prior work [Liu et al., 2019; Wan et al., 2020]. Concretely, it consists of three stages: microarchitecture search, macroarchitecture search and training of the selected architecture $\alpha \in \mathscr{F}$. The first stage searches for layer types and connections using a model of a single cell and fixed channel dimensions. After obtaining the optimal candidate cell, the macroarchitecture stage constructs a model with k sequential cells sequentially and searches for the optimal channel dimensions cell-wise using the Dmasking method [Wan et al., 2020]. In both stages, each architectural decision (i.e, type of operator in the former and number of channels in the latter) is modelled by a probability simplex of dimension m equal to the number of choices and is parameterized by Gumbel-Softmax [Jang et al., 2017].

4.3 Experiments

To evaluate the effectivenes of the proposed method, we perform image classification experiments on the `CIFAR10` and `ImageNet100` datasets and compare our results with prior work. In this section, we first explain our experimental setup, then analyse the characteristics of the DNN architectures obtained with the proposed method, and finally, report and discuss the performance results of our experiments.

Experimental setup

We perform experiments on widely used computer vision datasets, namely `CIFAR10` [Krizhevsky, 2009] and `ImageNet100`, which is a subset of the Imagenet (ILSVRC 2012) classification dataset [Deng et al., 2009] with randomly-selected 100 classes. As in prior work [Liu et al., 2019; Wu et al., 2019], the optimal-architecture search stage for both datasets is performed on a proxy dataset, namely `CIFAR10`. We compare the results of our proposed method against three hardware-aware NAS methods that use *FLOPS* [Gordon et al., 2018], *Roofline* [Li et al., 2021a], and *Blackbox* [Wu et al., 2019] models to estimate the latency. In FLOPS baseline, we simply calculate the latency as the number of operations required to perform inference divided by the theoretical peak throughput of inference platform assuming full-utilization. In Roofline baseline, we consider two modes, namely memory-bound and compute-bound. While the compute-bound mode is the same as the FLOPS baseline, in memory-bound mode, we calculate the latency as the memory footprint size divided by the off-chip bandwidth. In Blackbox baseline, we fill a lookup table with latency values for all layer types and dimensions with a quantization of 16 obtained with the hardware simulator, and retrieve these values during architecture search using nearest-neighbor interpolation.

Search Space

The cell architecture and search space are inspired by the DARTS architecture [Liu et al., 2019] with a few minor modifications. In all search and training stages, the candidate architecture consists of a preparatory block, k stack of cells, and a fully connected classifier. Each cell is a multigraph whose edges represent different operators, including depthwise separable, dilated, and standard convolutional layers as well as identity and zero operations corresponding to residual and no connections, respectively. Candidate kernel sizes for all convolutional layers are 3×3 and 5×5. Each cell has two input nodes connected to the output nodes of two previous cells. Each convolution operation has a stride of 1 and is followed by batch normalization and ReLU activation functions. The channel search space corresponds to a dimension range of 64 to 280 with increments of 8. For `CIFAR10`, we use a stack of three cells ($k = 3$), each of which is followed by a 2×2 maxpooling layer. To accomodate the increased complexity of `ImageNet100`, we use a stack of nine cells ($k = 9$), where only one of every three cells is followed by maxpooling. More details about the search space are given in appendix.

NAS settings

During the microarchitecture and channel search stages, the first 80% of the batches of each epoch is used to train model weights, while the last 20% is used to train the architectural parameters using a batch size of 64. The weights are optimized with Stochastic Gradient Descent (SGD) with learning rate 0.05, momentum 0.9 and weight decay $3e-4$, while the architectural parameters use Adam [Kingma & Ba, 2015] with learning rate 0.1. The microarchitecture and channel search stages last 10 and 30 epochs, respectively. To improve convergence, the

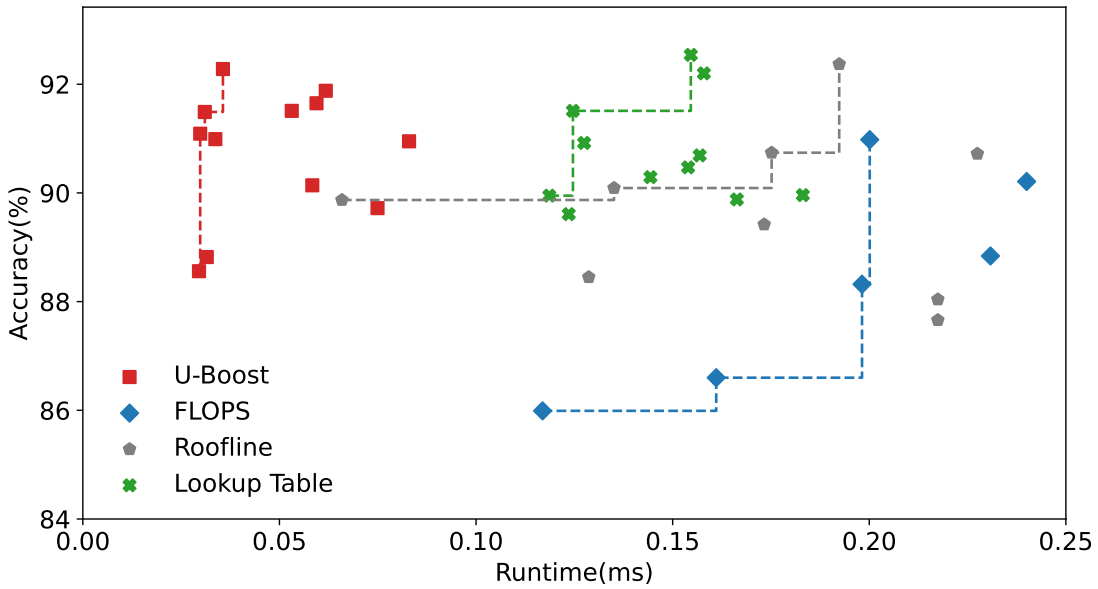


Figure 4.4: Experiments on CIFAR10 dataset. Upper left corner is optimal. The dashed lines connect the points in the Pareto Front of each method.

temperature parameter τ of the Gumbel-Softmax is annealed exponentially by 0.95 per epoch from the initial value of 1. For fairness, we use the same NAS algorithm and hyperparameters for all baselines and the proposed method. After the search stages are completed, the selected DNN architecture is trained from scratch. In `CIFAR10` experiments, we train the models for 200 epochs with a batch size of 64 using the original image resolution of 32×32. In `ImageNet100` experiments, we train the models for 70 epochs with a batch size of 256 using an input resolution of 128×128. For both datasets, we use a preprocessing stage consisting of normalization, random crop and vertical flip.

Metrics

For all experiments, we report top-1 classification accuracy from the test datasets. Runtime and utilization values are measured by running the DNN models on our custom-made cycle-accurate hardware simulator. Correctness of our hardware simulator is validated against an RTL design of a systolic array architecture. During the hardware simulations, we assumed an array size of 128×128 as in Cloud TPUv4 [Jouppi et al., 2020] with a 15 MB on-chip memory and an 80 GB/s off-chip memory bandwidth and 1 GHz clock frequency. To quantify the trade-off between accuracy and latency, we calculate the hypervolume score [Zitzler & Thiele, 1999], which is calculated as the volume of the union of axis-aligned rectangles from each point in a Pareto front [Désidéri, 2012]. We select the reference point to calculate the hypervolume score as the perfect oracle: 100% accuracy with zero runtime. Consequently, lower scores indicate design points that are close to the ideal.

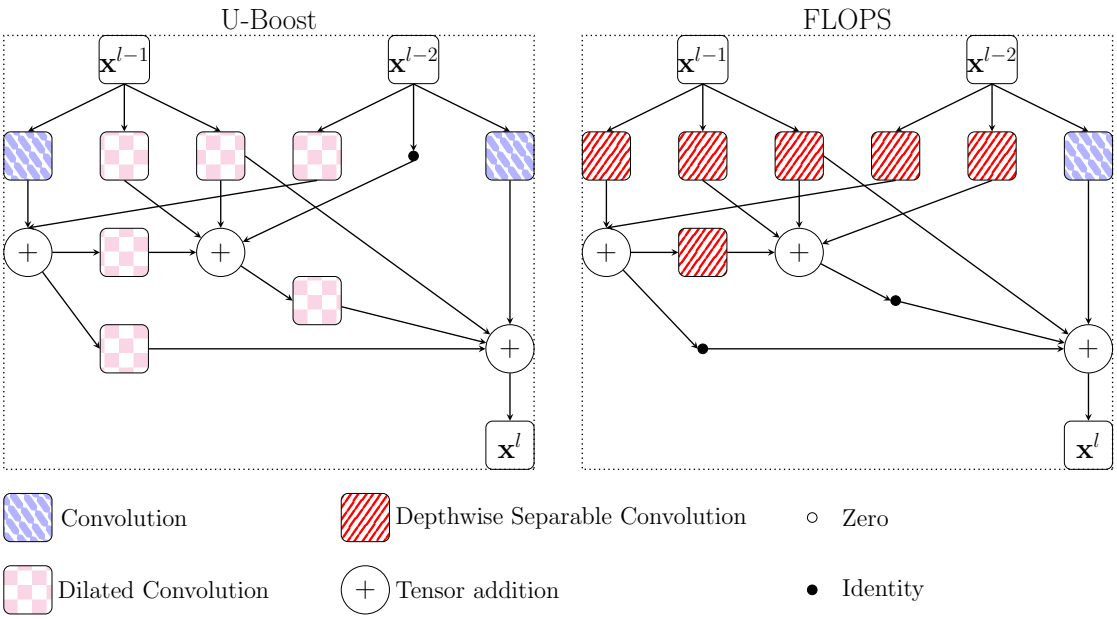


Figure 4.5: Visualization of the `CIFAR10` cells obtained from U-Boost and FLOPS models during the microarchitecture search stage.

4.3.1 `CIFAR10` experiments

To evaluate the proposed method on `CIFAR10` dataset, we set the utilization coefficient $\beta = 1$ in Equation 4.3 and vary the latency coefficient $\lambda \in \{0.1, 0.5, 1, 5\}$ for all baselines to control accuracy-latency trade-off. Figure 4.4 shows the accuracy and latency of the DNN architectures found by the proposed method and baselines. We observe that U-Boost significantly improves the accuracy-latency Pareto front with a $2.8-4\times$ speedup in runtime compared to baseline methods while achieving comparable accuracy. The improvement in the Pareto front is also reflected in the hypervolume metric: U-Boost has a hypervolume of 0.39 whereas FLOPS, Roofline, and Blackbox baselines have hypervolumes of 2.68, 1.86, and 1.47, respectively, corresponding to an improvement in the range of $3.7-6.8\times$.

The reason why U-Boost achieves better accuracy-latency Pareto front is mainly because the selected cell microarchitecture and channel dimensions are well-suited for the target inference platform. To validate this insight, we analyze and compare the cell microarchitecture and channel dimensions selected by U-Boost and other baselines. Figure 4.5 depicts examples of cell microarchitectures selected by U-Boost and FLOPS baseline. We observe that the cell microarchitecture selected by FLOPS baseline mostly consists of depthwise separable convolutional layers because they require a smaller number of operations. However, these layers run at low utilization at the inference platforms, which increases their latency. By contrast, the cell microarchitecture selected by U-Boost consists of standard or dilated convolutional layers because U-Boost is utilization-aware and it chooses layers that run at higher utilization in target platforms, reducing the latency.

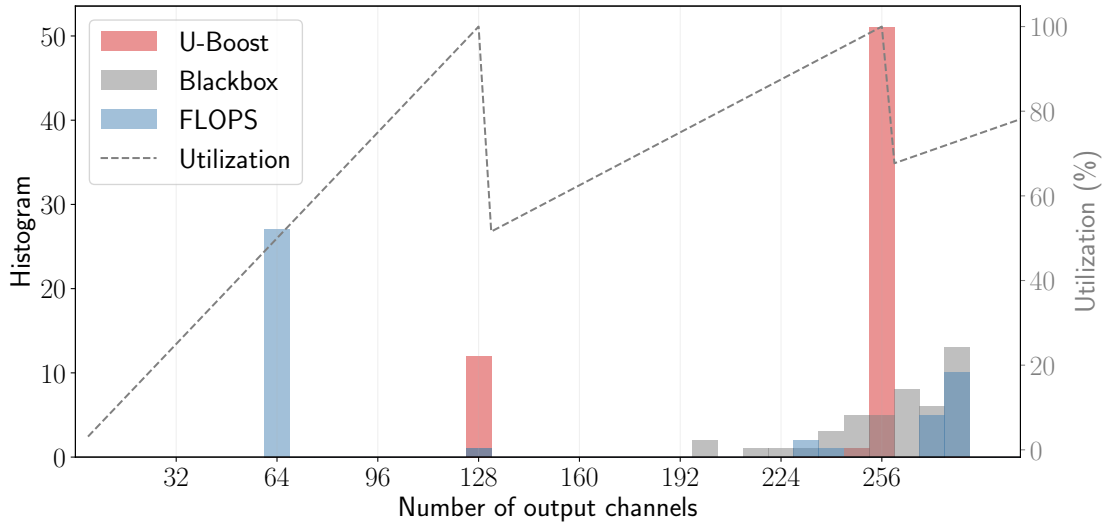


Figure 4.6: Histogram of channel dimensions found by U-Boost as well as FLOPS and Blackbox baselines on `CIFAR10` dataset.

Besides the cell microarchitecture, we also analyze the channel dimensions selected by the U-Boost and other baselines. Figure 4.6 shows the histogram of channel dimensions selected by U-Boost, FLOPS, and Blackbox baselines. We observe that the channel dimensions selected by FLOPS and Blackbox baselines are mostly concentrated on each end of the search space, which is bounded by channel dimensions of 64 and 280, rather than dimensions that correspond to high utilization. As a consequence, DNN architectures with such layers run at low utilization in target inference platforms. Unlike FLOPS and Blackbox baselines, we observe that the channel dimensions selected by U-Boost are concentrated on either 128 or 256, which are multiples of the array size and correspond to high utilization. As such, the DNN architectures selected by U-Boost run at high utilization, accelerating the inference at target platforms.

4.3.2 `ImageNet100` experiments

To show the effectiveness of the proposed method on a more complex dataset, we also perform a set of experiments on `ImageNet100`. For this set of experiments, we set the latency coefficient $\lambda \in \{0.1, 1.0, 5.0\}$ to control the accuracy-latency tradeoff. Table 4.2 reports the results of these experiments. We observe that FLOPS and Roofline baselines result in poor inference hardware utilization ($< 10\%$) as they estimate hardware performance inaccurately during the architecture search. The second best method in terms of utilization, namely Blackbox, improves the hardware utilization to 69% as it can estimate the hardware performance accurately during the search. Still, around 30% of hardware resources remain unutilized during inference as the Blackbox method can not find the optimal channel dimension since it operates on a discrete search space and is unable to exploit gradient information to successfully navigate the search.

By contrast, the proposed U-Boost method, which both estimates the hardware performance accurately and uses the information from gradients to find the optimal cell microarchitecture and channel dimensions, achieves inference hardware utilization up to 91%, which is 1.3× higher than the second best baseline. Consequently, DNN architectures obtained with U-Boost achieve the best top-1 accuracy (87.9%), which is 0.1%, 0.7%, and 1.4% higher than the best of Blackbox, FLOPS, and Roofline baselines, respectively, while achieving speedups of 2.1× and 3.3× compared to the second best baselines across λ values. These results reiterate the importance of incorporating and correctly modeling utilization in hardware-aware NAS.

Table 4.2: Experimental results for `ImageNet100` experiments. Underlined measurements show best per column (λ), bold show best per metric. Number of parameters reported in millions.

	Accuracy (%, ↑)			Runtime (ms, ↓)			HV (↓)
	$\lambda = 0.1$	$\lambda = 1.0$	$\lambda = 5.0$	$\lambda = 0.1$	$\lambda = 1.0$	$\lambda = 5.0$	(across λ)
Blackbox	87.5	87.8	<u>87.9</u>	4.8	4.05	3.8	45.98
Roofline	86.5	84.0	74.2	4.7	3.5	2.9	100.62
FLOPS	87.2	78.4	80.2	6.1	3.45	3.42	102.02
U-Boost	<u>87.8</u>	<u>87.9</u>	86.3	<u>2.2</u>	<u>1.05</u>	<u>0.77</u>	**13.94**

4.3.3 Sensitivity to array size

In the experiments so far, we assumed an array size of 128 × 128 due to its adoption in commercially available DNN accelerators [Jouppi et al., 2021]. However, in Chapter 3, we showed that the utilization is highly sensitive to the array size. Therefore, we also investigate the proposed method's sensitivity to the array size of the target DNN accelerators. For this purpose, we repeat our experiments on `CIFAR10` for varying array sizes and measure the speedup that we obtain using the proposed U-Boost method over the FLOPS baseline. We measure the speedup by comparing the runtimes of these two methods' Pareto fronts at the classification accuracy of 91%.

Figure 4.7 shows the speedups obtained for various array sizes including the optimal array size that we found in Chapter 3, namely 32 × 32. In Chapter 3, we made the observation that the systolic architectures with smaller arrays sizes suffer less from the underutilization problem. Aligned with this observation, the speedup that we achieve using the proposed U-Boost method decreases with the array size from 4.2× at the array size of 128 × 128 to 1.3× at the array size of 32 × 32. Nevertheless, this experiment shows the importance of utilization-aware neural architecture search as the proposed U-Boost method improves the runtime of DNN inference by a factor of 1.3× even on a systolic array that is optimized for utilization.

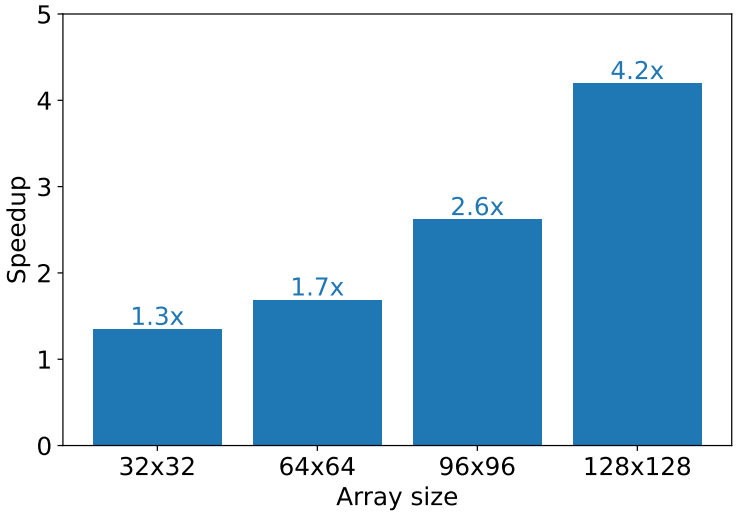


Figure 4.7: Speedups obtained using U-Boost over the FLOPS baseline for various array sizes.

4.4 Conclusion

In this chapter, we have illustrated the importance of resource utilization in runtime characteristics on target inference platforms. We demonstrated that by optimizing DNN architectures in terms of resource utilization as well as task accuracy and latency, we achieve significant improvement in accuracy-latency Pareto front. We proposed a utilization-aware differentiable neural architecture search method, namely U-Boost. We provided an analytical model for resource utilization in widely used array-based hardware accelerators, which allows estimating the utilization efficiently and accurately during the architecture search. Through extensive experiments on popular computer vision datasets and detailed hardware simulations, we showed that the proposed U-Boost NAS method achieves $2.8-4\times$ inference latency speedup with similar or improved accuracy, compared to utilization-agnostic methods. This work highlights the importance of a holistic approach for hardware-aware neural architecture search method and the proposed method enables the design of DNNs with improved performance in inference accelerators.

5 Flexible Channel Dimensions for Differentiable Architecture Search

5.1 Introduction

In Chapter 4, we showed that neural architecture search can effectively find DNN architectures that maximize resource utilization on target inference platforms. When we investigate the DNN architectures found by U-Boost, we observe that the channel dimensions (i.e., the number of features in the input and output of DNN layers) are equal to the multiples of the array dimensions as expected. This was possible because we deliberately design the search space of U-Boost to include the channel dimensions with maximum resource utilization using our domain knowledge on systolic arrays. However, the reliance on the domain knowledge and the requirement of a manually designed search space partially defeat the purpose of automatic search of DNN architectures. Therefore, in this section, we focus on developing a neural architecture search method that can find optimal channel dimensions without the need for manually designing the search space for channel dimensions.

The channel dimensions of DNN architectures do not only influence the resource utilization in systolic arrays but also other performance metrics (e.g., latency) of inference platforms in general. On the one hand, larger channel dimensions correspond to DNNs with more parameters and often lead to higher accuracy in the given task. On the other hand, DNNs with larger channel dimensions require more computation and memory to perform the inference task. As such, designers need to carefully tune the channel dimensions of DNN architectures to achieve the desired accuracy under the given resource constraints of the target inference platforms.

Many of the widely popular DNN architectures such as Alexnet [Krizhevsky et al., 2012], ResNet [He et al., 2016], and InceptionNet [Szegedy et al., 2016] have channel dimensions that follow a specific design pattern, where the channel dimensions are increased with the depth of the network by doubling them in certain layers. However, this design pattern has been created heuristically and does not necessarily correspond to the optimal design choice. In fact, prior work [Gordon et al., 2018] has demonstrated that optimizing the channel dimensions of DNN layers individually using an iterative process can improve the accuracy significantly.

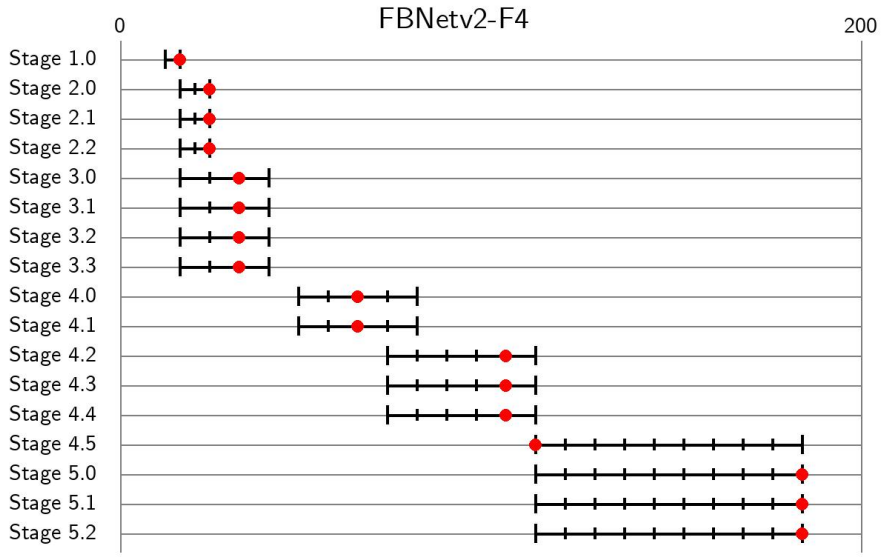


Figure 5.1: Prior work's search space for channel dimensions [Wan et al., 2020]. Rows correspond to the channel range (between 0 and 200) of the layers in FBNetv2-F4. Ticks denote the options for channel dimensions and red circles represent the channel dimensions found.

Moreover, other neural architecture search methods also adopt iterative optimization algorithms such as Bayesian optimization [Bergstra et al., 2011] and Evolutionary Algorithms (e.g., Nevergrad [Rapin & Teytaud, 2018]). Unfortunately, such iterative process requires several costly trials where the networks need to be trained for a number of epochs, which has an overwhelming computational complexity.

To mitigate the high computational cost of iterative optimization frameworks, prior work proposed differentiable neural architecture search methods, which allows to optimize the architectural parameters that define the properties of network in a single training run [Liu et al., 2019; Wu et al., 2019]. Compared to the iterative optimization frameworks, differentiable neural architecture search methods achieve a reduction in computational cost by a factor of 2-3 orders of magnitude. This brings down the cost of DNN architecture optimization to only a few GPU-hours [Dong & Yang, 2019a] and make it available for a wide range of applications. While the early differentiable methods mostly focused on searching for cell structures (i.e., basic building blocks of a DNN architecure) [Liu et al., 2019; Wu et al., 2019], the substantial improvement in efficiency also encouraged researchers to apply the same principles in the search for channel dimensions, giving rise to the differentiable frameworks such as DMaskingNAS [Wan et al., 2020] that can search channel dimensions efficiently using a gradient descent optimizer.

Despite their potential to find optimal channel dimensions with great efficiency, the existing differentiable neural architecture search frameworks require a well-designed search space. Figure 5.1 shows the search space of an existing differentiable neural architecture search framework for channel dimensions [Wan et al., 2020]. We observe that the channel ranges in

this search space are narrowed down to a few options out of a wide range using heuristics, while allowing only a limited degree of freedom for the optimizer. Unfortunately, determining the channel range of each layer is nontrivial as it requires expert knowledge on DNN architectures. Also, the resulting search space is specific to the problem that it is designed for, so it may not be transferable to the settings with different objectives and constraints. Moreover, when we investigate the channel dimensions found within this search space, as shown by red circles in Figure 5.1, we observe that many channel dimensions are located at the boundary of the search space, which indicates that the optimal channel dimensions mostly lie outside of the engineered search space, resulting in a DNN architecture that does not correspond to the optimal solution for the given objectives and constraints.

A trivial solution to the problems inherent to the fixed search space would be to choose excessively large channel ranges in order to increase the likelihood of finding the optimal channel dimensions. However, expanding the range would require training a larger supernet, which would linearly increase the amount of computation to be performed during the search. As a result, the trivial solution of excessively expanding the channel range would exacerbate the computational cost of the search. Therefore, the fixed search space in the existing differentiable neural architecture search frameworks are either unpractical or computationally inefficient while searching for the optimal channel dimensions.

In this chapter, we propose a novel differentiable neural architecture search framework that searches optimal channel dimensions within a flexible range. We reformulate the problem of differentiable channel search to enable adapting the search space freely and seamlessly based on the progress during the search phase. Moreover, we also propose a novel dynamic channel allocation mechanism that appends new channels to the layers of the supernet and frees those that are no longer used in order to reduce memory size and increase the efficiency of the search. Our experiments show that the proposed differentiable neural architecture search framework finds optimal channel dimensions for DNN architectures without any need for a manually-engineered search space and with faster search time than existing channel masking methods with fixed ranges.

The rest of this chapter is organized as follows: We first give a background information on the differentiable channel masking method, which is the basis of our work. We then elaborate on the proposed FlexCHarts method as well as our novel dynamic channel allocation mechanism. Then, we give the details of our experiments and discuss the results. Finally, we conclude this chapter with a summary.

5.2 Differentiable Channel Masking

In standard search methods, each candidate channel dimension requires an additional convolutional kernel in the supernet, which increases the computational cost and memory requirements of the search linearly with the number of channel options. The channel masking method, on the other hand, simulates various candidate channel dimensions on a single

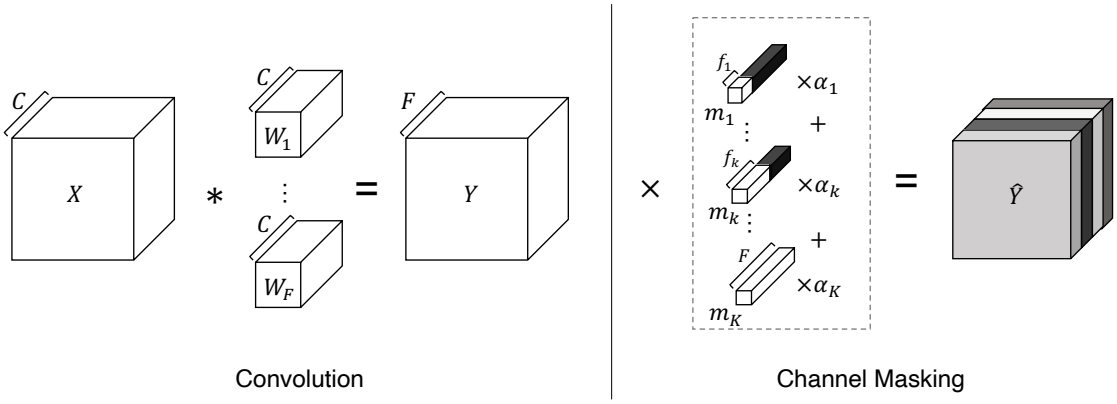


Figure 5.2: Illustration of a convolutional operation followed by channel masking to simulate various output channel dimensions.

overparameterized kernel, incurring only minimal computational overhead. Because of its computational efficiency, we use the channel masking method as a basis for the proposed framework.

Let us assume that we have K candidate channel dimensions for the output of a convolutional layer where $\{f_k \mid f_k \in \mathbb{Z}, k \in \mathbb{Z}, 1 \leq k \leq K\}$ denotes the set of possible channel dimensions. The channel masking method defines a set of trainable parameters $\{\alpha_k \mid \alpha_k \in \mathbb{R}, 1 \leq k \leq K\}$, where α_k corresponds to the weight assigned to the channel dimension f_k.

Figure 5.2 illustrates how the output of a convolutional layer with multiple candidate channel dimensions is calculated with the channel masking method. Let us assume that input activation (X) and weight (W) tensors have channel dimensions of C and the output of the convolutional operation (Y) has a channel dimension of F, which is equal to the number of filters. The channel masking method exploits the fact that any convolutional layer with an output channel dimension f_k that is smaller than F can be obtained simply by selecting f_k channels from Y and masking out the rest. For this purpose, the channel masking method instantiates a set of masks $\{m_k \mid k \in \mathbb{Z}, 1 \leq k \leq K\}$, where $m_k = (1)_{i=1}^{f_k} \cup (0)_{i=f_k+1}^{F}$. In other words, the first f_k elements of m_k are one whereas the remaining elements are zero; thus, the m_k allows selecting the first f_k channels of Y and zeroes out the channels that are greater than f_k.

During the search phase, the DMaskingNAS method multiplies the output activation Y by the masks m_k and calculate $\hat{Y}$, which is the weighted sum of the output of simulated layers with various channel dimension using the following formula:

$$\hat{Y} = \sum_{k=1}^{K} g_\tau(\alpha_k) m_k Y \tag{5.1}$$

where g_τ is a Gumbel softmax function with the temperature constant τ that maps the α

values between 0 and 1 [Wan et al., 2020]. The expression in Equation 5.1 can be simplified by taking Y out of the summation, reducing the overhead of masking to only a weighted sum of low-dimensional masks. Therefore, the channel masking method simulates multiple channel dimensions with negligible computational overhead.

During the search phase, the channel masking method updates the α values with a gradient descent optimizer by minimizing the following loss function:

$$\min_{\alpha} \min_{W} \mathscr{L}_{acc}(\mathscr{N}_{\alpha,W}(x), y) + \lambda \mathscr{L}_{latency}(\mathscr{N}_{\alpha,W}) \tag{5.2}$$

where $\mathscr{N}_{\alpha,W}$ represents the supernet, and x and y represent training samples and ground-truth, and $\mathscr{L}_{acc}$ and $\mathscr{L}_{latency}$ represent the loss functions for classification accuracy and latency, respectively. The coefficient λ controls the trade-off between accuracy and latency. As suggested by the prior work [Liu et al., 2019], the loss function is minimized by calculating the gradients using a first-order approximation in order to reduce the computational cost of the search. At the end of the search phase, the final channel dimensions are selected as the channel dimensions that correspond to the maximum α_i, where $i = \arg\max_k \alpha_k$.

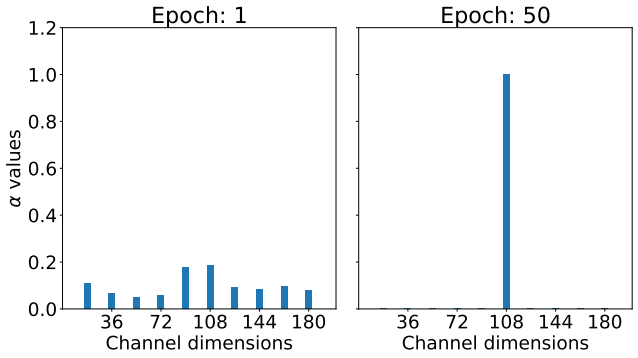
The channel masking method allows searching for channel dimensions among various options with minimal computational overhead. However, the standard channel masking methods proposed in prior work can search only within a fixed range of channel dimensions [Wan et al., 2020], which hinders its effectiveness and practicality. Thus, in the next section, we introduce FlexCHarts, which is a differentiable channel masking method that allows searching for channel dimensions in a flexible range.

5.3 FlexCHarts

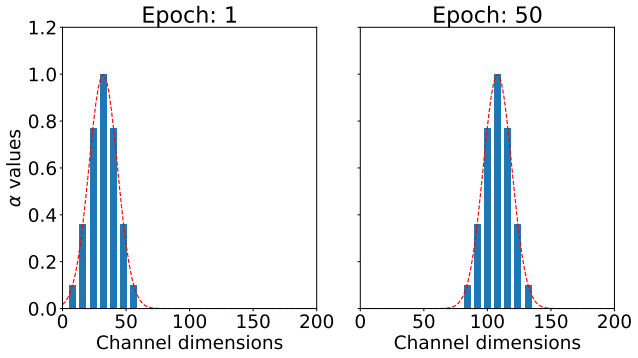
To enable searching for channel dimensions in a flexible range, in this section, we first introduce the flexible channel masking method, which reformulates the α variables in order to permit to change the range of channel dimensions as the search progresses. Then, we elaborate on our dynamic channel allocation mechanism, which modifies the supernet to accommodate the changes in the channel dimension ranges during the search.

5.3.1 Flexible channel masking

Despite its computational efficiency, the vanilla channel masking method can search only within a fixed range of channel dimensions. To overcome this limitation, FlexCHarts reformulates the α variables in such a way that the channel dimension range can be changed on-the-fly during the search while still benefiting from the computational efficiency of the vanilla channel masking method. Instead of defining each α_k as an independent variables



(a) Vanilla channel masking.



(b) FlexCHarts.

Figure 5.3: An example of α values in vanilla channel masking method versus the proposed FlexCHarts methods between the first and last epoch of a search.

as in the vanilla channel masking method, we define α_k as a smooth function of the channel dimension that it corresponds to, where it has the highest value at the center and close to zero at the edges of the range. While various smooth functions would be equally applicable, we used the following exponential function to define α_k in this work due to its simplicity:

$$\alpha_k = \exp(-\frac{1}{2}(\frac{f_k - \mu}{\sigma})^2) \tag{5.3}$$

where μ and σ are the mean and standard deviation of the exponential function. The formulation given above differs from the vanilla channel masking method in the sense that α is no longer a trainable variable. Instead, the proposed flexible channel masking method defines μ as the trainable variable and derives the α values from the exponential function given in Equation 5.3. This formulation smoothly adapts the α values to different channel ranges based on the information on the gradients of μ.

Figure 5.3 shows an example of how α values change in vanilla channel masking and FlexCHarts between the first and last epochs of a search. In vanilla channel masking method, as shown in Fig. Figure 5.3a, α values are initialized randomly. During the search, the α values are updated independently and eventually in the last epoch, the α value that corresponds to the optimal channel dimension becomes significantly higher than the others, setting the channel dimension for the final architecture. In contrast, the α values in FlexCHarts, as shown in Figure 5.3b, are taken from the exponential function given in Equation 5.3 (shown with red dashed line in the figure). The mean value of the exponential function is updated at every step of the search phase, gradually shifting the α values towards the optimal channel dimension.

The proposed reformulation of α variables in Equation 5.3 has two main advantages. First, the given expression is already differentiable; thus, it eliminates the need for tuning additional optimization hyperparameters such as temperature and noise in Gumbel softmax to make the optimization amenable to solution with gradient descent as in the prior work [Wan et al., 2020]. Second, thanks to α_k values that are close to zero at the edges of the range, shifting the range to larger or smaller values gently introduces the new parameters to the kernels, preventing abrupt changes in the loss value that would otherwise be detrimental to the search process. In short, the proposed formulation of α variables is more efficient, flexible and easier to tune than the vanilla channel masking method. Thus, it provides a better solution for searching optimal channel dimensions in DNN architectures.

5.3.2 Dynamic channel allocation

In vanilla channel masking method, the supernet kernels do not require any dimension changes during the search phase as the channel range remains as the same. However, with the reformulation of the α variables in FlexCHarts that permits to change the channel dimension range as the search progresses, the dimensions of the supernet kernel must also be efficiently adapted during the search phase. In this section, we elaborate on how we modify the supernet kernels for dynamic channel dimension range.

Supernet kernels in channel masking methods must have a number of channels equal to or greater than the maximum of the channel dimension range. Therefore, when the range of the channel dimensions shifts to higher values, we need to instantiate a larger supernet kernel. Likewise, when the range of channel dimensions shifts to smaller values, a part of the supernet kernel becomes redundant due to multiplication with an α_k value that is close to zero and is therefore no longer needed. As such, we can reduce the memory footprint and improve the computational efficiency of the search by switching to a smaller supernet kernel. To adjust the dimensions of the supernet kernels based on the changes in the channel dimension range, we introduce the dynamic channel allocation algorithm.

There are two critical design considerations for the dynamic channel allocation algorithm. First, changing the supernet kernels should have minimal impact on the on-going search to prevent loss of progress. Second, allocating new channels should incur only an insignificant

Supernet $\mathcal{N}_{\alpha,W}$ *with parameters* W *and* α*;*
Optimizer *weight optimizer* O_w*, arch optimizer* O_a*;*
Dataset *training:* D_t*, search:* D_s*;*
Initialize;
for *all epoch e* **do**
  for *all steps s* **do**
    Read training batch $b_t \leftarrow D_t$;
    Backpropagate $\mathcal{N}_{\alpha,W}$ with b_t;
    Update $W \leftarrow O_w$;
    Read search batch $b_s \leftarrow D_s$;
    Backpropagate $\mathcal{N}_{\alpha,W}$ with b_s;
    Update $\alpha \leftarrow O_a$;
  end
  Update channel dimensions of $\mathcal{N}_{\alpha,W}$;
end

Algorithm 2: FlexCHarts algorithm for channel dimension search with dynamic channel allocation.

computational overhead. To achieve the first condition, when the proposed dynamic channel allocation algorithm changes the dimensions of a kernel, it transfers the trained weights of the old kernel to the new one where applicable, which preserves the progress made in earlier training steps. For the latter objective, the proposed algorithm shall not react to changes in α_k values in every step. Instead, it waits until the end of an epoch to perform the changes to the supernet kernels to reduce the computational overhead. Altogether, the proposed dynamic channel allocation algorithm enables changing the dimensions of the supernet kernel with minimal impact on the search process and negligible computational overhead.

In more details, the dynamic channel allocation proceeds as described in Algorithm 2. The algorithm takes a supernet $\mathcal{N}_{\alpha,W}$ with weights W and channel parameters α as inputs, as well as the gradient descent optimizers O_w and Oa to update the weights and channel parameters, respectively. It also takes three datasets as inputs: equally sized D_t and D_s to train the weights and channel parameters, respectively. In each step of the algorithm, it reads a batch of samples from D_t, performs a backpropagation on the supernet, and updates the weights. Then, it repeats the same steps for channel parameters by reading a batch from D_s, performing a backpropagation, and updating them. When the same operations are performed for all samples in datasets D_t and D_s, it updates the channel dimensions of the supernet based on the changes made to α values. We repeat the same operations for a predefined number of epochs.

In short, the proposed FlexCHarts algorithm permits to search for optimal channel dimensions in a flexible channel dimension range while automatically managing the changes in the supernet with minimal computational overhead.

5.4 Experiments

We now show the effectiveness of the proposed method through a number of experiments. In this section, we first give details about the search space, datasets, and hyperparameters that we use in the experiments, then we compare the proposed FlexCHarts method against the baseline methods through extensive experiments and discuss the results.

5.4.1 Experimental setup

We perform experiments on a widely used image classification dataset, namely `CIFAR10` [Krizhevsky, 2009] with a preprocessing pipeline for training that consists of a random crop of the input image with a size of 32 and padding of 4, random horizontal flip, normalization, and a cut-out with a length of 16 [DeVries & Taylor, 2017]. For the validation and test phases, we use only a normalization layer in the preprocessing pipeline. We use a batch size of 96 for both the search and training phases. We perform all the experiments on an NVIDIA Tesla V100-SXM2 GPU with a 32GB memory.

For the search phase, we randomly split the training data set into two equally sized subsets to train the weights and channel parameters separately. We use a stochastic gradient descent (SGD) optimizer with a momentum coefficient of 0.9, a weight decay of $3e-4$, a gradient clip of 5 to train the weights. We initialize the learning rate of the SGD optimizer to 0.025 and anneal it every step with a cosine annealing scheduler down to 0 at the end of the last step. We use an Adam optimizer [Kingma & Ba, 2015] with a learning rate of 0.1, running average coefficients of 0.5 and 0.999, and a weight decay of 0 to train the channel parameters. We use a dropout with a probability that starts at 0 and linearly increases to 0.2 until the end of the last step. The search phase takes 50 epochs to complete. After the search phase is completed, we train the DNN architecture with the discovered channel dimensions from scratch for 100 epochs to obtain its final accuracy. We also use an SGD optimizer in the training phase with the same hyperparameters as for the search phase.

As widely adopted by the community [Liu et al., 2019; Wu et al., 2019; Wan et al., 2020], we use a fixed stem and head stages at the beginning and end of our DNN architectures while we are searching for the optimal channel dimensions for the intermediate stages. The stem and head stages consist of convolutional blocks with kernel sizes of 3×3 and 1×1 and channel dimensions of 108 and 256, respectively. The intermediate block consists of 20 stages, each with a microarchitecture identical to the DARTS architecture [Liu et al., 2019]. Each stage may have different channel dimensions as a result of the search phase while all the layers in a stage share the same channel dimension. For the training phase, we also use an auxiliary head that consists of three fully-connected layers with an auxiliary weight of 0.4 as proposed by the prior work [Liu et al., 2019].

Table 5.1: Results of the DMaskingNAS and FlexCHarts methods targeting low and high-resource scenarios. Check and cross marks indicate whether the requirement is satisfied or not.

		Evaluation			Search	
Scenario	Search algorithm	Top-1 acc. (%)	Latency (ms)	FLOPS $\times 10^9$	Search time (GPU-hours)	Search memory (GB)
Low-resource (<0.3ms latency)	DMask-small	95.62	0.366 (✗)	0.258	2.63	10.3
	DMask-large	93.40	0.288 (✓)	0.095	5.75	28.3
	FlexCHarts	94.10	0.287 (✓)	0.093	3.36	16.6
High-resource (>96% accuracy)	DMask-small	95.67 (✗)	0.452	0.433	2.74	12.0
	DMask-large	96.06 (✓)	0.606	0.736	5.78	28.3
	FlexCHarts	96.04 (✓)	0.654	0.773	4.37	19.1

5.4.2 Performance of the differentiable channel search

To evaluate the effectiveness of the proposed FlexCHarts method, we first compare it against the DMaskingNAS method, which has a fixed search space. Because the effectiveness and efficiency of DMaskingNAS method is highly sensitive to their predefined range of channel dimensions, we create two baselines that represent DMaskingNAS methods with small and large range of channel dimensions, which we simply refer to as DMask-small and DMask-large. The details of these search spaces are deferred to Section B.1.

To mimic target inference platforms with different resource constraints, we perform our experiments under low- and high-resource scenarios. For the low-resource scenario, we prioritize the computational requirements of the searched DNN architectures and target an inference latency under 0.3 millisecond per sample. In contrast, for the high-resource scenario, we prioritize the accuracy and aim for DNN architectures that achieve a top-1 test accuracy higher than 96% on `CIFAR10`. We adjust the latency coefficients (i.e., λ in Equation 5.2) to fulfill these accuracy and computational complexity requirements.

Table 5.1 summarizes the results of our experiments with the FlexCHarts, DMask-small, and DMask-large methods under the low- and high-resource scenarios. For the low-resource scenario, while the DMask-small method has the highest top-1 accuracy and lowest search time and memory, it fails to find a DNN architecture that achieves the target of 0.3 millisecond inference latency per batch due to its limited channel range. The DMask-large and FlexCHarts methods succeed to find DNN architectures that achieve the given inference latency target. However, the DMask-large method requires 5.75 GPU-hours and 28.3 GB of memory to find the DNN architecture as it needs to train a larger supernet whereas the FlexCHarts method finds an equivalent architecture only in 3.36 GPU-hours and using 16.6 GB of memory.

The FlexCHarts method also outperforms the DMaskingNAS method in the high-resource scenario. Due to its limited range of channel dimensions, DMask-small fails to find a DNN architecture that is large enough to achieve a top-1 accuracy greater than 96%. In contrast,

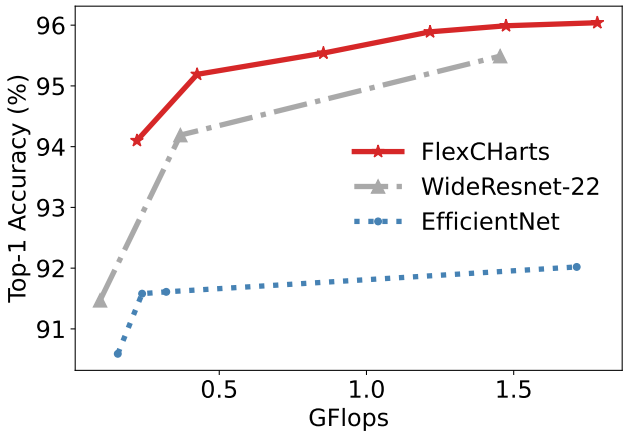


Figure 5.4: Accuracy on `CIFAR10` versus computational complexity in terms of FLOPS for DNN architectures found by FlexCHarts as well as the baseline WideResnet and EfficientNet models.

both DMask-large and FlexCHarts methods are able to find DNN architectures with the target accuracy requirements. Similarly to the low-resource scenario, the DMask-large method requires 5.78 GPU-hours and 28.3 GB of memory to achieve this goal whereas the FlexCHarts method finds an equivalent DNN architecture in 4.37 GPU-hours and using 19.1 GB of memory.

These experiments clearly show that the proposed FlexCHarts method can find the channel dimensions that meet the requirements of varying resource constraints and optimization goals without the restrictions of a fixed search space. Moreover, it does not require to train a redundantly large supernet, thus it searches for the channel dimensions efficiently with lower GPU-hours and memory requirements than the DMaskingNAS methods.

5.4.3 Comparison with other dimension adaptation methods

We now proceed with the experiments that compare the proposed FlexCHarts method against other channel dimension scaling methods. For this purpose, we use the following baselines: WideResnet architectures [Zagoruyko & Komodakis, 2016], which proposes to scale DNN architectures by simply multiplying its channel dimensions by a predetermined coefficient, and EfficientNet architectures [Tan & Le, 2019], which proposes compound scaling, in which the depth and width are scaled uniformly by a coefficient.

In these experiments, we use WideResnet with a depth of 22 and width factors of 1, 2, and 4 and EfficientNet with its four largest variants, namely *B6*, *B7*, *B8*, and *L2*, which requires a similar range of FLOPS with the DNN architectures found by FlexCHarts. To eliminate the performance discrepancies caused by differences in their implementations, we compare their computational complexity in terms of FLOPS. For different search runs with FlexCHarts, we used the latency coefficients (λ) varying between $1e-3$ and $1e-1$. We train all DNN architectures using the same training parameters given in Section 5.4.1.

Table 5.2: Comparison of DMaskingNAS and FlexCHarts for utilization-aware search.

	Util-aware	Util. (%)	Search time (GPU-hours)	Search mem. (GB)	Top-1 acc. (%)	FLOPS $\times 10^9$	Latency (ms)
DMask	No	72.6	7.85	31.3	93.34	0.74	0.99
DMask	Yes	93.6	7.79	31.3	93.05	1.11	1.15
FlexCHarts	No	81.9	5.21	15.2	94.0	0.78	0.93
FlexCHarts	Yes	92.8	5.15	17.8	94.03	0.91	0.96

Figure 5.4 shows the top-1 accuracy and FLOPs requirements of the DNN architectures found by the proposed FlexCHarts method as well as the baseline methods. Because FlexCHarts can automatically search the optimal channel dimensions within a flexible range, the architectures found by FlexCHarts achieve between 0.5-1% and 2-4% better accuracy than WideResnet and EfficientNet for similar FLOPS requirements, respectively. Moreover, the effectiveness of the channel scaling methods used in WideResnet and EfficientNet is highly sensitive to the design of initial DNN architectures, which requires heuristics and manual development efforts. In contrast, the proposed FlexCHarts method finds channel dimensions that achieve better accuracy and/or FLOPS and it does so completely in an automatic fashion.

5.4.4 Channel search for improved resource utilization

We have so far evaluated FlexCHarts on accuracy, latency, and FLOPs metrics. In Chapter 4, we have shown that a utilization-aware neural architecture search finds DNN models that exhibit improved resource utilization at target array-based accelerators, which consequently leads to better accuracy and/or latency. Therefore, we now evaluate FlexCHarts with an objective function that includes a utilization term as in Equation 4.3. We perform an architecture search with FlexCHarts and DMaskingNAS using the cell structure that we found in Chapter 4. For the channel range of DMaskingNAS, we use a channel range similar to what we use in Chapter 4, which is shown in Section B.1. We use the analytical model for systolic arrays that we developed in Chapter 4 to estimate the utilization and latency terms during the search while assuming an array size of 32×32 following our analysis in Chapter 3. We finally use the cycle-accurate systolic array simulator that we introduced in Chapter 3 to obtain the latency and utilization of the DNNs that are found by FlexCHarts and DMaskingNAS.

Table 5.2 summarizes the results of experiments for the utilization-aware search. We observe that both DMaskingNAS and FlexCHarts methods find DNN architectures with similar resource utilizations, namely 93.6% and 92.8%, which is about 10-20% higher than the DNNs found by a utilization-agnostic search. However, the FlexCHarts completes the search in 5.15 GPU-hours using only 17.8 GB of memory, which is 1.5× faster and 40% more memory efficient than the DMaskingNAS method thanks to its flexible search space.

5.5 Conclusion

In this chapter, we addressed the limitations of neural architecture search methods that have fixed search space for channel dimensions. We reformulated the architectural variables in the differentiable channel masking method to enable searching for channel dimensions smoothly and freely without a fixed range. We also introduced a new dynamic channel allocation mechanism that allows changing the dimensions of the supernet efficiently during the search.

Through extensive experiments, we demonstrated that the proposed FlexCHarts framework finds optimal channel dimensions for DNN architectures under various resource constraints and performance objectives without the limitations of the existing methods with fixed search spaces. Moreover, it searches the optimal channel dimensions faster and with reduced memory requirements than the existing methods.

6 Conclusions

6.1 Summary

In this thesis, we addressed the underutilization problem in DNN inference accelerators from both hardware and software perspectives. We first studied the multi-pod systolic architectures for DNN inference workloads. Then, we analyzed the characteristics of popular DNN workloads such as CNNs and Transformer and performed a design space exploration using the effective throughput per Watt metric to find the optimal array dimensions. While the optimal array size that we found is workload-dependent, this design space exploration is important to understand the trade-off between power efficiency and utilization with respect to the array dimensions in systolic architectures. It further demonstrates the gap between the existing accelerators and the optimal design points. Moreover, the proposed design space exploration is open-sourced: it can easily be used to determine the optimal array dimensions for any give set of target DNN workloads and hardware specifications.

We then investigated how we can design a multi-pod systolic architecture using the optimal array dimensions. To that end, we first analyzed various interconnect topologies in order to efficiently connect large numbers of systolic pods. We showed that the expanded Butterfly topology outperforms all other candidate topologies due to its high bisection bandwidth and short round-trip latency. Then, we demonstrated that the existing tiling and scheduling strategies do not exploit the degree of parallelism in DNN workloads to its full extent. Thus, they fail to maintain high utilization for a large number of pods. Therefore, we also introduced a novel tiling strategy that maximizes utilization in multi-pod systolic architectures. Based upon our findings in array granularity, interconnect topology, and tiling strategy, we proposed a novel scale-out systolic array architecture. Through extensive experiments with various DNN workloads and detailed cycle-accurate hardware simulations, we demonstrated that the proposed scale-out systolic array architecture achieves significantly higher resource utilization than the existing architectures, paving the way for the development of faster and more efficient DNN accelerators.

Despite the improvements in accelerator architectures, we observe that DNNs that are not

optimized for the target accelerators still can not achieve high resource utilization. Thus, we then focused on optimizing DNNs for target accelerators using hardware-aware neural architecture search. For this purpose, we developed an analytical model for the resource utilization in array-based DNN accelerators and proposed a smooth approximation that makes it differentiable. We then developed a utilization-aware differentiable neural architecture search framework, which builds DNNs with high resource utilization at target inference accelerators. We demonstrated that the DNNs built with the proposed framework achieve shorter inference latency and/or higher accuracy thanks to improved resource utilization, allowing higher quality inference at reduced cost.

While the proposed framework can successfully find DNNs with high resource utilization, its fixed search space imposes limitations on its efficiency and practicality. Therefore, we then focused on improving the efficiency of the proposed neural architecture search framework and addressing the limitations imposed by the common search space methods. To that end, we reparametrized the architectural parameters of the channel masking method used in differentiable neural architecture search frameworks. We proposed a novel differentiable neural architecture search framework with flexible search space for channel dimensions, which automatically updates the search space on-the-fly based on the progress made during the search. We showed that the proposed framework with the flexible search space can find DNNs equivalent to those that are found by the existing frameworks. However, this is achieved without the need for manually designing a search space while significantly reducing the required GPU-hours. Combined with the utilization-aware neural architecture search, the proposed framework finally permits to obtain DNNs that exhibit high resource utilization at target array-based accelerators at a reduced search cost in a fully automated manner.

In summary, this thesis makes contributions to the fields of DNN accelerators and architectures. The proposed scale-out systolic array architecture and the utilization-aware differentiable neural architecture search framework with the flexible search space offer significant improvements in the resource utilization of the accelerators and the efficiency of neural architecture search. These improvements will allow researchers and engineers to develop more efficient and cost-effective DNN applications that can be deployed on a wider range of platforms.

6.2 Future Directions

In Chapter 3, we targeted a generic DNN accelerator that works well for both computer vision and natural language processing tasks. Thus, we selected a mixture of CNN and Transformer models as benchmark to find the optimal array granularity. However, recent work has shown that Transformer models can outperform their CNN counterparts in vision tasks as well [Dosovitskiy et al., 2021]. As a result, we observe that the usage of Transformers is rapidly increasing in vision tasks, which will encourage researchers to develop DNN accelerators that are more specialized to Transformer models. Although this thesis have already covered the optimal

array dimensions for NLP Transformer models, the three key design pillars that have been discussed in this thesis should be reiterated for Vision Transformers considering their distinct computational characteristics and requirements. Likewise, the utilization-aware neural architecture search framework that is presented in Chapter 4 should be adapted to Transformer models in order to improve the resource utilization of inference accelerators for both Vision and NLP Transformers.

While designing the SOSA, we have assumed that the DNN models fit on the on-chip memory banks, which is a common and valid assumption due to the moderate size of today's CNN and Transformer models. However, in recent years, we have observed a significant increase in the popularity and the size of language models such as OpenAI's GPT-3 [Brown et al., 2020], which do not fit on the on-chip memory of standard inference accelerators. As a result, the efficiency of inference accelerators would drop due to frequent off-chip memory accesses. To mitigate the overhead of off-chip memory accesses for the large language models, future research should focus on three-dimensional integrated circuit (3D-IC) technologies [Li et al., 2021b] to efficiently integrate memory and logic in inference accelerators.

In parallel with the increase in the size of the models, some companies develop wafer-scale accelerators [Lie, 2021], which offer up to two orders of magnitude larger power envelope and silicon area than today's server form-factors. To scale the SOSA design to such large settings, we need to rethink how the systolic pods are connected to each other. In this thesis, we demonstrated that the Butterfly topology is ideal for standard server form-factors. However, the wire lengths in a Butterfly topology grows with a logarithmic complexity with respect to the number of pods and might incur long latencies for large numbers of pods in a wafer-scale accelerator. As a result, the SOSA design with a Butterfly topology might not scale well to such large settings. To overcome the limitations due to the wire lengths in the Butterfly topology, future work should investigate the usage of a *hierarchical* interconnect topology, where a different topology might connect the clusters of systolic pods that are far from each other in the chip layout.

While searching for DNN models using neural architecture search, the task accuracy is naturally one of the optimization objectives besides the resource utilization. However, prior work has shown that adversarial perturbations may easily reduce a DNN's accuracy to zero [Goodfellow et al., 2015; Moosavi-Dezfooli et al., 2017]. Thus, finding DNN architectures with high *clean* accuracy using neural architecture search does not guarantee that these DNNs can be reliably deployed on real-world settings. Moreover, despite the research efforts in some recent work [Guo et al., 2020b; Hosseini et al., 2021], the impact of DNN architectures on adversarial robustness is not yet understood well. Therefore, the neural architecture search proposed in this thesis can be extended to cover the adversarial robustness aspect of DNN architectures in order to find DNNs that not only exhibit high resource utilization and clean accuracy but are also resilient against adversarial perturbations.

A Appendix of Chapter 4

A.1 Micro-architecture search

Table A.1 presents the candidate operations in a cell. We include standard, dilated and depthwise separable (DWS) convolutions along with the identity and zero operations. For simplicity, we only consider ReLU activations.

Table A.1: Microarchitecture search space. DWS: Depthwise Separable.

block name	type	kernel	dilation	nonlinearity
`conv2d_3x3`	Convolution	3	1	ReLU
`conv2d_5x5`	Convolution	5	1	ReLU
`dws_3x3`	DWS Conv.	3	1	ReLU
`dws_5x5`	DWS Conv.	5	1	ReLU
`dil_3x3`	Convolution	3	2	ReLU
`dil_5x5`	Convolution	5	2	ReLU
`identity`	-	-	-	-
`zero`	-	-	-	-

A.2 Utilization and Runtime details

In this section, we analyze the utilization and runtime of all the building blocks. We consider the operations of Table A.1 as well as fully connected layers (for the classifier). Maxpooling layers, batch normalization and activation functions, i.e., ReLUs, are characterized by full utilization and zero runtime, since they need no matrix multiplications.

Let k_1 and k_2 be the kernel sizes, c and f the input and output channels, s_1 and s_2 the systolic array dimensions, h and w the height and width of the input, b the batch size. The number of operations is

Table A.2: Utilizations and runtimes for all building blocks. Symbols explained in text. † includes all other layer types: identity, zero, maxpooling, ReLUs.

Block Type	Runtime	Utilization
Convolution	$\left\lceil \frac{k_1 k_2 c}{s_1} \right\rceil \left\lceil \frac{f}{s_2} \right\rceil hwb$	$\frac{k_1 k_2 cf}{s_1 s_2 \left\lceil \frac{k_1 k_2 c}{s_1} \right\rceil \left\lceil \frac{f}{s_2} \right\rceil}$
Depthwise Convolution	$c \left\lceil \frac{k_1 k_2}{s_1} \right\rceil fhwb$	$\frac{k_1 k_2}{\left\lceil \frac{k_1 k_2}{s_1} \right\rceil f}$
Fully connected	$\left\lceil \frac{c}{s_1} \right\rceil \left\lceil \frac{f}{s_2} \right\rceil b$	$\frac{cf}{s_1 s_2 \left\lceil \frac{c}{s_1} \right\rceil \left\lceil \frac{f}{s_2} \right\rceil}$
†	0	1

$$\texttt{MACs} = hwbk_1 k_2 cf \tag{A.1}$$

The utilization of a specific layer is computed by dividing the number of MACs by the runtime.

Convolution

The runtime and utilization of a convolution are computed in Section 4.1 of the main text:

$$\texttt{RUNTIME}_{\text{conv}} = \left\lceil \frac{k_1 k_2 c}{s_1} \right\rceil \left\lceil \frac{f}{s_2} \right\rceil hwb \tag{A.2}$$

$$\texttt{UTIL}_{\text{conv}} = \frac{k_1 k_2 cf}{s_1 s_2 \left\lceil \frac{k_1 k_2 c}{s_1} \right\rceil \left\lceil \frac{f}{s_2} \right\rceil} \tag{A.3}$$

Depthwise Convolution

A single convolutional filter is applied to each input channel. In this case the number of input and output channels is the same $c = f$. There is no input reuse, meaning that only one column of the systolic array is used. In other words, the $\left\lceil \frac{f}{s_2} \right\rceil$ term in Equation A.2 is replaced by $\left\lceil \frac{1}{s_2} \right\rceil = 1$. Finally, the operation is repeated c times, yielding the following runtime:

$$\texttt{RUNTIME}_{\text{depthwise}} = c \left\lceil \frac{k_1 k_2}{s_1} \right\rceil hwb \tag{A.4}$$

$$\texttt{UTIL}_{\text{depthwise}} = \frac{k_1 k_2}{s_1 s_2 \left\lceil \frac{k_1 k_2}{s_1} \right\rceil} \tag{A.5}$$

The utilization is calculated by dividing the number of multiply-accumulates (MACs) by the runtime. Equation A.5 shows the ineffectiveness of the depthwise convolution, which is inversely proportional to the second dimension of the systolic array.

Depthwise Separable (DWS) Convolution

The depthwise separable convolution is the sequence of a depthwise convolution and a (standard) convolution. Thus, the runtime and utilization are computed via addition of the respective terms.

Fully Connected layers

The runtime and utilization can be derived from the convolution formulae by setting $k_1 = k_2 = 1$ and $h = w = 1$. Concretely, the kernel size can be considered to be 1×1, while the fully connected layer has c inputs and f outputs.

$$\texttt{RUNTIME}_{\text{fc}} = \left\lceil \frac{c}{s_1} \right\rceil \left\lceil \frac{f}{s_2} \right\rceil b \tag{A.6}$$

$$\texttt{UTIL}_{\text{fc}} = \frac{cf}{s_1 s_2 \left\lceil \frac{c}{s_1} \right\rceil \left\lceil \frac{f}{s_2} \right\rceil} \tag{A.7}$$

A.3 Additional experimental results

In this Section, we present additional experiments on `CIFAR10` and `ImageNet100` datasets.

A.3.1 `CIFAR10` dataset

Figure A.1 shows the cells found during the micro-architecture search stage for *all* methods. The methods opt for different configurations. Specifically, the FLOPS model selects mainly depthwise separable convolutions, since they correspond to fewer operations. However, such convolutions result in very increased runtimes and severe mitigation in utilization, as Equation A.4 and Equation A.5 show. The Roofline model operates on the compute-bound region and behaves identically as the FLOPS model. The Blackbox model tries to compensate

Table A.3: Experimental results for `CIFAR10` over 3 random seeds.

	Accuracy (%, ↑)				Runtime (μs, ↓)				HV (↓)
λ	0.1	0.5	1.0	5.0	0.1	0.5	1.0	5.0	
Blackbox	$91.4_{\pm 1.07}$	$90.2_{\pm 0.25}$	$91.3_{\pm 0.66}$	$90.4_{\pm 0.83}$	$209_{\pm 57}$	$155_{\pm 9}$	$147_{\pm 14}$	$122_{\pm 2}$	1.47
Roofline	$91.7_{\pm 0.68}$	$89.2_{\pm 0.85}$	$88.7_{\pm 0.91}$	$87.6_{\pm 4.58}$	$214_{\pm 43}$	$175_{\pm 33}$	$137_{\pm 62}$	$252_{\pm 53}$	1.86
FLOPS	$90.0_{\pm 0.88}$	$88.4_{\pm 1.91}$	$84.0_{\pm 6.39}$	$87.0_{\pm 0.99}$	$235_{\pm 26}$	$320_{\pm 37}$	$251_{\pm 55}$	$159_{\pm 33}$	2.68
U-Boost	$90.9_{\pm 0.88}$	$91.4_{\pm 0.90}$	$91.3_{\pm 0.24}$	$89.5_{\pm 1.14}$	$73_{\pm 8}$	$51_{\pm 10}$	$39_{\pm 9}$	$30_{\pm 0}$	0.386

(in terms of utilization) by omitting convolutions, including depthwise separable convolutions. This suggests that it is able to understand that DWS are antithetical to the utilization objective and opts for operations with no utilization overhead, such as the identity and zero gates.

Table A.3 presents the experimental results for `CIFAR10` in more detail. The proposed method achieves significantly lower runtimes for all λ values outperforming the baselines in a range of $\sim 2.8-5\times$. It is also worth mentioning that the FLOPS and Roofline models do not exhibit decreasing runtimes as λ increases. They are also characterized by high variance in the runtime measurements, indicating an unsophisticated search. This drawback can be attributed to the loss function for the utilization term which does not take into account the number of channels. The blackbox model and our proposed method have lower standard deviations and a monotonically decreasing runtime. Finally, our proposed method has better quality of exploration for the tradeoff of accuracy and runtime, as the Hypervolume metric indicates.

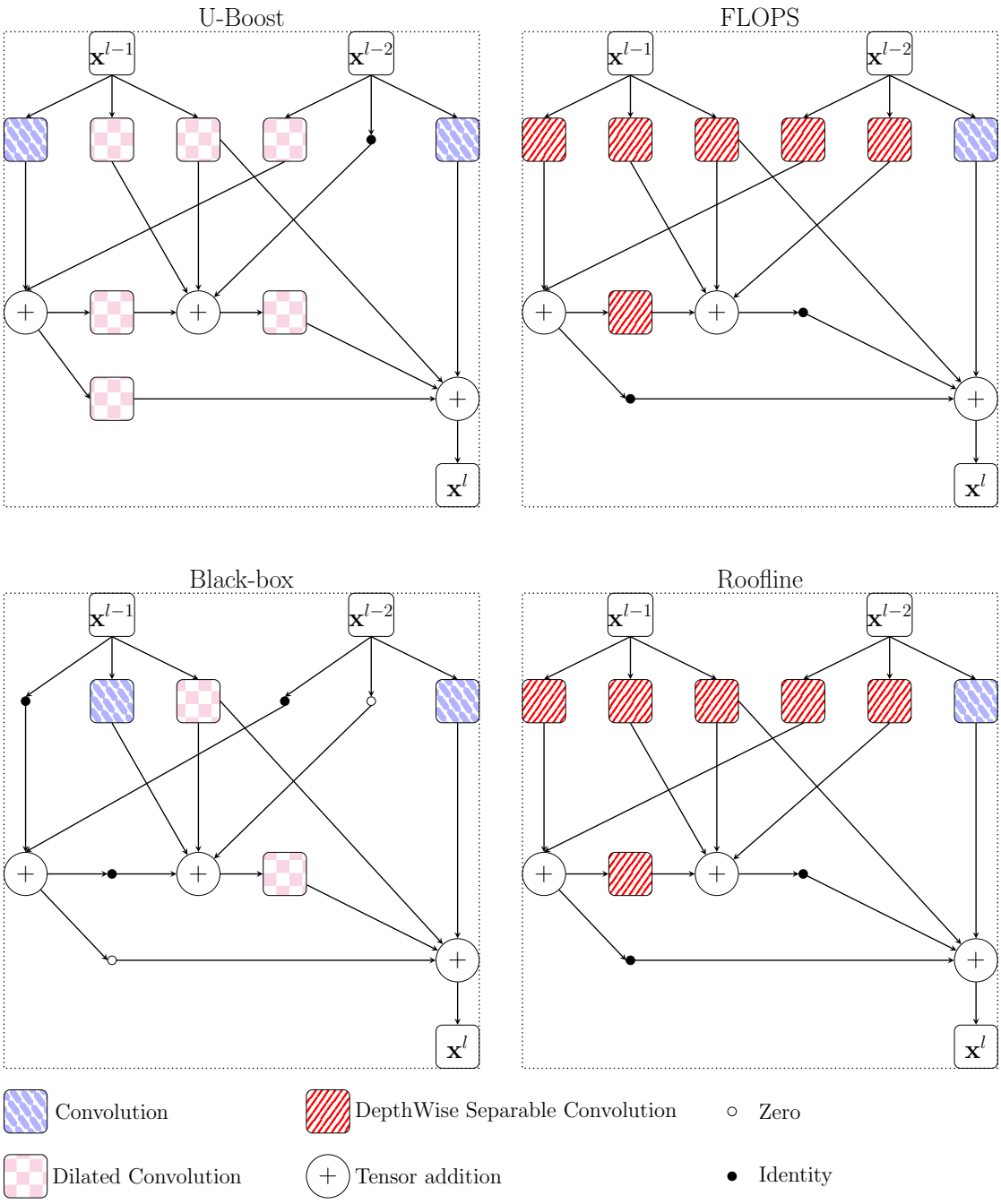


Figure A.1: Cell architectures found for $\lambda = 0.1$ on the `CIFAR10` dataset.

A.4 Hyperparameters

The complete list of hyperparameters is presented in Table A.4.

Table A.4: Experiment Hyperparameters. – indicates that the ImageNet100 experiment uses the same settings as the CIFAR10 experiment. †: the architecture for ImageNet100 is produced by search on CIFAR10. MS: micro-architecture search, CS: channel search, FT: final training.

	CIFAR10	ImageNet100
ms_no_epoch	10	†
cs_no_epoch	30	†
ft_no_epoch	100	70
array_size	[128, 128]	–
start_arch_train	0	–
weight_vs_arch	0.8	–
search_sgd_init_lr	0.05	–
search_sgd_momentum	0.9	–
search_sgd_weight_decay	3e-4	–
search_weight_grad_clip	0.5	–
adam_init_lr	0.1	–
adam_weight_decay	0	–
init_tau	1.0	–
tau_anneal_rate	0.95	–
min_tau	0.001	–
search_batch_size	64	–
train_batch_size	256	–
train_sgd_init_lr	0.1	–
train_sgd_momentum	0.9	–
train_sgd_weight_decay	5e-4	–
train_weight_grad_clip	0.5	–

B Appendix of Chapter 5

B.1 Channel ranges of DMask baselines

Table B.1: Channel ranges of DMask-small and Dmask-large baselines for the experiments in Section 5.4.2 and DMask-systolic for the experiments in Section 5.4.4.

	DMask-small			DMask-large			DMask-systolic		
Cell id	start	end	step	start	end	step	start	end	step
0	24	32	8	16	160	16	16	200	8
1	24	32	8	16	160	16	16	200	8
2	24	32	8	16	160	16	16	200	8
3	24	32	8	16	160	16	16	200	8
4	24	32	8	16	160	16	16	200	8
5	24	32	8	16	160	16	16	200	8
6	48	64	8	16	160	16	16	200	8
7	48	64	8	16	160	16	16	200	8
8	48	64	8	16	160	16	16	200	8
9	48	64	8	16	160	16	16	200	8
10	48	64	8	16	160	16	16	200	8
11	48	64	8	16	160	16	16	200	8
12	48	64	8	16	160	16	16	200	8
13	96	160	16	16	160	16	16	200	8
14	96	160	16	16	160	16	16	200	8
15	96	160	16	16	160	16	16	200	8
16	96	160	16	16	160	16	16	200	8
17	96	160	16	16	160	16	16	200	8
18	96	160	16	16	160	16	16	200	8
19	96	160	16	16	160	16	16	200	8

Bibliography

Alwani, M., Chen, H., Ferdman, M., and Milder, P. Fused-layer CNN accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture MICRO*. IEEE Press, 2016.

Ashok, A., Rhinehart, N., Beainy, F., and Kitani, K. M. N2N learning: Network to network compression via policy gradient reinforcement learning. In *6th International Conference on Learning Representations, ICLR, Conference Track Proceedings*, 2018.

Baek, E., Kwon, D., and Kim, J. A multi-neural network acceleration architecture. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA*, pp. 940–953. IEEE, 2020.

Bender, G., Kindermans, P., Zoph, B., Vasudevan, V., and Le, Q. V. Understanding and simplifying one-shot architecture search. In *Proceedings of the 35th International Conference on Machine Learning, ICML*, volume 80 of *Proceedings of Machine Learning Research*, pp. 549–558. PMLR, 2018.

Beneš, V. E. Optimal rearrangeable multistage connecting networks. *The Bell System Technical Journal*, 43(4):1641–1656, 1964.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems*, pp. 2546–2554, 2011.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2020.

Cai, H., Chen, T., Zhang, W., Yu, Y., and Wang, J. Efficient architecture search by network transformation. In McIlraith, S. A. and Weinberger, K. Q. (eds.), *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, pp. 2787–2794. AAAI Press, 2018.

Cai, H., Zhu, L., and Han, S. ProxylessNAS: direct neural architecture search on target task and hardware. In *7th International Conference on Learning Representations, ICLR*, 2019.

Chang, J., Zhang, X., Guo, Y., Meng, G., Xiang, S., and Pan, C. DATA: differentiable architecture approximation. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, NeurIPS*, pp. 874–884, 2019.

Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pp. 269–284, New York, NY, USA, 2014a. Association for Computing Machinery.

Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., and Temam, O. DaDianNao: A machine-learning supercomputer. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, pp. 609–622. IEEE Computer Society, 2014b.

Chen, Y., Emer, J. S., and Sze, V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA*, pp. 367–379, 2016.

Chen, Y., Krishna, T., Emer, J. S., and Sze, V. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE J. Solid State Circuits*, 52(1): 127–138, 2017.

Chen, Y., Yang, T., Emer, J. S., and Sze, V. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE J. Emerg. Sel. Topics Circuits Syst.*, 9(2):292–308, 2019.

Cherlopalle, D., Rengan, X., Han, F., and Ta, Q. INFERENCE using the NVIDIA T4. https://www.dell.com/support/kbdoc/en-us/000143470/inference-using-the-nvidia-t4, 2019. Accessed: 2022-12-23.

Cho, H. Risa: A reinforced systolic array for depthwise convolutions and embedded tensor reshaping. *ACM Trans. Embed. Comput. Syst.*, 20(5s):53:1–53:20, 2021.

Cho, M., Soltani, M., and Hegde, C. One-shot neural architecture search via compressive sensing. *arXiv preprint*, 2019.

Choi, K., Hong, D., Yoon, H., Yu, J., Kim, Y., and Lee, J. DANCE: differentiable accelerator/network co-exploration. In *58th ACM/IEEE Design Automation Conference, DAC*, pp. 337–342. IEEE, 2021.

Choi, Y. and Rhu, M. PREMA: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *IEEE International Symposium on High Performance Computer Architecture, HPCA*, pp. 220–233. IEEE, 2020.

Chollet, F. Keras. https://keras.io, 2015.

Choquette, J., Gandhi, W., Giroux, O., Stam, N., and Krashinsky, R. NVIDIA A100 tensor core GPU: performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

Ci, Y., Lin, C., Sun, M., Chen, B., Zhang, H., and Ouyang, W. Evolving search space for neural architecture search. In *IEEE/CVF International Conference on Computer Vision, ICCV*, pp. 6639–6649. IEEE, 2021.

Dai, X., Zhang, P., Wu, B., Yin, H., Sun, F., Wang, Y., Dukhan, M., Hu, Y., Wu, Y., Jia, Y., Vajda, P., Uyttendaele, M., and Jha, N. K. Chamnet: Towards efficient network design through platform-aware model adaptation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 11398–11407. Computer Vision Foundation / IEEE, 2019.

Deng, J., Dong, W., Socher, R., Li, L., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 248–255. IEEE Computer Society, 2009.

Désidéri, J.-A. Multiple-gradient descent algorithm (MGDA) for multiobjective optimization. *Comptes Rendus Mathematique*, 350:313–318, 2012.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, pp. 4171–4186, 2019.

DeVries, T. and Taylor, G. W. Improved regularization of convolutional neural networks with cutout. *arXiv preprint*, 2017.

Dong, X. and Yang, Y. Searching for a robust neural architecture in four GPU hours. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 1761–1770. Computer Vision Foundation / IEEE, 2019a.

Dong, X. and Yang, Y. Network pruning via transformable architecture search. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, NeurIPS*, pp. 759–770, 2019b.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations, ICLR*, 2021.

Drumond, M., Coulon, L., Zarandi, A. P., Yüzügüler, A. C., Falsafi, B., and Jaggi, M. Equinox: Training (for free) on a custom inference accelerator. In *MICRO: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 421–433. ACM, 2021.

Drumond, M. P. Coltrain: Co-located dnn training and inference. EPFL, 2020.

Farabet, C., Martini, B., Corda, B., Akselrod, P., Culurciello, E., and LeCun, Y. Neuflow: A runtime reconfigurable dataflow processor for vision. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR Workshops*, pp. 109–116, 2011.

Fowers, J., Ovtcharov, K., Papamichael, M., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., Adams, L., Ghandi, M., Heil, S., Patel, P., Sapek, A., Weisz, G., Woods, L., Lanka, S., Reinhardt, S. K., Caulfield, A. M., Chung, E. S., and Burger, D. A configurable cloud-scale DNN processor for real-time AI. In *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA*, pp. 1–14, 2018.

Gao, M., Yang, X., Pu, J., Horowitz, M., and Kozyrakis, C. TANGRAM: optimized coarse-grained dataflow for scalable NN accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pp. 807–820, 2019.

Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. In *3rd International Conference on Learning Representations, ICLR, Conference Track Proceedings*, 2015.

Google. Cloud TPU. https://cloud.google.com/tpu, 2017. Accessed: 2018-01-31.

Google. BERT. https://github.com/google-research/bert, 2020. Accessed: 2020-10-5.

Gordon, A., Eban, E., Nachum, O., Chen, B., Wu, H., Yang, T., and Choi, E. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 1586–1595. Computer Vision Foundation / IEEE Computer Society, 2018.

Guo, C., Zhou, Y., Leng, J., Zhu, Y., Du, Z., Chen, Q., Li, C., Yao, B., and Guo, M. Balancing efficiency and flexibility for DNN acceleration via temporal GPU-systolic array integration. In *57th ACM/IEEE Design Automation Conference, DAC*, pp. 1–6. IEEE, 2020a.

Guo, M., Yang, Y., Xu, R., Liu, Z., and Lin, D. When NAS meets robustness: In search of robust architectures against adversarial attacks. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 628–637. Computer Vision Foundation / IEEE, 2020b.

Gupta, S. and Akin, B. Accelerator-aware neural network design using AutoML. *arXiv preprint*, 2020.

Hazan, E., Klivans, A. R., and Yuan, Y. Hyperparameter optimization: a spectral approach. In *6th International Conference on Learning Representations, ICLR, Conference Track Proceedings*, 2018.

Hazelwood, K. M., Bird, S., Brooks, D. M., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A., Law, J., Lee, K., Lu, J., Noordhuis, P., Smelyanskiy, M., Xiong, L., and

Wang, X. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture, HPCA*, pp. 620–629. IEEE Computer Society, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 770–778, 2016.

He, Y., Zhang, X., and Sun, J. Channel pruning for accelerating very deep neural networks. In *IEEE International Conference on Computer Vision, ICCV*, pp. 1398–1406. IEEE Computer Society, 2017.

He, Y., Lin, J., Liu, Z., Wang, H., Li, L., and Han, S. AMC: automl for model compression and acceleration on mobile devices. In Ferrari, V., Hebert, M., Sminchisescu, C., and Weiss, Y. (eds.), *ECCV - 15th European Conference on Computer Vision, Proceedings, Part VII*, volume 11211 of *Lecture Notes in Computer Science*, pp. 815–832. Springer, 2018.

Hock, A. Introducing Cerebras Systems . https://cerebras.net/introducing-cerebras-systems/, 2019. Accessed: 2020-04-08.

Hosseini, R., Yang, X., and Xie, P. DSRNA: differentiable search of robust neural architectures. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 6196–6205. Computer Vision Foundation / IEEE, 2021.

Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 2261–2269. IEEE Computer Society, 2017.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Bach, F. R. and Blei, D. M. (eds.), *Proceedings of the 32nd International Conference on Machine Learning, ICML*, volume 37 of *JMLR Workshop and Conference Proceedings*, pp. 448–456, 2015.

Jang, E., Gu, S., and Poole, B. Categorical reparameterization with gumbel-softmax. In *5th International Conference on Learning Representations, ICLR, Conference Track Proceedings*, 2017.

Jha, N. K., Ravishankar, S., Mittal, S., Kaushik, A., Mandal, D., and Chandra, M. DRACO: co-optimizing hardware utilization, and performance of DNNs on systolic accelerator. In *IEEE Computer Society Annual Symposium on VLSI, ISVLSI*, pp. 574–579. IEEE, 2020.

Jordà, M., Valero-Lara, P., and Peña, A. J. Performance evaluation of cuDNN convolution algorithms on NVIDIA Volta GPUs. *IEEE Access*, 7:70461–70473, 2019.

Jouppi, N. P., Young, C., Patil, N., Patterson, D. A., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., 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., Killebrew, D., 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., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA*, pp. 1–12, 2017.

Jouppi, N. P., Yoon, D. H., Kurian, G., Li, S., Patil, N., Laudon, J., Young, C., and Patterson, D. A. A domain-specific supercomputer for training deep neural networks. *Commun. ACM*, 63(7): 67–78, 2020.

Jouppi, N. P., Hyun Yoon, D., Ashcraft, M., Gottscho, M., Jablin, T. B., Kurian, G., Laudon, J., Li, S., Ma, P., Ma, X., Norrie, T., Patil, N., Prasad, S., Young, C., Zhou, Z., and Patterson, D. Ten lessons from three generations shaped google's tpuv4i : Industrial product. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, 2021.

Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D. T., Jammalamadaka, N., Huang, J., Yuen, H., et al. A study of BFLOAT16 for deep learning training. *arXiv preprint*, 2019.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR*, 2015.

Krizhevsky, A. Learning multiple layers of features from tiny images. pp. 32–33, 2009.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012*, pp. 1106–1114, 2012.

Kung, H. T. Why systolic architectures? *IEEE Computer*, 15(1):37–46, 1982.

Kung, H. T., McDanel, B., Zhang, S. Q., Dong, X., and Chen, C. Maestro: A memory-on-logic architecture for coordinated parallel use of many systolic arrays. In *30th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP*, pp. 42–50, 2019.

Li, S., Chen, K., Ahn, J. H., Brockman, J. B., and Jouppi, N. P. CACTI-P: architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *IEEE/ACM International Conference on Computer-Aided Design, ICCAD*, pp. 694–701, 2011.

Li, S., Tan, M., Pang, R., Li, A., Cheng, L., Le, Q. V., and Jouppi, N. P. Searching for fast model families on datacenter accelerators. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 8085–8095. Computer Vision Foundation / IEEE, 2021a.

Li, Y., Tang, J., Gao, B., Yao, J., Xi, Y., Li, Y., Li, T., Zhou, Y., Liu, Z., Zhang, Q., Qiu, S., Li, Q., Qian, H., and Wu, H. Monolithic 3D integration of logic, memory and computing-in-memory for one-shot learning. In *2021 IEEE International Electron Devices Meeting (IEDM)*, pp. 21.5.1–21.5.4, 2021b.

Liao, H., Tu, J., Xia, J., and Zhou, X. Davinci: A scalable architecture for neural network computing. In *IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–44. IEEE, 2019.

Lie, S. Multi-million core, multi-wafer AI cluster. In *IEEE Hot Chips 33 Symposium, HCS*, pp. 1–41. IEEE, 2021.

Liew, S. C. and Lee, T. T. *Principles of Broadband Switching and Networking*, volume 32. John Wiley & Sons, 2010.

Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L., Fei-Fei, L., Yuille, A. L., Huang, J., and Murphy, K. Progressive neural architecture search. In *ECCV - 15th European Conference on Computer Vision, Proceedings, Part I*, volume 11205 of *Lecture Notes in Computer Science*, pp. 19–35. Springer, 2018.

Liu, H., Simonyan, K., and Yang, Y. DARTS: differentiable architecture search. In *7th International Conference on Learning Representations, ICLR*, 2019.

Liu, Z. G., Whatmough, P. N., and Mattina, M. Systolic tensor array: An efficient structured-sparse GEMM accelerator for mobile CNN inference. *IEEE Comput. Archit. Lett.*, 19(1): 34–37, 2020a.

Liu, Z.-G., Whatmough, P. N., and Mattina, M. Sparse systolic tensor array for efficient CNN hardware acceleration. *arXiv preprint*, 2020b.

Lu, W., Yan, G., Li, J., Gong, S., Han, Y., and Li, X. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *IEEE International Symposium on High Performance Computer Architecture, HPCA*, pp. 553–564, 2017.

Luo, J., Wu, J., and Lin, W. Thinet: A filter level pruning method for deep neural network compression. In *IEEE International Conference on Computer Vision, ICCV*, pp. 5068–5076. IEEE Computer Society, 2017.

Marchisio, A., Massa, A., Mrazek, V., Bussolino, B., Martina, M., and Shafique, M. Nascaps: A framework for neural architecture search to optimize the accuracy and hardware efficiency of convolutional capsule networks. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD*, pp. 114:1–114:9. IEEE, 2020.

Moosavi-Dezfooli, S., Fawzi, A., Fawzi, O., and Frossard, P. Universal adversarial perturbations. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 86–94. IEEE Computer Society, 2017.

Nayman, N., Noy, A., Ridnik, T., Friedman, I., Jin, R., and Zelnik-Manor, L. XNAS: neural architecture search with expert advice. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, NeurIPS*, pp. 1975–1985, 2019.

Nicol, C. A dataflow processing chip for training deep neural networks. In *Proceedings of the IEEE Hot Chips 29 Symposium*, 2017.

Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., Huang, R., Hock, J. O. G., Liew, Y. T., Srivatsan, K., Moss, D. J. M., Subhaschandra, S., and Boudoukh, G. Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA*, pp. 5–14. ACM, 2017.

NVIDIA. NVDLA Primer. http://nvdla.org/primer.html, 2018. Accessed: 2022-12-23.

NVIDIA. NVIDIA A100 40GB PCIe GPU accelerator product brief, 2020.

Ouyang, J., Noh, M., Wang, Y., Qi, W., Ma, Y., Gu, C., Kim, S., Hong, K., Bae, W., Zhao, Z., Wang, J., Wu, P., Gong, X., Shi, J., Zhu, H., and Du, X. Baidu kunlun an AI processor for diversified workloads. In *IEEE Hot Chips 32 Symposium, HCS*, pp. 1–18. IEEE, 2020.

Patterson, D., Gonzalez, J., Le, Q., Liang, C., Munguia, L.-M., Rothchild, D., So, D., Texier, M., and Dean, J. Carbon emissions and large neural network training. *arXiv preprint*, 2021.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. Efficient neural architecture search via parameter sharing. In *Proceedings of the 35th International Conference on Machine Learning, ICML*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4092–4101. PMLR, 2018.

Qin, E., Samajdar, A., Kwon, H., Nadella, V., Srinivasan, S., Das, D., Kaul, B., and Krishna, T. SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In *IEEE International Symposium on High Performance Computer Architecture, HPCA*, pp. 58–70. IEEE, 2020.

Raihan, M. A., Goli, N., and Aamodt, T. M. Modeling deep learning accelerator enabled GPUs. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, pp. 79–92. IEEE, 2019.

Rapin, J. and Teytaud, O. Nevergrad - A gradient-free optimization platform. https://GitHub.com/FacebookResearch/Nevergrad, 2018.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. In *The Thirty-Third AAAI Conference on Artificial Intelligence*, pp. 4780–4789. AAAI Press, 2019.

Ross, J. Prefetching weights for use in a neural network processor, 2017. US 2017/0103314 A1.

Samajdar, A., Joseph, J. M., Zhu, Y., Whatmough, P. N., Mattina, M., and Krishna, T. A systematic methodology for characterizing scalability of DNN accelerators using scale-sim. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, pp. 58–68. IEEE, 2020.

Sandler, M., Howard, A. G., Zhu, M., Zhmoginov, A., and Chen, L. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 4510–4520. Computer Vision Foundation / IEEE Computer Society, 2018.

Shao, Y. S., Clemons, J., Venkatesan, R., Zimmer, B., Fojtik, M., Jiang, N., Keller, B., Klinefelter, A., Pinckney, N. R., Raina, P., Tell, S. G., Zhang, Y., Dally, W. J., Emer, J. S., Gray, C. T., Khailany, B., and Keckler, S. W. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *MICRO: The 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 14–27, 2019.

Stamoulis, D., Ding, R., Wang, D., Lymberopoulos, D., Priyantha, B., Liu, J., and Marculescu, D. Single-path NAS: designing hardware-efficient convnets in less than 4 hours. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD, Proceedings, Part II*, volume 11907 of *Lecture Notes in Computer Science*, pp. 481–497. Springer, 2019.

Stevens, J. R., Ranjan, A., Das, D., Kaul, B., and Raghunathan, A. Manna: An accelerator for memory-augmented neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, pp. 794–806. ACM, 2019.

Sze, V., Chen, Y., Yang, T., and Emer, J. S. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 2818–2826, 2016.

Tan, M. and Le, Q. V. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML*, volume 97 of *Proceedings of Machine Learning Research*, pp. 6105–6114. PMLR, 2019.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 2820–2828. Computer Vision Foundation / IEEE, 2019.

Tencent. Turbo Transformers. https://github.com/Tencent/TurboTransformers, 2020. Accessed: 2020-09-26.

Wan, A., Dai, X., Zhang, P., He, Z., Tian, Y., Xie, S., Wu, B., Yu, M., Xu, T., Chen, K., Vajda, P., and Gonzalez, J. E. Fbnetv2: Differentiable neural architecture search for spatial and channel

dimensions. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 12962–12971. Computer Vision Foundation / IEEE, 2020.

Williams, S., Waterman, A., and Patterson, D. A. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., Tian, Y., Vajda, P., Jia, Y., and Keutzer, K. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 10734–10742. Computer Vision Foundation / IEEE, 2019.

Xie, S., Zheng, H., Liu, C., and Lin, L. SNAS: stochastic neural architecture search. In *7th International Conference on Learning Representations, ICLR*, 2019.

Xiong, Y., Liu, H., Gupta, S., Akin, B., Bender, G., Wang, Y., Kindermans, P., Tan, M., Singh, V., and Chen, B. Mobiledets: Searching for object detection architectures for mobile accelerators. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 3825–3834. Computer Vision Foundation / IEEE, 2021.

Xu, Y., Xie, L., Zhang, X., Chen, X., Qi, G., Tian, Q., and Xiong, H. PC-DARTS: partial channel connections for memory-efficient architecture search. In *8th International Conference on Learning Representations, ICLR*, 2020.

Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Cong, J. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2017.

Yang, T., Howard, A. G., Chen, B., Zhang, X., Go, A., Sandler, M., Sze, V., and Adam, H. Netadapt: Platform-aware neural network adaptation for mobile applications. In *ECCV - 15th European Conference on Computer Vision, Proceedings, Part X*, volume 11214 of *Lecture Notes in Computer Science*, pp. 289–304. Springer, 2018.

Yu, J. and Huang, T. Autoslim: Towards one-shot architecture search for channel numbers. *arXiv preprint*, 2019.

Yüzügüler, A. C., Dimitriadis, N., and Frossard, P. U-Boost NAS: utilization-boosted differentiable neural architecture search. In *ECCV - 17th European Conference on Computer Vision, Proceedings, Part XII*, volume 13672 of *Lecture Notes in Computer Science*, pp. 173–190. Springer, 2022.

Zagoruyko, S. and Komodakis, N. Wide residual networks. In Wilson, R. C., Hancock, E. R., and Smith, W. A. P. (eds.), *Proceedings of the British Machine Vision Conference 2016, BMVC*. BMVA Press, 2016.

Zhang, X., Wang, J., Zhu, C., Lin, Y., Xiong, J., Hwu, W. W., and Chen, D. DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs. In Bahar, I. (ed.), *Proceedings of the International Conference on Computer-Aided Design, ICCAD*, pp. 56. ACM, 2018.

Zhu, H., Wang, Y., and Shi, C. R. Tanji: a general-purpose neural network accelerator with unified crossbar architecture. *IEEE Des. Test*, 37(1):56–63, 2020.

Zitzler, E. and Thiele, L. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Trans. Evol. Comput.*, 3(4):257–271, 1999.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR*, 2017.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pp. 8697–8710. Computer Vision Foundation / IEEE Computer Society, 2018.

Ahmet Caner Yüzügüler

Contact Info

Address Avenue du 24-Janvier, 28,
1020, Renens, VD, Switzerland

Mobile +41 (0) 78 769 96 41

E-mail ahmet.yuzuguler@epfl.ch

Website acyuzuguler.github.io

Education

Sept.,2018 – Dec.,2022 **Ph.D., Computer Science**, *École Polytechnique Fédérale de Lausanne*, Lausanne, Switzerland, *Advisor: Prof. Pascal Frossard*

Sept.,2015 – March,2018 **M.Sc., Computer Science**, *École Polytechnique Fédérale de Lausanne*, Lausanne, Switzerland, *GPA – 5.52 / 6.0*

2009–2014 **B.Sc., Electrical and Electronics Engineering**, *Middle East Technical University*, Ankara, Turkey, *GPA – 3.73 / 4.0*

Honors

2018 EPFL PhD Fellowship

2015 EPFL Master Research Scholarship

2014 High Honor Roll at B.Sc. degree

Experience

Sept.,2018 – Dec.,2022 **Ph.D. Candidate**,
EPFL

- Developed a neural architecture search framework that optimizes deep neural networks for target hardware accelerators.
- Developed a scale-out systolic array architecture that increases hardware utilization and power efficiency in DNN accelerators.
- Developed a novel analog circuit for deep neural networks.

Feb.,2017 – March,2018 **Research Intern**,
ABB Corporate Research, Automation System Architecture Department

- Implemented a real-time software for power grid simulations in C++ for multi-core processors and GPUs.
- Master's thesis on the subject of approximating power grid models with deep neural networks for real-time simulations.

Sept.,2015 – August,2016 **Research Assistant**,
EPFL, Rigorous System Design Laboratory

- Algorithm design and software implementation of a medical imaging platform on an FPGA and a many-core Kalray MPPA-256 processor.

July,2014 – August,2015 **Hardware Design Engineer**,
Aselsan Inc., Division of Defense Systems Technologies

- PCB and FPGA design of embedded computers for defense applications.

Skills

Competence

Deep learning, Hardware accelerators, FPGA/ASIC design, Computer architecture, Embedded systems

Programming Languages

C/C++, Python, VHDL, Verilog, CUDA, Tensorflow, Pytorch

Design Tools

Synopsys Design Compiler, Xilinx ISE/Vivado Design Tools, Altera Quartus II, Modelsim, Matlab, Simulink

Languages

Turkish **Native**

English **Advanced**, *TOEFL iBT: 107/120*

French **Beginner**, *CEFR Level: A2/B1*

Publications

2022 **A.C.Yüzügüler**, N.Dimitriadis, P.Frossard, *U-Boost NAS: Utilization-Boosted Differentiable Neural Architecture Search*, European Conference on Computer Vision (ECCV).

2022 **A.C.Yüzügüler**, C.Sönmez, M.Drumond, Y.Oh, B.Falsafi, P.Frossard, *Scale-out Systolic Arrays*, ACM Transactions on Architecture and Code Optimization

2021 M.Drumond, L.Coulon, A.Pourhabibi, **A.C.Yüzügüler**, B.Falsafi, M.Jaggi, *Equinox: Training (for Free) on a Custom Inference Accelerator*, MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture

2019 **A.C.Yüzügüler**, F.Celik, M.Drumond, P.Frossard, B.Falsafi, *Analog Neural Networks with Deep-submicrometer Nonlinear Synapses*, IEEE Micro Special Issue on Machine Learning Acceleration

2018 **A.C.Yüzügüler**, A.Moga, C.Franke, *Towards Commoditizing Simulations of System Models Using Recurrent Neural Networks*, IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)

2017 F. Angiolini, A. Ibrahim, W.Simon, **A.C.Yüzügüler**, M. Arditi, J.-P Thiran and G. De Micheli, *1024-Channel 3D ultrasound digital beamformer in a single 5W FPGA*, Design, Automation & Test in Europe Conference & Exhibition (DATE)

2016 **A.C.Yüzügüler**, W.Simon, A. Ibrahim, F. Angiolini, M. Arditi, J.-P Thiran and G. De Micheli, *(Demo) Single-FPGA 3D Ultrasound Beamformer*, International Conference on Field-Programmable Logic and Applications (FPL)

2014 **A.C.Yüzügüler** and E. Şahin, *Changing Utilization Rates in Real Time to Investigate FPGA Power Behavior*, Xilinx Xcell Journal Issue 89 pg.60

2014 **A.C.Yüzügüler**, E. Vural and P. Frossard, *Transformation-invariant Dictionary Learning for Fast Image Classification*, IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)

1015, Lausanne, Switzerland
+41 (0) 78 769 96 41 • ahmet.yuzuguler@epfl.ch
acyuzuguler.github.io