Ph.D. Dissertation

Hardware and Software Techniques for Improving NPU Resource Utilization

Young H. Oh

Department of Electrical and Computer Engineering

The Graduate School

Sungkyunkwan University

Ph.D. Dissertation

Hardware and Software Techniques for Improving NPU Resource Utilization

Young H. Oh

Department of Electrical and Computer Engineering

The Graduate School

Sungkyunkwan University

Hardware and Software Techniques for Improving NPU Resource Utilization

Young H. Oh

A Ph.D. Dissertation Submitted to the Department of Electrical and Computer Engineering and the Graduate School of Sungkyunkwan University in partial fulfillment of the requirements for the degree of Ph.D. in Engineering

October 2021

Supervised by

Jae W. Lee and Joonwon Lee

Major Advisor

This certifies that the Ph.D. Dissertation
of Young H. Oh is approved.

Committee Chair:

Committe Member 1:

Committe Member 2:

Committe Member 3:

Major Advisor:

Major Advisor:

The Graduate School

Sungkyunkwan University

December 2021

.

Contents

List of Tables

List of Figures

Abstract

Hardware and Software Techniques for Improving NPU Resource Utilization

With the proliferation of AI-based applications and services, there are strong demands for the efficient processing of deep neural networks (DNNs). DNNs are known to be both compute- and memory-intensive as they require a tremendous amount of computation and large memory space. To efficiently service DNN applications, many service providers often employ specialized hardware accelerators, called Neural Processing Units (NPUs). However, there are a myriad of NPUs featuring different number representations and compute-to-memory bandwidth ratios. Hence, for a specific NPU running various DNNs, finding an efficient number representation and its execution schedule is very challenging.

We identified that the current data quantization and resource scheduling frameworks for NPUs have the following critical limitations. First, when searching for an efficient data format for a DNN, existing quantization techniques significantly restrict exploration space due to a combinatorial explosion of feasible number representations with varying bit widths. Moreover, they often target a specific DNN framework and/or hardware platform, lacking portability across various execution environments. Second, existing NPU resource schedulers cannot maximize both compute TOP/s and DRAM bandwidth, leading to low resource utilization and throughput. This is because

there is an intrinsic mismatch between the compute-to-memory bandwidth ratio of NPUs and the arithmetic intensity of DNNs. What makes the problem more difficult is that datacenters typically impose QoS constraints (i.e., deadlines) on service latency.

To address such problems, we propose three software optimization techniques for NPUs. The first one is *libnumber*, a portable, automatic quantization framework that efficiently explores an ever-wider range of possible number representations. While *libnumber* effectively encapsulates the internal representation of a number, it systematically finds a compact number representation (type, bit width, and bias) to minimize the user-supplied objective function. The second is Layerweaver, a layer-wise time-multiplexing scheduler for NPUs. Layerweaver reduces the temporal waste of computation resources by interweaving layer execution of multiple different DNNs. Finally, we extend Layerweaver to consider QoS constraints (Layerweaver+), thereby finding a balanced schedule between latency and throughput. We evaluated our automatic data quantization and resource scheduling techniques by simulating NPU-based systems running various DNN models. For realistic DNN serving scenarios, Layerweaver and Layerweaver+ significantly improve resource utilization over the baseline scheduler serving one DNN at a time. In addition, *libnumber* substantially reduces weight and activation data size over the baseline quantization frameworks.

Keywords: Neural Networks, DNN Serving System, Quantization, Resource Scheduling, Neural Processing Units

I. Introduction

With widespread adoption of deep learning-based applications and services, computational demands for efficient deep neural network (DNN) processing have surged in datacenters [3]. In addition to already popular services such as advertisement [4, 5], social networks [6, 7, 8, 9], and personal assistants [10, 11], emerging services for automobiles and IoT devices [12, 13, 14] are also attracting great attention.

To accelerate key deep-learning applications, datacenter providers widely adopt high-performance serving systems [15, 16, 17, 18] based on specialized DNN accelerators, or *neural processing units* (*NPUs*) [19, 20, 21]. Such accelerators have a massive amount of computation units delivering up to several hundreds of tera-operations per second (TOP/s) as well as hundreds of gigabytes per second (GB/s) DRAM bandwidth. NPUs targeting only inference tasks feature a relatively high compute-to-memory bandwidth ratio (TOP/GB) [20, 22, 23], whereas those targeting both inference and training a much lower ratio due to their requirement for supporting floating-point arithmetic, which incurs larger area and bandwidth overhead.

Because designing an efficient data format is crucial for both energy efficiency and performance of NPUs, NPUs often feature customized data formats depending on their target applications. Representing a number with fewer bits accelerates computation and reduces memory footprint, hence improving overall energy efficiency. Since DNNs often have a lot of redundancy in network parameters, an 8-bit fixed-point type, instead of the 32-bit

full-precision floating-point type, may be sufficient to execute them without noticeable degradation of the output quality [23, 24, 25]. For applications requiring higher precision data (e.g., backpropagation), variants of 16-bit half-precision floating point are widely adopted, but their detailed bit fields are often different case by case [22, 26, 27]. Due to intrinsic trade-offs between accuracy and performance, it is important to find an optimal quantization scheme for a given DNN.

However, finding suitable number representations for a given network is challenging due to a combinatorial explosion in feasible number representations with varying bit widths. Furthermore, an optimal representation may vary layer by layer as different layers have different degrees of performance and accuracy sensitivity to representational changes [28]. Existing quantization frameworks for DNNs have various limitations as they fail to support multiple types [23, 24, 29, 30, 31, 32, 33], layer-wise optimization [25, 31], and are bound to a specific hardware and DNN framework [23, 31, 32, 34] (e.g., Caffe on GPU [25, 35]). Besides, the optimization goal is often hard-coded in the tuning algorithm, thus failing to accommodate various deployment scenarios with different accuracy-performance trade-offs.

Another aspect of important challenges in existing AI serving systems is that there is no one-size-fits-all accelerator that utilizes NPU resources well for all DNN models. This is because DNN models serviced in datacenters have a wide range of arithmetic intensities that cannot be covered by a single NPU. For example, convolutional neural networks (CNNs) [36, 37, 38] are conventionally known to be most compute-intensive, hence featuring the highest arithmetic intensities. In contrast, natural language processing

(NLP) [39, 40] and recommendation models [6, 41] often consist of fully-connected (FC) layers with little weight reuse to be more memory-intensive. Forming a larger batch also increases the arithmetic intensity by increasing the weight reuse, incurring further variance of arithmetic intensities among DNN models.

For an NPU required to run diverse DNN models, this mismatch between its compute-to-memory bandwidth ratio and the arithmetic intensity of the model can cause a serious imbalance between compute and memory access time. This yields a low system throughput due to under-utilization of either processing elements (PEs) or off-chip DRAM bandwidth, which in turn translates to the cost inefficiency in datacenters. Once either resource gets saturated (while the other resource is available), it is difficult to increase throughput without scaling the bottlenecked resource further.

Latency-hiding techniques such as double-buffering [42, 43], pipelining [44, 45, 46], and decoupled memory access and execution [47] have been proposed to improve NPU resource utilization. However, while mitigating the problem to a certain extent, the imbalance between compute and memory access time eventually limits their effectiveness. Moreover, merely optimizing throughput is not sufficient for inference as datacenters typically have SLA (Service-level Agreement) goals [2] that set a bound for the maximum service latency. Unfortunately, maximizing system throughput is often at odds with providing QoS for individual requests; thus, the scheduler plays a key role in balancing latency and throughput.

In summary, this thesis's eventual goal is to build an efficient software framework to boost up NPU resource utilization. We identified and solved

various problems of NPU-incorporated systems concentrating on two main categories: 1) Optimized number representation for DNNs 2) NPU resource scheduling. The main contributions of this thesis are summarized as follows:

- ***libnumber.*** To find an efficient number representation for various kinds of DNNs and NPUs, we propose *libnumber*, a portable, automatic data quantization framework for DNNs. It provides a handy and portable API that allows users to specify their optimization goals for a variety of DNN frameworks and hardware platforms. At the heart of the *libnumber* API is Number abstract data type (ADT), which subsumes most of the popular number representations for DNNs. Then the auto-tuner of *libnumber* takes inputs from the user (for target layers, accuracy constraints, and optimization goals) and the platform engineer (for a list of supported types by the target platform) to find a suitable (type, bit width, bias) tuple for each layer efficiently while satisfying the accuracy constraints.

- **Layerweaver.** To address an intrinsic mismatch between the compute-to-memory bandwidth ratio of an NPU and the arithmetic intensity of the model, which incur resource underutilization, we introduce Layerweaver. Layerweaver reduces the temporal waste of computation resources by interweaving layer execution of multiple different models with opposing characteristics: compute-intensive and memory-intensive. Layerweaver hides the memory time of a memory-intensive model by overlapping it with the relatively long computation time of

a compute-intensive model, thereby minimizing the idle time of the computation units waiting for off-chip data transfers.

- **Layerweaver+.** We further extend Layerweaver to support QoS-aware scheduling of multiple inference requests assuming more realistic datacenter use-cases (Layerweaver+). To prevent SLA violation, Layerweaver+ scheduler preciesly predicts the deadline violation incurred by throughput-oriented scheduling. And then, the scheduler switches between two operation modes to balance latency and throughput: 1) Maximizing throughput, 2) Prioritizing a request whose deadline is imminent.

Before presenting the main contributions in detail, we first discuss the preliminary of NPU-based inference serving systems and quantization techniques for DNNs in Chapter II. And then, we discuss the motivation of a more systematic approach for DNN quantization and the detailed source of NPU resource underutilization problem.

Chapter III introduces the detailed design of *libnumber*, including operation flow, Number ADT, auto-tuning algorithm, etc. This chapter is based on the previous publication [48]: Young H. Oh, Quan Quan, Daeyeon Kim, Seonghak Kim, Jun Heo, Sungjun Jung, Jaeyoung Jang and Jae W. Lee, "A Portable, Automatic Data Quantizer for Deep Neural Networks", the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18), Limassol, Cyprus, 2018.

Chapter IV presents Layerweaver's layer-wise scheduling algorithm and discuss how Layerweaver can be integrated with existing DNN serving sys-

tems. This chapter is based on the previous publication [1]: Young H. Oh, Seonghak Kim, Yunho Jin, Sam Son, Jonghyun Bae, Jongsung Lee, Yeonhong Park, Dong Uk Kim, Tae Jun Ham, and Jae W. Lee, "Layerweaver: Maximizing Resource Utilization of Neural Processing Units via Layer-Wise Scheduling", the 27th IEEE International Symposium on High Performance Computer Architecture (HPCA '21), Korea (Virtual), 2021.

Chapter V proposes a more advanced resource scheduler that dynamically balances latency and throughput, thereby preventing QoS (or deadline) violations. This chapter is revised from the unpublished manuscript: Young H. Oh, Yunho Jin, Tae Jun Ham and Jae W. Lee, "Layerweaver+: A QoS-aware, "Layer-wise DNN Scheduler for Multi-tenant Neural Processing Units", IEICE Transactions on Information and Systems (IEICE TIS' 22), 2022.

Chapter VI summarizes and concludes the thesis.

II. Background and Motivation

II.1. Preliminary: NPU-based Model Serving Systems

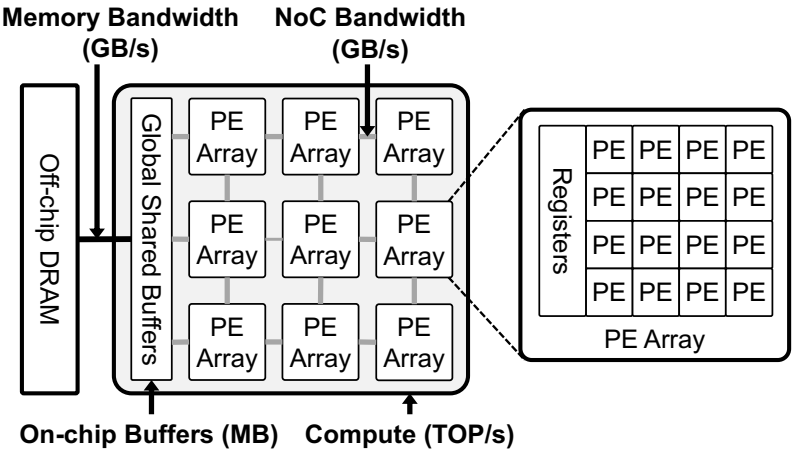


Figure II.1: An abstract single-chip NPU architecture.

NPU hardware. While GPUs and FPGAs are popular for accelerating DNN workloads, a higher efficiency can be expected by using specialized ASICs [19, 20, 45, 49], or neural processing units (NPUs). Figure II.1 depicts a canonical NPU architecture. A 2D array of processing elements (PE) are placed and interconnected by a network-on-a-chip (NoC), where each PE array has local SRAM registers. There are also globally shared buffers to reduce off-chip DRAM accesses.

Recently, with ever growing demands for energy-efficient DNN processing, specialized accelerator platforms have been actively investigated for both datacenters [10, 20, 23, 45, 50, 51] and mobile/IoT devices [14, 52, 53, 54, 55].

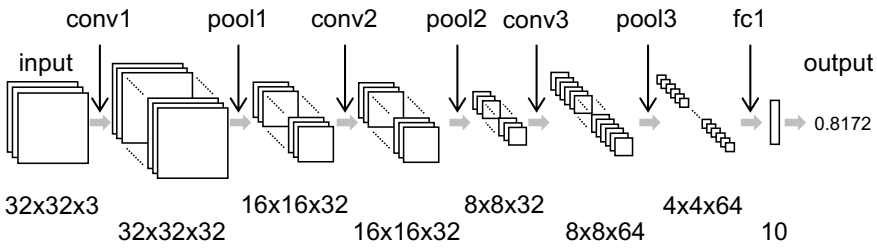


Figure II.2: Structure of Cuda-convnet

Depending on the application each request may be a single input instance or mini-batch with multiple instances. When a request arrives at the host device to which an NPU is attached, it transfers model parameters and input data to the device-side memory via the PCIe interface. Then the NPU fetches the data and commences inference computation.

DNN models. Figure II.2 shows the structure of Cuda-convnet, representing the basic layer structure of CNNs often used for image classification [56]. It takes as input an image file of 32×32 pixels with three input channels (for RGB color components) from CIFAR-10 dataset [57]. The image passes through a pipeline of kernels (layers), composed of three alternating pairs of convolution and pooling layers, followed by a fully-connected (FC) layer at the end. Each layer performs distinct matrix computations to extract more complex features toward the end of the pipeline. Finally, the FC layer is a classifier layer, which computes a vector of the probability for each of the 10 classes.

The characteristics of a DNN are determined by the characteristics of the layers it comprises. For convolution neural networks (CNNs), convolution layers typically take up most of the inference time with the remaining time

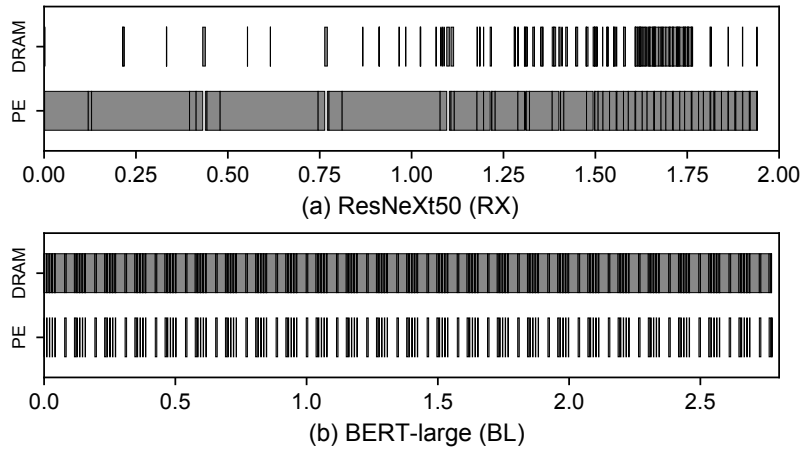


Figure II.3: Layer-wise execution timeline on a unit-batch (batch 1) input in milliseconds with (a) compute-intensive and (b) memory-intensive workload.

spent on batch normalization (BN), pooling, activation, etc. In a convolution layer most of the data (activations and filter weights) are reused with a sliding window, so it has a very high compute-to-memory bandwidth ratio (i.e., highly compute-intensive). Figure II.3 shows an example NPU execution timeline of ResNeXt50 [36], where PEs are very heavily used while DRAM bandwidth is under-utilized. In contrast, most neural language processing (NLP) and recommendation models are dominated by fully-connected (FC) layers which have little reuse of weight data. Thus, the FC layer has memory-intensive characteristics. As shown in Figure II.3, BERT-large [39] has high utilization of off-chip DRAM bandwidth but much lower utilization of PEs.

Figure II.4 is a sequential C reference code for the simple FC layer. This kernel takes a three-dimensional matrix representing 64 4×4 feature maps as input to generate a one-dimensional probability vector for the 10 classes. The operations being performed are multiplying an input activation (`input[j]`

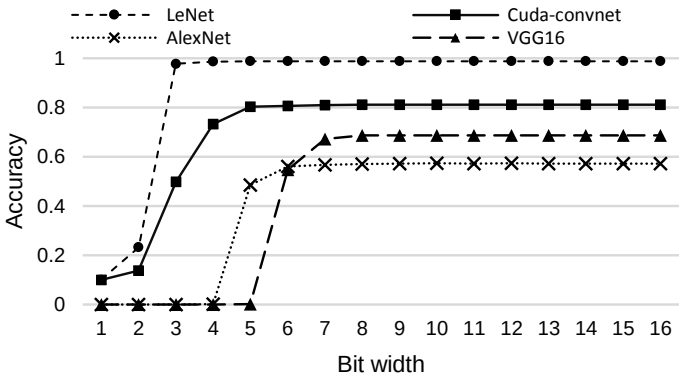
```
1  float output[out_c];
2  float input[in_c][h][w];
3  float filter[out_c][in_c][h][w];
4
5  // Computation of fully connected layer
6  for (int i=0; i<out_c; ++i) {
7    for (int j=0; j<in_c; ++j) {
8      for (int k=0; k<h; ++k) {
9        for (int l=0; l<w; ++l) {
10         output[i]+=filter[i][j][k][l]*
11                    input[j][k][l];
12 }}}}
```

Figure II.4: FC layer reference code (simplified)

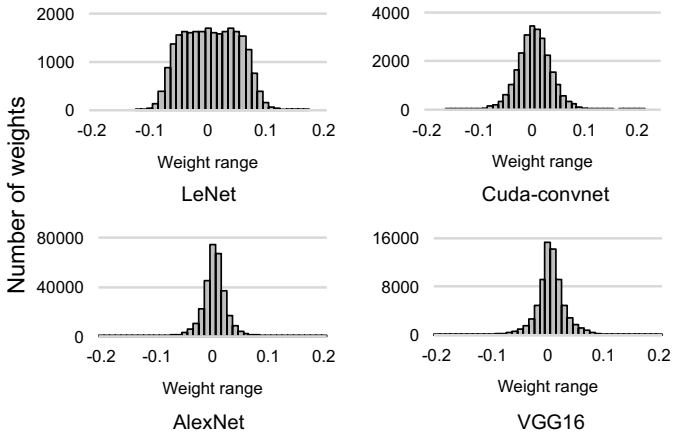
`[k][l]`) by a weight (`filter[i][j][k][l]`) and accumulating it into the output vector (`output[i]`). This multiply-accumulate (MAC) computation is the most common operations not only for the FC layers but also for the convolution layers.

However, it is well known that the network parameters in DNNs have a significant amount of redundancy [58] and we can improve computational efficiency by eliminating it. Figure II.5(a) shows the top-1 accuracy of four popular CNNs with varying fixed-point bit widths from 1 through 16. All of the four CNNs fully recover their accuracy by using 8 or more bits. Moreover, LeNet requires only 3 bits to achieve the full accuracy.

Thus, there are ample opportunities for quantization to compress the network parameters and boost computational efficiency by using lower-precision arithmetic. Figure II.5(b) shows the distribution of the weights for the four CNNs. The distribution features a relatively short dynamic range of the values. If a wide dynamic range is unnecessary, simpler (and faster) fixed-point computation can replace expensive full-precision floating-point computation. To exploit these opportunities on a GPU, Nvidia has recently introduced



(a) Top-1 accuracy-bit width



(b) Distribution of weight parameters

Figure II.5: Redundancy in CNN weight parameters

mixed-precision support for their state-of-the-art CUDA 9 [59]. Microsoft's BrainWave deploys a custom 9-bit floating-point format, called *ms-fp9*, to achieve a sub-millisecond latency of a DNN inference task on their FPGA fabric [60].

Serving System. The inference serving system is built on top of NPU hardware to efficiently handle a massive amount of DNN inference requests. Such systems often come with a scheduling mechanism that can meet SLA goals for

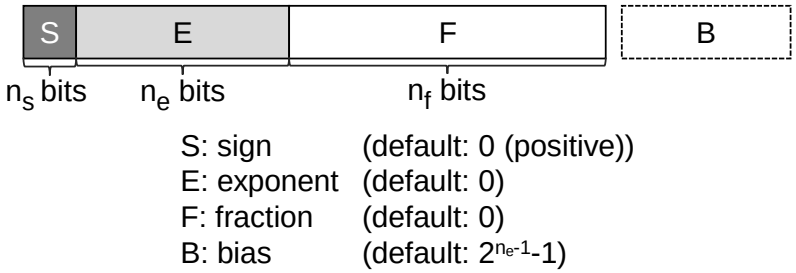


Figure II.6: Canonical number format based on the IEEE 754 standard

each application. One challenge in designing such a scheduling mechanism is that the system does not know when the next inference requests will arrive. Thus, the system needs to judiciously decide whether to wait for additional requests for higher throughput (by forming a larger batch) or swiftly process already arrived requests for lower latency. Several existing inference serving systems [15, 17, 18, 61] utilize several heuristics to determine the amount of time the system waits for to form a batch. Moreover, over 80% of applications in Google datacenters already require multi-tenancy [22] for utilizing multiple models and A/B testing new features. The more detailed step-by-step process of DNN deployment on NPU-based systems are described in the later at "Deployment" paragraph in Section IV.1.

II.2. Optimizing Number Representations

Table II.1 surveys popular number representations used by DNNs. Floating-point, fixed-point, and exponent are the three most widely used types for quantization. (The binary type is a special case of any of the three.) We observe that all of the three types can be represented by the canonical

Table II.1: Survey of popular number representations for DNNs (– means not disclosed.)

Format	**Number Type**	**Bit Width**	**Network**	**Quantized Layer**	**Canonical Form** $\langle n_s, n_e, n_f, B \rangle$
FLOAT32	floating-point	32	Default for all DNNs	CONV, FC	$\langle 1, 8, 23, 127 \rangle$
Half-precision		16	CNN, LSTM, DCGAN, DeepSpeech2 [59]	CONV, FC	$\langle 1, 5, 10, 15 \rangle$
MS-fp9, BFP		9	DNNs on Brainwave [26, 62]	CONV, FC	$\langle 1, 5, 3, - \rangle$
FIXED16	fixed-point	16	CaffeNet, VGG16, VGG16-SVD [31]	CONV, FC	$\langle 1, 0, 15, 0 \rangle$
FIXED11		10	CNN [30]	CONV	$\langle 1, 0, 10, - \rangle$
FIXED8		8	MLP, CNN, LSTM [23]	CONV, FC	$\langle 1, 0, 7, - \rangle$
LogQuant4	exponent	4	AlexNet, VGG16 [33]	CONV, FC	$\langle 1, 3, 0, 3 \rangle$
BIN	binary	1	BinarizedNN [63]	CONV, FC	$\langle 1, 0, 0, 0 \rangle$

number format based on IEEE 754 Standard as shown in Figure II.6. The canonical format consists of four fields: sign (S), exponent (E), fraction (F) and bias (B). [1] If the lengths of the S, E, and F fields are denoted by n_s, n_e, and n_f, respectively, a number format can be characterized by a 4-tuple of $\langle n_s,n_e,n_f,B\rangle$. Assuming $E = \sum_{i=0}^{n_e-1} 2^i \cdot e_i,\ F = \sum_{i=0}^{n_f-1} 2^i \cdot f_i,$, where e_i and f_i denote the i-th bit of E and F, respectively, the number can be interpreted as follows:

$$\begin{cases} (-1)^S(1 + F \cdot 2^{-n_f})\ 2^{E-B} & if\ E \neq 0 \\ (-1)^S F \cdot 2^{-n_f} \cdot 2^{1-B} & if\ E = 0 \end{cases} \tag{II.1}$$

More specifically, the three popular number types are mapped to the the canonical format as follows. (1) N-bit ***floating-point*** (`FLOAT(N)`) is represented by a 4-tuple of $\langle 1,\ n_e,\ n_f,\ B\rangle$, where $N = 1 + n_e + n_f$ and $B = 2^{n_e-1}$-1. For example, single-precision (`FLOAT(32)`) and half-precision (`FLOAT(16)`) types correspond to $\langle 1,\ 8,\ 23,\ 127\rangle$ and $\langle 1,\ 5,\ 10,\ 15\rangle$, respectively. (2) N-bit ***dynamic fixed-point*** (`FIXED(N)`) is represented by a 4-tuple of $\langle 1,\ 0,\ n_f,\ B\rangle$, where B is used to control the position of the radix point (zero by default). Unlike the floating-point type, the fixed-point type has a narrow dynamic range, but is easier to implement using integer arithmetic. (3) N-bit ***exponent*** (`EXP(N)`) is represented by a 4-tuple of $\langle 1,\ n_e,\ 0,\ B\rangle$, where $B = 2^{n_e-1}$-1 by default. The exponent type has the widest dynamic range at the cost of reduced precision.

[1] Throughout this thesis a *bias* refers to a constant subtracted from an exponent to form a biased exponent in compliance with the IEEE 754 standard. It is different from a constant input (bias) to an artificial neuron.

In this setup, the task of optimizing number representations is reduced to finding an optimal set of the four parameters (n_s, n_e, n_f, and B) in the 4-tuple. However, this task easily becomes intractable due to a combinatorial explosion in feasible number formats. Suppose an FPGA device that can accommodate the three number types: floating-point (32 and 16 bits), fixed-point (1 through 32 bits), and exponent (2 through 9 bits). For example, if we are to find an optimal representation out of the 42 formats for both activations and weights, and each of the nine convolution layers in DarkNet [64], the search space is as large as $(42^2)^9 = 1.65 \times 10^{29}$.

To avoid this cost, prior proposals for quantization sacrifice coverage of the search space by either tuning the bit width only for a fixed number type [23, 24, 29, 30, 31, 32, 33] or turning off layer-wise optimization [25, 31]. This can lead to a suboptimal result. Furthermore, the proposed techniques often target a specific DNN framework and hardware device [23, 31, 32, 34], lacking portability across different execution environments for DNNs.

Thus, we need a quantization framework for DNNs with broad coverage in both optimization space and execution environments. The auto-tuner should find a suitable number format efficiently for each target layer. Besides, it is highly desirable for the framework to provide a portable, easy-to-use API that can flexibly support a variety of DNN frameworks and hardware platforms.

II.3. Scheduling to Maximize NPU Resource Utilization

Compute-centric vs. Memory-centric NPUs. NPUs designed for training in datacenter environments are generally capable of servicing multiple requests

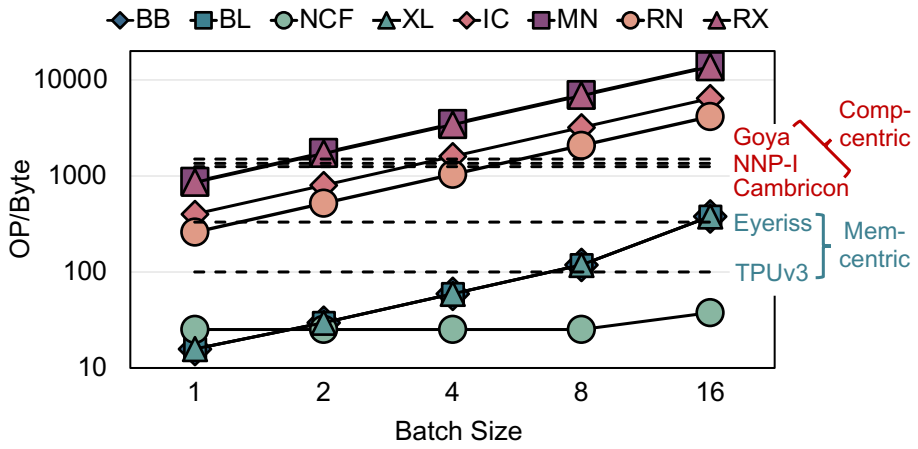


Figure II.7: Compute-to-memory bandwidth ratios across varying batch sizes. Arithmetic intensity (Number of operations divided by off-chip access bytes) is measured for DNN models. Peak TOPS is divided by peak off-chip DRAM bandwidth for NPUs.

at once as training throughput is the most important measure. In contrast, for inference, NPUs are often required to provide a low latency for a single request to not degrade user experience. As such, depending on the requirements from the target environment, NPUs have varying compute-to-memory bandwidth ratios.

Figure II.7 overlays the arithmetic intensity of eight popular DNN models with the compute-to-memory bandwidth ratio of five NPUs while varying the batch size of the input. The following four of the eight models are known to be *compute-intensive*: InceptionV3 (IC) [65], MobileNetV2 (MN) [66], ResNet50 (RN) [37] and ResNeXt50 (RX) [36]. The remaining four models are *memory-intensive*: BERT-base (BB), BERT-large (BL) [39], NCF (NCF) [41] and XLNet (XL) [40]. Furthermore, we also classify five NPUs into *compute-centric* designs [20, 21, 67], which have a relatively high compute-to-memory bandwidth ratio, and *memory-centric* designs [19, 68], which are characterized

by a low ratio. If a compute-intensive model (e.g., ResNeXt50) is executed on a memory-centric NPU (e.g., TPUv3), the memory bandwidth resource is likely to be under-utilized (i.e., bottlenecked by PEs) to yield sub-optimal system throughput.

Imbalance between DNN Model and NPU. There is a wide spectrum of NPUs and DNN models to make it nearly impossible to balance the NPU resources with the requirements from *all* DNN models. In Figure II.7, for a given NPU, the farther the distance from the horizontal line of the NPU to the arithmetic intensity of the workload, the greater degree of imbalance exists to yield poor resource utilization of either PEs or DRAM bandwidth. If the arithmetic intensity of the DNN model is above the horizontal line of the NPU, DRAM bandwidth is under-utilized, or vice versa.

Figure II.8 characterizes the resource utilization in greater detail. The two bars for each DNN model show normalized active cycles (i.e., temporal utilization) of both resources for unit-batch (batch 1) and multi-batch (batch 16) inputs, respectively. We use both compute-centric (NNP-I) and memory-centric NPUs (TPUv3) in Figure II.8(a) and (b). The details of this experimental setup are available in Section IV.5.A.

In Figure II.8(a), the memory-centric NPU (TPUv3-like) under-utilizes DRAM bandwidth for compute-intensive models and PE resources of memory-intensive models at the unit batch (batch 1, white bars). This under-utilization is explained by the large gap between the horizontal line of the NPU and the DNN arithmetic intensity. However, at batch 16 (gray bars), memory-intensive DNN models show high PE utilization and lower DRAM bandwidth utilization compared to unit batch size except for NCF, which

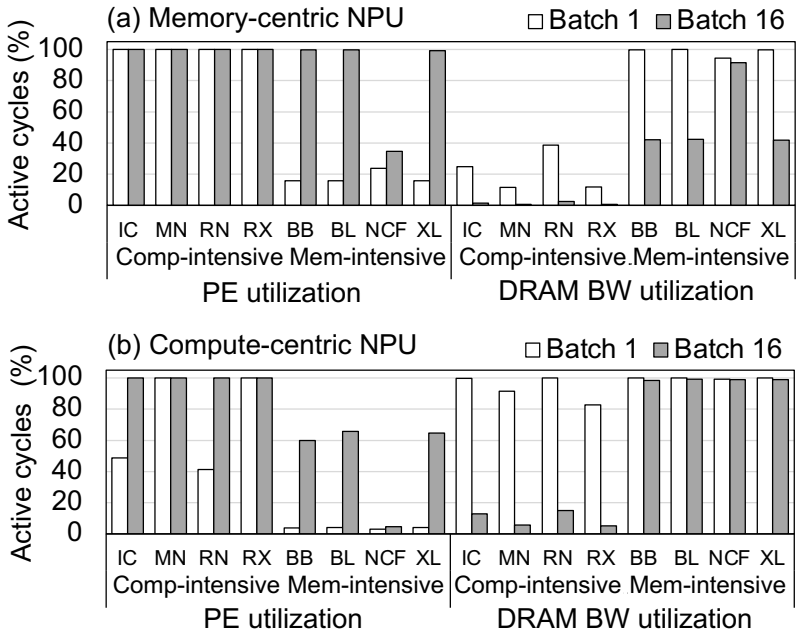


Figure II.8: Normalized active cycles of compute-centric and memory-centric NPUs on various DNN models.

is extremely memory-intensive, due to an increase in the arithmetic intensity. Figure II.7 shows that the points of the memory-intensive workloads are located above the horizontal line of TPUv3 except for NCF. Still, the compute-intensive models show very low DRAM bandwidth utilization.

In Figure II.8(b), the compute-centric NPU (NNP-I-like) shows opposite results. Some compute-intensive workloads demonstrate relatively low PE utilization because of insufficient computations at the unit batch. However, with batch 16, DRAM bandwidth is under-utilized while PE resources are fully saturated. Conversely, in the case of memory-intensive workloads, PE resources are still not fully saturated even at batch 16 as DRAM bandwidth gets saturated first.

III. *libnumber*: Portable, Automatic Data Quantizer for DNNs

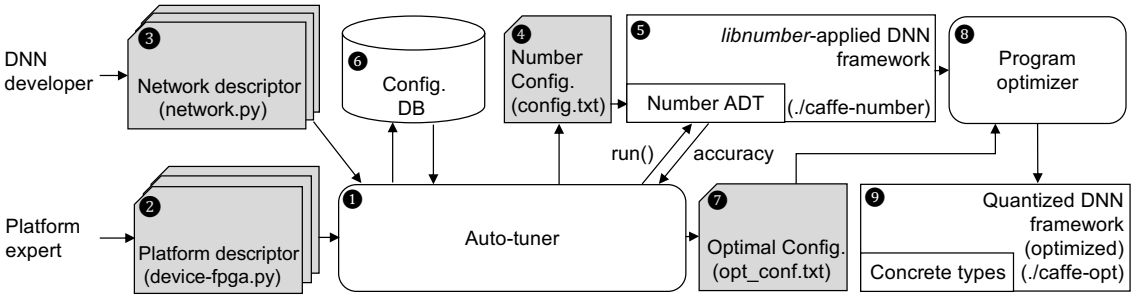


Figure III.1: Overall operation flow of the *libnumber* quantization framework.

III.1. Overview

Figure III.1 shows the overall operation flow of *libnumber*. The output of the quantization framework is a DNN framework (e.g., TensorFlow) optimized for a given network model by applying quantization to target layers as identified by the DNN developer. The two main components of *libnumber* are `Number` abstract data type (Number ADT) and an auto-tuner for number representations, which are presented in greater details in Section III.2 and Section III.3, respectively. The rest of this section sketches a flow of operations to quantize the activations and weights of each layer with *libnumber*.

Step 1 (Initializing the Auto-tuner). At the core of *libnumber* is the auto-tuner ❶ around which multiple components interact with each other. It can be flexibly configured using two descriptor files written in Python. The first

one is a *platform descriptor* ❷ containing a key-value pair, where the key is a hardware device name and the value is a list of supported number formats. The second one is a *network descriptor* ❸, which specifies the optimization goal (i.e., accuracy tolerance, objective function) and a list of target layers among others. Using this information the auto-tuner determines search space, and enters the tuning loop to find a configuration that *minimizes* the objective function, denoted by f_{obj}, subject to satisfying the accuracy constraint.

Step 2 (Entering the Tuning Loop). Once initialization is completed, the auto-tuner selects the first configuration to try, and generates a Number configuration file ❹ in text format. This file contains a list of object name-number format pairs (e.g., `conv1.weight FLOAT 32`, `fc1.weight EXP 8 127`). Then the auto-tuner launches an inference job by invoking the *libnumber*-applied DNN framework ❺ in which data to be quantized (e.g., layer inputs, weights, or both) are declared as `Number` type. A `Number` type subsumes all number formats that can be represented in the canonical format in Figure II.6. At runtime all `Number` objects in the DNN framework are bound to concrete number formats as specified by the Number configuration file.

Step 3 (Assessing the Configuration). When the DNN inference is finished, the auto-tuner first updates the configuration database ❻ with accuracy and the value of $f_{obj}(cfg)$ for the current configuration `cfg`. If a configuration that minimizes f_{obj} is found, the auto-tuner reports the configuration in a plain text file (`opt_conf.txt`) ❼ and exits the loop. Otherwise, it selects the next candidate configuration and repeats the process in Step 2. The auto-tuning algorithm is presented in greater details in Section III.3.

Step 4 (Generating Optimized DNN Kernels). Once the optimal configuration

Method	Descriptions
void Number::parseConfig (string fname)	Parse the `Number` configuration file to build a global object-format dictionary
void Number::setName (string key)	Set the name of the `Number` object and initialize its 4-tuple format attributes $\langle n_s, n_e, n_f, B \rangle$ for its generic representation
void Number::operator=()	Assign a value from another [`Number`|`float`|`int`] value
Number& Number::operator+()	Add a [`Number`|`float`|`int`] value to the `Number` value
Number& Number::operator*()	Multiply the `Number` value by a [`Number`|`float`|`int`] value
float Number::asFloat()	Export the `Number` value as a `float` type
int Number::asInt()	Export the `Number` value as an `int` type

Table III.1: `Number` ADT API in C++

is found, the program optimizer ❽, either a human or compiler, generates optimized DNN kernels ❾ by applying quantization to the target layers. Since the optimal configuration file contains a list of object name-optimal format pair, the optimizer may simply replace each `Number` object with an object of the corresponding concrete type. Furthermore, reducing the bit width can expose additional optimization opportunities, which may not be feasible in the out-of-the-box kernels. At this point this transformation and optimization is done manually, but we believe this can be automated in the future.

III.2. Number Abstract Data Type

API Design. Table III.1 summarizes the C++ API of the `Number` ADT. The methods are divided into three categories: configuration, computation, and type conversion. The `parseConfig` method takes a `Number` configuration file generated by the auto-tuner as input and builds a dictionary of (object name: (type, format)) pairs (e.g., `conv1.weight: (EXP, <1, 7, 0,`

```
1  Number::parseConfig("config.txt");
2
3  float output[out_c];
4  Number input[in_c][h][w];
5  Number filter[out_c][in_c][h][w];
6  // Format setting
7  // 'layername' is passed by caller
8  setArrayName((Number*)filter,
9    out_c*in_c*h*w, layername+".weight");
10 setArrayName((Number*)input,
11   in_c*h*w, layername+".input");
12
13 // Computation of fully connected layer
14 for (int i=0; i<out_c; ++i) {
15   for (int j=0; j<in_c; ++j) {
16     for (int k=0; k<h; ++k) {
17       for (int l=0; l<w; ++l) {
18         output[i]+=(filter[i][j][k][l]*
19              input[j][k][l]).asFloat();
20 }}}}
```

(a) FC layer with `Number` ADT

```
1  // config.txt
2  conv1.weight    EXP 8
3  conv1.input     EXP 6
4  ...
5  fc1.weight      FIXED   15  -3
6  fc1.input       FLOAT   32
7  ...
```

(b) Configuration file

```
1  // Produced by Number::parseConfig
2  {
3    conv1.weight : (EXP, <1,7,0,63>),
4    conv1.input  : (EXP, <1,5,0,15>),
5    ...
6    fc1.weight   : (FIXED, <1,0,14,-3>),
7    fc1.input    : (FLOAT, <1,8,23,127>),
8    ...
9  }
```

(c) Dictionary of obj name and format pairs

Figure III.2: Application of `Number` ADT to fully-connected layer reference code

`63>)`). This dictionary is a global structure shared by all `Number` objects. `setName(key)` sets the name of the object (e.g., `conv1.weight`), which will be used as a key to access the dictionary, and initializes the type (`_type`) (e.g., `EXP`) and format attributes (e.g., `<1, 7, 0, 63>`) for the object. If a dictionary lookup fails, the full-precision 32-bit floating-point format is used by default.

There are three computational operators for the `Number` ADT: assignment, multiplication, and addition. When an assignment is made from another `Number` object, both the value and format attributes are copied over from the object. If the right-hand side (RHS) operand has a native type (e.g., `float`), its format attributes are set appropriately (e.g., (`FLOAT`, `<1, 8, 23, 127>`). For multiplication and addition, we generally follow the operation rules in the IEEE 754 standard. However, a special care should

be given when the two operands have different format attributes. In this case, these methods internally convert both objects into full-precision 32-bit floating-point values, compute the result, and put it back to a `Number` object by taking the format attributes from the object with a wider dynamic range, except that we favor `FLOAT` type over `EXP` for higher precision. In this way we can prevent value saturation.

Finally, the `Number` ADT API provides a couple of type-conversion methods: `asFloat()` and `asInt()`. These methods extract the value from a `Number` object and export it as either `float` or `int` value.

Example of Applying `Number` ADT. Figure III.2(a) shows a transformed reference code of the fully-connected layer taken from Figure II.4, with modified part shown in gray. In this example, we assume the user wants to quantize both weight parameters and input activations; thus, we declare the `filter` and `input` arrays in `Number` type instead of `float` (Line 4 and 5). In Line 1, `parseConfig()` class method is invoked to load the format configuration from a `Number` configuration file. Figure III.2(b) is an example of the file. It contains a list of object name and format pairs. The format field has two parameters, type and bit width, and an optional third parameter for bias (in Line 5, for example). By invoking `parseConfig`, a dictionary is created as shown in Figure III.2(c).

Lines 8-11 set the object configuration of each element in the `filter` and `input` arrays, respectively, by invoking `setArrayName()`, which internally calls the `setName()` method. In this example, object names are `fc1.weight` and `fc1.input`; thus, the format attributes will be set to be $\langle n_s, n_e, n_f, B \rangle = \langle 1, 0, 14, -3 \rangle$ with `_type` = `FIXED` and $\langle n_s, n_e, n_f, B \rangle = \langle 1,$

```
1  supported_formats = {
2       'FPGA':{                  # Platform name
3       'FLOAT': [16, 32],        # FLOAT 16 / 32 bits
4       'FIXED': range(2, 33),    # FIXED 2 to 32 bits
5       'EXP'  : range(2, 10)}}   # EXP   2 to 9  bits
```

Figure III.3: Example platform descriptor file

8, 23, 127⟩ with `_type = FLOAT`. The loop body in Lines 18-19 requires only minor modifications as the multiply operator (*) handles a multiply of a `Number` operand by a `float` operand. As the result has a `Number` type, it needs to be converted to a `float` value by invoking the `asFloat` method. Overall, it only takes minimal effort to port an existing DNN kernel to the `Number` ADT.

III.3. Auto-tuner Design

III.3.A. Configuring the Auto-tuner

The auto-tuner takes two descriptor files to configure it: platform and network descriptors. The platform descriptor specifies a `supported_formats` attribute as shown in Figure III.3. It contains a key-value pair, where the key is a device name and the value is a list of supported number formats. There can be multiple platform descriptor files to support multiple hardware devices. Besides, one can specify different supported types for activations and weights by appending `.activation` and `.weight` suffix to the device name.

The network descriptor file specifies various network parameters as summarized in Table III.2. We need two command lines (`cmd_smallset` and

Table III.2: Auto-tuner configuration parameters

Platform descriptor		
Parameter	**Description**	**Example**
supported_formats	Key-value pair of device name and list of supported number formats	Refer to Figure III.3

Network descriptor		
Parameter	**Description**	**Example**
platform	Device name	“FPGA”
cmd_smallset	Command line to run small test set	”./caffe test -model LeNet.prototxt -weight Lenet.caffemodel -iterations 10”
cmd_fullset	Command line to run full test set	”./caffe test -model LeNet.prototxt -weight Lenet.caffemodel -iterations 100”
err_margin	Error margin relative to baseline accuracy	0.07
f_obj	Objective function to specify optimization goal	“Default” (sum(bit_width[i]*layer_size[i]))
layer_name	Name of target layers in Python list format	[”conv1”, ”conv2”]
sub_layer_name	Name of branches in same layer in Python list format	[[],[”branch1”, ”branch2”...]...]
layerwise_opt	Enable layer-wise optimization [BOTH|WGT-ONLY|ACT-ONLY|NONE]	BOTH
wgt_size	Size of target layers (weight count) in Python list format	[500, 25000]
act_size	Size of target layers (activation count) in Python list format	[784, 2880]
act_max_abs	Maximum absolute values of activations for target layers in Python list format	[0.996094, 5.05664]
wgt_max_abs	Maximum absolute values of weights for target layers in Python list format	[0.5794976, 0.16474459]
get_result_regex	Regular expression to extract accuracy from standard output	”Accuracy : (.*)\n”

cmd_fullset) to invoke the DNN framework with small and full test sets, respectively, as the tuning algorithm in Section III.3.B uses both. The user specifies an accuracy constraint and optimization goal by setting err_margin and f_obj. At this point we only use the default objective function, which is a sum of bit width-layer size products for all target layers, to minimize overall storage requirements for activations and weights. However, one can override this function to customize the optimization goal. A regular expression is provided by get_result_regex to extract the accuracy number from the console output at the end of execution of every run in the tuning loop. The user can also control layer-wise optimization for both activations and weights by setting the layerwise_opt field. The remaining parameters are self-explanatory, except that sub_layer_name is used to group multiple sublayers using the same format. This feature is useful for optimizing networks with branches such as ResNet [69] and GoogleNet [70].

III.3.B. Tuning Algorithm

To achieve high quality and efficiency at the same time, the auto-tuner takes a two-pass approach, where each pass consists of two phases. During the first pass, the auto-tuner uses a small test set (specified by cmd_smallset) to reduce not only the number of iterations but also the cost of each iteration. Using a smaller dataset can yield an order of magnitude reduction in execution time, hence leading to significant savings in total search time, especially for large and deep networks. Starting from the best configuration of the first-pass, the second pass further tunes (type, bit width, bias) tuples

Algorithm 1 Auto-tuning algorithm

```
Input: Error margin err_margin
Output: Best configuration best_cfg
 1: init_cfg ← InitializedConfig()
 2:
 3: test_set ← SmallSet
 4: threshold = GetAccuracy(init_cfg) × (1 - err_margin) // small set
 5:
 6: /* Phase 1 (small set): Coarse-grained search */
 7: cfg ← init_cfg
 8: while GetAccuracy(cfg) >= threshold do
 9:     cfg ← CoarseGrainedSearch(cfg)
10: end while
11:
12: /* Phase 2 (small set): Fine-tuning bias, bit width and type */
13: cfg ← FinetuneBiasBitWidthAndType(cfg)
14:
15: test_set ← FullSet
16: threshold = GetAccuracy(init_cfg)×(1 - err_margin) // full set
17:
18: /* Phase 3 (full set): Accuracy recovery */
19: if GetAccuracy(cfg) < threshold then
20:     cfg_set ← GetConfigsWithOneMoreBit(cfg) ∪
21:                 GetConfigsWithDecrementedBias(cfg)
22:     while IsAllBelowThreshold(cfg_set) do
23:         cfg ← SelectBestConfig(cfg_set)
24:         cfg_set ← GetConfigsWithOneMoreBit(cfg) ∪
25:                     GetConfigsWithDecrementedBias(cfg)
26:     end while
27: end if
28:
29: /* Phase 4 (full set): Fine-tuning bias, bit width and type */
30: best_cfg ← FinetuneBiasBitWidthAndType(cfg)
```

using the full test set (specified by `cmd_fullset`). The rest of this section presents the details of the four phases described in Algorithm 1 and 2 using a real example of auto-tuning LeNet in Figure III.4.

Phase 1 (Coarse-grained Search). The first phase of the algorithm is de-

Algorithm 2 Fine-tuning bias, bit width and type

Input: Configuration cfg
Output: Best configuration best_cfg

```
 1: while True do
 2:     cfg_set ← GetConfigsWithIncrementedBias(cfg)
 3:     while ¬ IsAllBelowThreshold(cfg_set) do
 4:         cfg ← SelectBestConfig(cfg_set)
 5:         cfg_set ← GetConfigsWithIncrementedBias(cfg)
 6:     end while
 7:     cfg_set ← GetConfigsWithOneFewerBit(cfg)
 8:     while ¬ IsAllBelowThreshold(cfg_set) do
 9:         cfg ← SelectBestConfig(cfg_set)
10:         cfg_set ← GetConfigsWithOneFewerBit(cfg)
11:     end while
12:     cfg_set ← GetConfigsWithTypeChange(cfg)
13:     if IsAllBelowThreshold(cfg_set) then
14:         break
15:     else
16:         cfg ← SelectBestConfig(cfg_set)
17:     end if
18: end while
19: return cfg
```

scribed by Line 1-10 in Algorithm 1. This phase performs a coarse-grained search to reduce the bit width starting from the baseline configuration of using `FLOAT(32)` for all layers (Line 1) and a small test set (Line 3). The coarse-grained search procedure (`CoarseGrainedSearch` in Line 9) reduces the bit width by half as long as the accuracy constraint is satisfied. Once the search hits the minimum bit width for `FLOAT`, it switches to the `FIXED` type and continues. If it hits a configuration that violates the accuracy constraint, it selects the configuration in the middle of the last two configurations to perform a binary search. Iterations 1-7 in Figure III.4 illustrate this phase when tuning the two convolution layers (L1 and L2) of LeNet. As a result, 3-bit

fixed-point (I3) is selected as best configuration thus far (Iteration 6). The procedure also attempts to further reduce the bit width using the `EXP` type but fails (Iteration 7). Note that the biases surrounded by parentheses are initialized based on value profiling, as summarized by `act_max_abs` and `wgt_max_abs` parameters in Table III.2.

Phase 2 (Fine-tuning Bias, Bit Width and Type). The second phase performs a fine-tuning of bias, bit width and type as described in Algorithm 2, starting from the best configuration from Phase 1. First, the algorithm attempts to tune bias for fixed-point and exponent types (Line 2-6), which is crucial to balance the precision and dynamic range of a quantized number. The goal is to minimize the dynamic range for each layer (i.e., maximizing bias), while satisfying the accuracy constraint. Stripes [71] performs similar tuning to minimize the number of integer bits. In Line 2, we first collect a set of all configurations (`cfg_set`) whose bias is incremented by one from the current configuration at a target layer (for either activations or weights. For example, Iterations 8-11 in Figure III.4 show the four elements in `cfg_set`. Among them the third one (I3(0)-I3(3) for weights and I3(0)-I3(-3) for activations in Iteration 10) is selected due to highest accuracy, where I3(0) denotes 3-bit fixed-point (I) with bias 0. This process is repeated until no configuration satisfies the accuracy constraint.

Second, the algorithm proceeds to reduce bit width (Line 7-11). In Line 7, it first collects a set of all configurations (`cfg_set`) that use one fewer bit than the current configuration for either activations or weights. In Figure III.4, if the current configuration is I2(1)-I3(3) for weights and I3(0)-I3(-2) for activations (selected at Iteration 25), the three configurations in Iterations

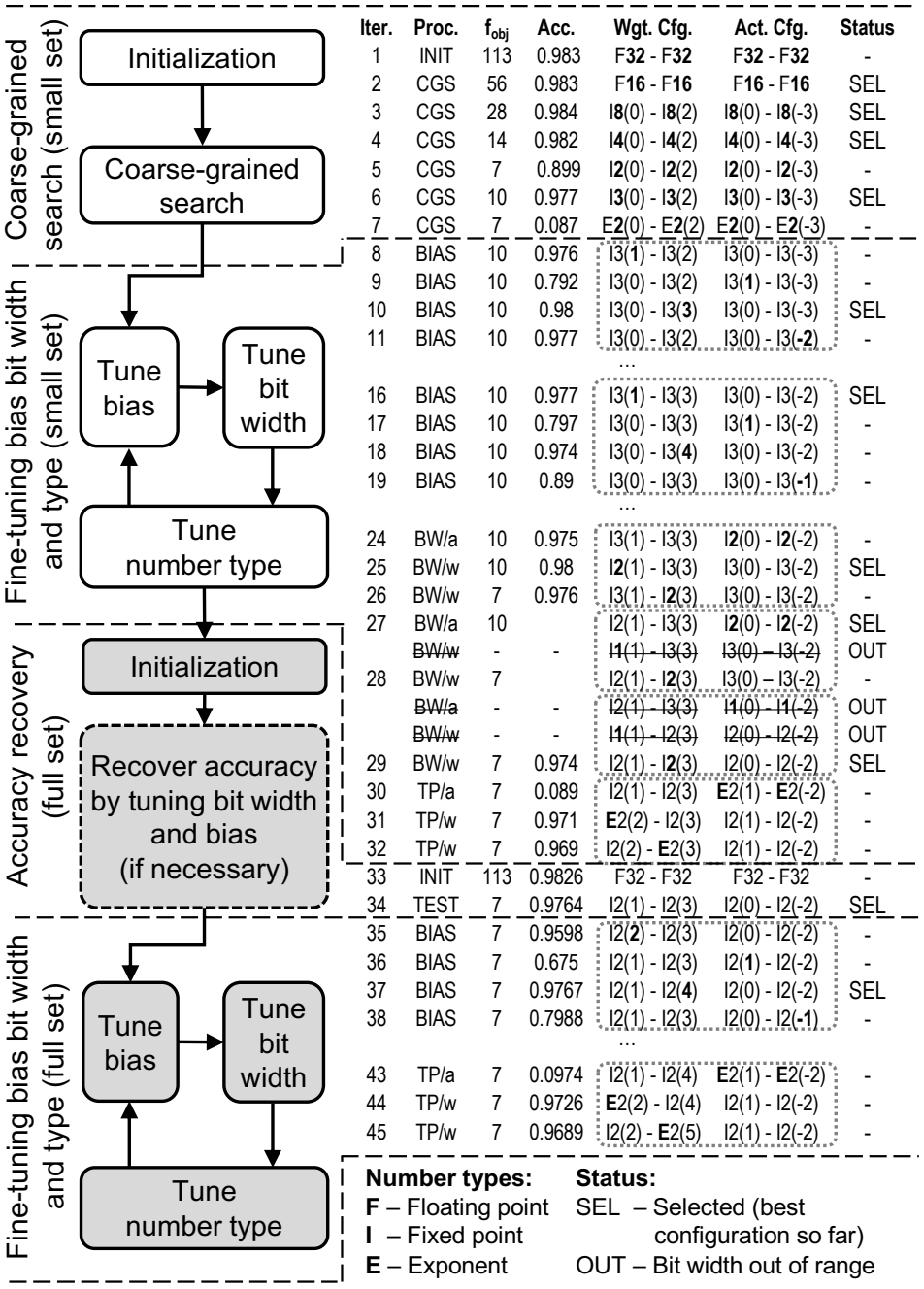


Figure III.4: Auto-tuning the two CONV layers of LeNet with maximum accuracy loss of 1% (accuracy threshold: 0.9728). Biases (B) are shown in parentheses. Layer-wise optimization is enabled only for weights.

27-28 represent `cfg_set`. Then the configuration that gives the most benefit (i.e., maximizing $\Delta f_{obj}\,(cfg)$) at the minimum cost of accuracy loss (i.e., minimizing $\Delta accuracy\,(cfg)$) is selected. This process is repeated until no configuration satisfies the accuracy constraint (Line 8-11). For example, the minimum bit widths are found at Iteration 29 in Figure III.4 (I2(1)-I2(3) for weights and I2(0)-I2(-2) for activations).

Finally, we perform fine-tuning of the number type (Line 12-17). In this step, we first collect a set of all configurations (`cfg_set`) that have only one type different from the current configuration. In Figure III.4, if the current configuration is I2(1)-I2(3) for weights and I2(0)-I2(-2) for activations (from Iteration 29), the configurations in Iterations 30-32 represent `cfg_set`. If no configuration satisfies the accuracy constraint, the algorithm exits the loop; otherwise, it selects the one with the highest benefit-to-loss ratio as in the previous step, and performs tuning of bias and bit width again. In the example of LeNet, the aforementioned configuration from Iteration 29 is selected as the best one.

Phase 3 (Accuracy Recovery). From this phase the algorithm enters the second pass to use the full test set for evaluating accuracy (Line 15-27 in Algorithm 1). Since we change the test set, there is no guarantee that the current best configuration still satisfies the accuracy constraint. If the current configuration satisfies the constraint, the algorithm just moves to the next phase; otherwise, it increases the bit width and/or decrements the bias until the constraint is satisfied (Line 22-26). This algorithm is the same as the fine-tuning algorithm for bit width in Phase 2 except that it increments (decrements) the bit width (bias) instead of decrementing (incrementing) it.

This loop is repeated until the algorithm finds the first configuration whose accuracy is above the threshold. In the example of LeNet in Figure III.4, we skip this phase as the accuracy constraint is already satisfied.

Phase 4 (Fine-tuning Bias, Bit Width and Type). Finally, we repeat the same fine-tuning process of bias, bit width and number type as in Phase 2, but using the *full* test set. In Figure III.4 the algorithm finally outputs the best configuration for LeNet, which is I2(1)-I2(4) for weights and I2(0)-I2(-2) for activations.

III.3.C. Discussion

Optimality of Search Algorithm. Due to the ever-wider range of number representations that *libnumber* considers, finding an optimal solution within a feasible time is very challenging. One possible way to handle this problem is to apply the traditional auto-tuning algorithm [72, 73, 74] used for compilers. However, because auto-tuning algorithms are often built with ensembles to handle a wide range of tuning parameters, they typically require more iterations. Additional tuning iterations for this problem are very costly because each iteration takes much longer to evaluate the entire test dataset than the code compilation time. Another option is a hyperparameter tuning tool often relying on tree-based gradient boosting [75] or simulated annealing [76]. However, they also increase the number of iterations incurred by several tree-boosting or random search, making the deep exploration of number representations practically impossible. More recent approaches [77, 78] support differentiable parameter search with a customized cost function. The strength

of differentiable search is that they enable a systematic quantization search guided by a cost function that is naturally integrated with DNN training. However, whether such techniques would work for a wide range of possible number representations is still in question. We compare the final compression ratio with the true optimal case of each dataset by performing an exhaustive search whenever possible. For Cifar10 dataset, our tuning algorithm shows only 0.1% less compression rate than the true optimal while achieving the optimal compression rate for MNIST dataset. Although the first-order design goal of *libnumber* is to design a portable interface that can cover various number representations, we have experimentally shown that our tuning algorithm provides good-enough solutions for various DNNs (Section III.4).

Effectiveness of Subsampling Datasets. Our two-pass algorithm is designed for fastly pruning search spaces, which are far from the final optimization goal. The search algorithm randomly subsamples a dataset at the first pass and then finds an efficient number representation. The second pass performs a similar search with the full dataset except for the initial point. Thus, we assume that output solutions would be similar with and without the first pass while validating the assumption empirically over benchmarks. Our insight here is that the randomly sampled dataset would represent the distribution of the full dataset [79]. However, the sample size of our search algorithm is a hyperparameter, and we set the size of datasets ranging from 10% to 50% over benchmarks, not to degrade the performance of the second pass. The overly-sampled dataset would incur a significant increase in the number of tuning iterations to recover the inference accuracy, which amortizes the single iteration speedup of sampling. We quantitively evaluated the speedup

Framework	Network	Dataset	Metric
Caffe	MLP-M [80]	MNIST [81]	Accuracy
	MLP-C [82]	CIFAR-10 [57]	Accuracy
	LeNet [83]	MNIST [81]	Accuracy
	Cuda-convnet [56]	CIFAR-10 [57]	Accuracy
	AlexNet [84]	ImageNet [85]	Accuracy
	NiN [86]	ImageNet [85]	Accuracy
	VGG-16 [87]	ImageNet [85]	Accuracy
	GoogleNet [70]	ImageNet [85]	Accuracy
	ResNet-50 [69]	ImageNet [85]	Accuracy
DarkNet	DarkNet-ref [64]	ImageNet [85]	Accuracy
	Tiny YOLO [88]	VOC2007 [89]	mAP

Table III.3: Network models for evaluation

of the two-pass algorithm and discussed its benefits on reducing search costs in Section III.4.

III.4. Evaluation

III.4.A. Methodology

To evaluate *libnumber*, we use nine CNN and two MLP models as summarized in Table III.3. We port two popular DNN frameworks to *libnumber* to demonstrate its portability: Caffe [90] and DarkNet [64]. For Tiny YOLO we use a mean-average precision (mAP) metric for quantifying object detection accuracy; for the others image classification accuracy. Following the methodology of Stanford DAWNBench [91], we target 93% relative accuracy of the full-precision 32-bit floating-point type (i.e., `err_margin` = 0.07) by default. We have also tested 99% accuracy to observe similar results, which are omitted due to limited space. As discussed in Section II.1, we quantize a pre-trained model for the given accuracy constraint [35, 58, 92]. If necessary,

one can recover the accuracy by re-training the quantized model.

We quantize both weights and activations of convolution layers (for CNNs) and fully-connected layers (for MLPs), but layer-wise optimization is applied to weights only. Otherwise, we would have to pay the cost of format conversion across layers, which may nullify performance benefits. We use the default objective function (f_{obj}) for the auto-tuner, which minimizes the total number of bits for both weights and activations, and hence storage requirements as well as computational strength.

We compare the quantization quality of *libnumber* against two well-known frameworks: Ristretto [93] and the Stripes quantizer [71]. We have faithfully reproduced both algorithms and validated them so that they perform at least comparably to or better than the reported results under the same constraints. Note that Stripes quantizes activations only and uses the `FIXED(16)` type for weights. For fair comparison we also run *libnumber* under the same constraints and compare speedups on a Stripes-like bit-serial hardware.

Our target hardware is a Xilinx Kintex UltraScale KU115 FPGA platform [94]. The FPGA can flexibly accommodate a variety of number formats featuring a large search space, to make it an ideal test vehicle for our work. For performance evaluation we use Xilinx Vivado High-Level Synthesis (HLS) tool Version 2017.4. The HLS tool compiles a C++ kernel to generate FPGA bitstream, from which we can calculate the cycle count and resource utilization of the kernel. The baseline implementation of HLS-generated kernels is optimized following the methodology of Zhang et al. [95] to achieve competitive performance to theirs. Once the best configuration is found by a quantizer, we manually apply a transformation to the original kernel, to re-

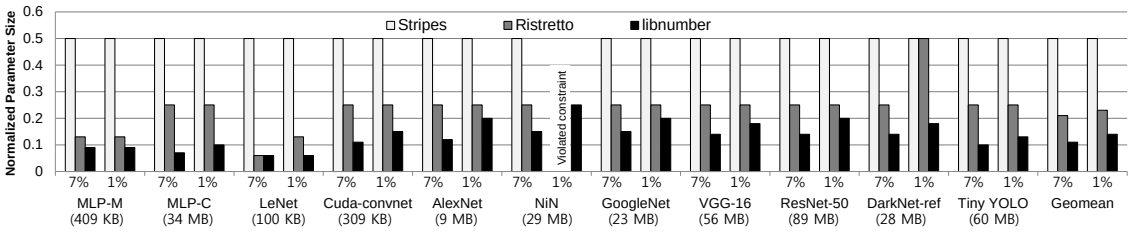


Figure III.5: Model parameter size normalized to the `FLOAT(32)` baseline with 7% and 1% of accuracy tolerance. Lower is better. Network name is annotated with the baseline parameter size.

place the `Number` ADT with a concrete, quantized type. Then we re-optimize the tiling factors and apply a bit-packing optimization, which packs multiple weights into a single word for SIMD-style execution and memory bandwidth reduction. Note that this is a new optimization enabled by quantization, and cannot be applied to the baseline.

III.4.B. Results

Search Quality. Figure III.5 compares the size of network parameters at the best configuration for 7% and 1% error tolerance. The results are normalized to the 32-bit floating-point baseline as in [93]. *libnumber* reduces the parameter size by 8.91× and 7.13× on average and total storage requirements (including activations) by 8.28× and 6.44× for 7% and 1% tolerance, respectively. These numbers translate to 69.6% (38.1%) reduction of storage space over Stripes (Ristretto) for 7% tolerance. Note that Stripes has a constant size of 16 bits as its weight format is fixed to `FIXED(16)` [71]. Both Ristretto and Stripes fail to tune NiN within 1% accuracy loss as NiN

Network	Framework	Per Layer Format (Act/Weights)
MLP-M	*libnumber*	I3 / E3-E2-E3
	Ristretto	I2 / I4-I4-I4
MLP-C	*libnumber*	I4 / I2-E3-I3-I4
	Ristretto	I2 / I8-I8-I8-I8
LeNet	*libnumber*	I2 / E2-E2
	Ristretto	I2 / I2-I2
Cuda-convnet	*libnumber*	I3 / I4-E3-I4
	Ristretto	I2 / I8-I8-I8
AlexNet	*libnumber*	I5 / I5-I6-E4-E3-E4
	Ristretto	I4 / I8-I8-I8-I8-I8
NiN	*libnumber*	I6 / I4-I4-I6-E4-I5-I5-I6-I4-I6-E4-I5-I6
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8
VGG-16	*libnumber*	I8 / I5-I3-I7-I5-I5-I4-E4-I4-I4-I5-I6-I4-I3
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8
GoogleNet	*libnumber*	I6 / I6-I6-I5-I6-I4-I6-I5-I5-I5-I5-I4
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8
ResNet-50	*libnumber*	I7 / I6-I7-I5-I6-I6-I4-I5-I5-I5-I5-I4-I4-I5-I6-I5-I3-I4
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8
DarkNet	*libnumber*	I7 / I8-I7-I8-I6-I6-I6-E4-I5
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8
Tiny YOLO	*libnumber*	I8 / I6-I6-I7-I6-I5-I5-I3-E3-I5
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8

Table III.4: Best configuration: In (`FIXED`), En (`EXP`)

requires careful bias tuning when using fixed-point numbers.[1] We find the quality of bias tuning can significantly affect the overall compression ratio.

Table III.4 shows the best configurations found by both *libnumber* and Ristretto. *libnumber* yields more compact representation to reduce total storage requirements by up to 71.3% for MLP-C with an average of 38.1%. This performance gap is attributed to the difference in the coverage of the configuration space. Ristretto has a much narrower format coverage as it considers only `FIXED(`2^n`)` formats and does not perform layer-wise optimization. For example, the search space of *libnumber* has 1.65×10^{29} configurations for

[1] Stripes [71] reports successful quantization of NiN within 1% accuracy loss, but their baseline (`FIXED(16)`) is different from ours (`FLOAT(32)`) having a higher baseline accuracy.

Network	# of target layers	# of iterations		
		Exhaustive	*libnumber*	Ratio
MLP-M	3	3.11×10^{6}	190	6.11×10^{-5}
MLP-C	4	1.31×10^{8}	359	2.75×10^{-6}
LeNet	2	7.41×10^{4}	74	9.99×10^{-4}
Cuda-convnet	3	3.11×10^{6}	208	6.68×10^{-5}
AlexNet	5	5.49×10^{9}	419	7.63×10^{-8}
NiN	12	1.27×10^{21}	2759	2.18×10^{-18}
VGG-16	13	5.31×10^{22}	3516	6.62×10^{-20}
GoogleNet	11	3.01×10^{19}	1783	5.92×10^{-17}
ResNet-50	17	1.65×10^{29}	5471	3.31×10^{-26}
DarkNet-ref	8	4.07×10^{14}	1130	2.78×10^{-12}
Tiny YOLO	9	1.71×10^{16}	1644	9.63×10^{-14}

Table III.5: Comparison of search costs

ResNet-50, whereas Ristretto has only 6. Even if we relax the constraints of Ristretto to consider all integers (i.e., `FIXED(n)` with $2 \leq n \leq 32$), instead of `FIXED(`2^n`)`, the parameter compression ratio is still significantly lower than *libnumber* (6.35× vs. 8.91×).

Search Cost. Another important metric is the cost of search. Table III.5 summarizes this cost for *libnumber* in terms of the number of iterations (i.e., tested configurations), which ranges from 74 to 5471, even if the search space can be as large as 1.65×10^{29}. The auto-tuning algorithm navigates through only a tiny fraction of the search space to find best configuration. Besides, the two-pass algorithm, using both small set and full set, effectively reduces the average cost per iteration. Using the small set yields a maximum speedup of 4.48× for Cuda-convnet with an average speedup of 2.52× over a version of *libnumber* using the full set only.

There is a tradeoff between search cost and quality. For example, one may reduce search time by using the same configuration for replicated blocks of layers (e.g., inception modules, residual blocks) for models such as ResNet

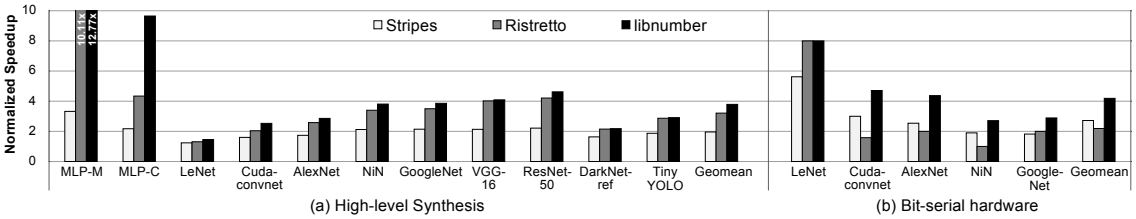


Figure III.6: Normalized speedups using (a) High-level Synthesis (HLS) for FPGA and (b) bit-serial hardware

and GoogleNet. This can be easily done in *libnumber* by having corresponding layers to share the same layer name. However, it increases the total parameter size by 82% and 22% over *libnumber* for ResNet-50 and GoogleNet, respectively. Note that this tradeoff can be easily explored using *libnumber*.

Speedups on FPGA. Figure III.6(a) shows the speedups for the target layers of 11 DNNs, synthesized by HLS for an FPGA. *libnumber* achieves a 3.79× geomean speedup over the baseline `FLOAT(32)` version, which translates to 93% and 18% speedups over Stripes and Ristretto, respectively. By representing numbers with fewer bits for each layer, *libnumber* significantly improves resource utilization. In particular, this enables more weights and activations to be packed in a single 32-bit word, leading to fewer bank accesses for a given amount of computation. Besides, reduced bit widths allow us to increase parallelism and execute more operations in a SIMD style. The relative speedup of *libnumber* is more pronounced for Stripes than Ristretto. It is because Ristretto often uses just a few more bits per layer than *libnumber* to end up packing the same number of elements per 32-bit word. In all cases MLPs demonstrate greater speedups than CNNs as they have more parallelism for not using convolutions.

Ideal Speedups on Bit-serial Hardware. The benefits of quantization can be best realized by bit-serial hardware, whose latency for a MAC operation scales (almost) linearly to the number of bits [71, 96]. We use a very simple analytical performance model to estimate ideal performance as in [71]. We assume a DNN accelerator like Stripes [71], which is based on DaDianNao [97] but serialized. Assuming the same constraints as Stripes (i.e., 16-bit fixed-point baseline, quantizing activations only using dynamic fixed-point type, 1% accuracy tolerance), we run *libnumber* to quantize the target layers. The corresponding bit widths for Stripes are taken from the original paper, instead of using our version of the Stripes quantizer. Assuming 100% PE utilization and 1-cycle latency for a 1-bit serial operation, the execution time of a kernel on DaDianNao is estimated to be the total GOPs of the MAC operations for the layer divided by the total number of PEs (in ops/cycle). We also assume a linear scaling of the throughput to reduction in bit width (i.e., speedup is $16/p$ if p is the bit width). Note that Stripes achieves speedups within 2% of the ideal speedups calculated similarly [71].

Figure III.6(b) shows ideal speedups of Stripes, Ristretto, and *libnumber* over DaDianNao. We only use a subset of 5 DNN models that are also used for evaluating Stripes [71]. *libnumber* achieves a geomean speedup of 4.19×, which outperforms both Ristretto and the Stripes quantizer. In this setup, a reduction in bit width leads to performance boost more directly than FPGA, to demonstrate greater performance gap between *libnumber* and the other two quantizers.

Portability and Programmer Effort. Finally, we estimate the programmer's porting effort to *libnumber* by counting the number of modified lines of code

(LoC). As we target an FPGA device, we use as baseline a sequential C++ version of the convolution kernel, which takes 696 (540) LoC for Caffe (DarkNet). The number of added/modified/deleted lines for porting is 64 and 57 for Caffe and DarkNet, respectively. Overall, porting takes only modest effort as the modified LoC is just a small fraction of the total LoC of the framework, which is 63,733 (25,144) for Caffe (DarkNet).

III.5. Related Work

Quantization for DNNs. There are proposals to reduce the precision of data to improve performance of pre-trained DNNs with little degradation of accuracy [24, 28, 29, 33, 59, 63, 98, 99]. However, they have limited coverage by considering only one type of numbers [28, 29, 33] or supporting mixed types in a very limited way [59, 100]. Commodity-level quantization toolchains such as TensorFlow Lite [34] or TVM [101] and some advanced quantization schemes utilizing differentiable nerual architecture search [77, 78] also significantly restrict the available number types into some set of fixed-point variants. Deep compression [58] uses a value clustering technique to identify representative values. However, these techniques suffer from irregular memory accesses and inefficient computation due to an extensive use of high-precision values. In contrast, *libnumber* selects an suitable representation for each layer by considering both number type and bit width to effectively eliminate redundancy in numbers. Zhou et al.[35] propose a quantization technique that utilizes multiple number types, and demonstrate savings in bit count. However, they have much narrower type coverage by using zero

and exponent types only, which can be problematic for complex networks. Besides, their work lacks evaluation on a realistic hardware platform.

Tuning Algorithms and Frameworks. Even before emergence of DNNs auto-tuning has been investigated in the research community for a long time, for compiler optimization [72, 102, 103, 104, 105], runtime parallelism adaptation [73, 74], and so on. While these frameworks may be used for tuning DNN workloads, their efficiency will be much lower without considering DNNs' algorithmic characteristics. In contrast, *libnumber* employs an efficient tuning algorithm customized for quantizing DNNs.

IV. Layerweaver: Layer-wise Time-multiplexing DNN Scheduler

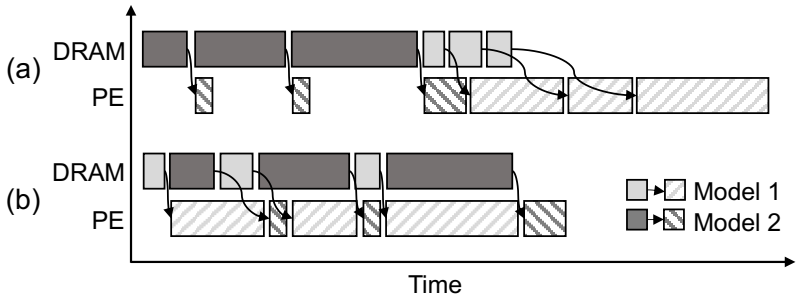


Figure IV.1: A timeline with different scheduling. Based on decoupled memory system, (a) illustrates the schedule without reordering and (b) with reordering.

IV.1. Overview

The analysis in Chapter II implies that (i) either PE or DRAM bandwidth resources (or both) may be under-utilized due to a mismatch between the arithmetic intensity of a DNN model and the compute-to-memory bandwidth ratio of the NPU; (ii) the imbalance of resource usage cannot be eliminated by simply adjusting the batch size of a single DNN model. Thus, we propose to execute two different DNN models with opposite characteristics via layer-wise time-multiplexing to balance the resource usage for a given NPU.

Figure IV.1 illustrates how interweaving the layers of two heterogeneous DNN models can lead to balanced resource usage. In this setup we assume that the DNN serving system needs to execute two hypothetical 3-layer mod-

els, a compute-intensive one (Model 1) and a memory-intensive one (Model 2). The baseline schedule in Figure IV.1(a) does not allow reordering between layers. During the execution of Model 2, the PE time is under-utilized, whereas during the execution of Model 1, the DRAM time is under-utilized. However, by reordering layers across the two models appropriately as in Figure IV.1(b), the two models as a whole utilize both PE and DRAM bandwidth resources in a much more balanced manner.

Challenges for Efficient Layer-wise Scheduling. Determining an efficient schedule to fully utilize both compute and memory resources is not a simple task. It would have been very easy if the on-chip buffer had an infinite size. In such a case, simply prioritizing layers having longer compute time than memory time would be sufficient to maximize compute utilization as the whole memory time will be completely hidden by computation. However, unfortunately, the on-chip buffer size is very limited, and thus one needs to carefully consider the prefetched data size as well as the remaining on-chip buffer size. This is because prefetching too early incurs memory idle time as described in the "Memory Idle Time" paragraph in Section IV.4.

Layerweaver. Layerweaver is an inference serving system with a layer-wise weaving scheduler for NPU. The core idea of Layerweaver is to interweave multiple DNN models of opposite characteristics, thereby abating processing elements (PEs) and DRAM bandwidth idle time. Figure IV.2 represents the overall architecture of Layerweaver. The design goal of Layerweaver is to find a layer-wise schedule of execution that can finish all necessary computations for a given set of requests in the shortest time possible.

Deployment. Layerweaver is comprised of a request queue, scheduler, and

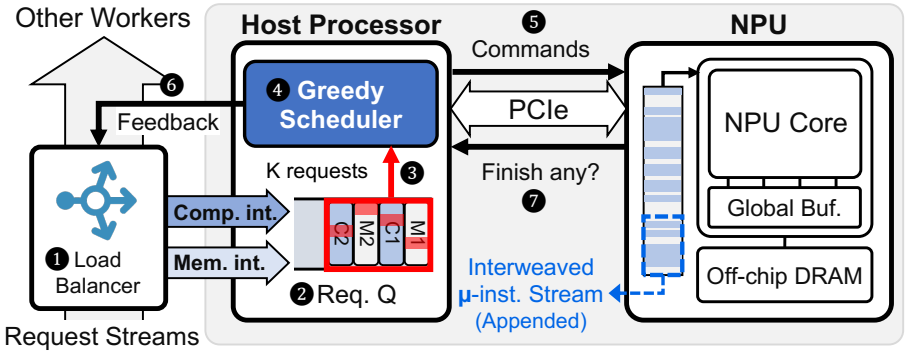


Figure IV.2: NPU-incorporated serving system with Layerweaver.

NPU hardware for inference computing. It could be often integrated with a cloud load balancer that ❶ directs the proper amount of inference queries for each compute- and memory-intensive models to a particular NPU instance [18, 106, 107, 108]. Such load-balancers are important in existing systems as well since supplying an excessive number of queries to a single NPU instance can result in an unacceptable latency explosion. ❷ Once request queues of Layerweaver accepts the set of requests for each model, ❸ the host processor dequeues a certain number of requests from each queue and pass them to the scheduler. And then, ❹ the host processor invokes Layerweaver scheduler and performs the scheduling. ❺ Following the scheduling result, the host processor dispatches those scheduled layers to NPU by appending to activated instruction streams. ❻ Depending on the scheduling results, one of two request queues may have higher occupancy. In that case, it is reported to the cloud load balancer so that the load balancer can utilize this information for the future load distribution. Finally, ❼ NPU executes those instructions, and once it finishes handling a single batch of requests, it returns the result to the host processor. Note that Layerweaver employs

a greedy scheduler (see the next section), and thus only needs information about the one next layer for each model.

Baseline NPU. There exist many different types of NPUs with their own unique architecture. However, Layerweaver is not really dependent on the specific NPU architecture. Throughout the paper, we assume a generic NPU that resembles many of the popular commercial/academic NPUs such as Google Cloud TPUv3 [19] or Intel NNP-I [20]. This generic NPU consists of compute units (PEs), and two shared on-chip scratchpad memory buffers, one for weights and one for activations. The host controls this NPU by issuing a stream of commands such as fetching data to on-chip memory or performing computation. Once the NPU finishes computation, it automatically frees the consumed weights, and stores the outcome at the specified location of the activation buffer. One important aspect is that the NPU processes computation and main memory accesses in a decoupled fashion. The NPU eagerly processes fetch commands from the host by performing weight transfers from its main memory to the on-chip weight buffer as long as the buffer has empty space. In a similar way, its processing unit eagerly processes computation commands from the host as long as its inputs (i.e., weights) are ready.

Supporting the layer-wise interweaving in this baseline NPU does not require an extension. In fact, the NPU does not even need to be aware that it is running layers from different models. The host can run Layerweaver scheduler to obtain the effective layer-wise interweaved schedule, and then simply issue commands corresponding to the obtained schedule. Unfortunately, we find that existing commercial NPUs do not yet expose the low-level API that enables the end-user to directly control the NPU's scratchpad

memory. However, it is reasonable to assume that such APIs are internally available [19, 23, 109]. In this case, we believe that the developer can readily utilize Layerweaver on the target NPU.

IV.2. Greedy Scheduler

The main challenge of finding an optimal layer-wise scheduling is the enormous size of the scheduling search space. Brute-force approach naturally leads to a burst of computation cost. For example, to find the optimal schedule for a model set consisted of ResNet50 and BERT-base, which has 53 and 75 layers respectively, ${}_{128}\mathrm{C}_{53}$ ($\simeq 4 \times 10^{36}$) candidate schedules should be investigated, which is not feasible. Note that the previous work [110] uses simplified heuristics to manage the high search cost, while Layerweaver presents a way to formally calculate the exact idle time of each resource and maximize the total resource utilization.

To determine suitable execution order of layers for k different models within a practically short time, the scheduler adopts a greedy layer selection approach. It estimates computation and memory idle time incurred by each candidate layer then selects a layer showing the least idle time as the next scheduled layer. Here, the algorithm maintains a *candidate group* and only considers layers in the group to be scheduled next. For one model, a layer belongs to the group if and only if it is the first unscheduled layer of the model. Assuming that an inspection of a single potential candidate layer takes $O(1)$ time, the complexity for making a single scheduling decision is $O(k)$. Assuming that this process is repeated for N layers, the overall complexity

```
1  def Schedule(M_0, ..., M_{k-1}):
2    totalSteps = sum_{i=0,...,k-1}(len(M_i))
3    curSchedule = [ ]
4    indexWindow = [0, ..., 0] # length k
5    schedState = [t_m : 0, t_c : 0, l :[ ]]
6      for step in range(totalSteps):
7        # checks for pause
8        for i in range(k):
9          CheckEnd(indexWindow[i], M_i)
10       # one scheduling step
11       candidateGroup = [M_i[idx] for (i, idx) in enumerate(indexWindow)]
12       stateList = [ ]
13       for modelNum in range(k):
14         # schedule state update
15         newSchedState = UpdateSchedule(schedState, candidateGroup[modelNum])
16         stateList.append(newSchedState)
17       # layer selection
18       schedState, selectedIdx = Select(schedState, stateList, candidateGroup)
19       curSchedule.append(candidateGroup[selectedIdx])
20       indexWindow[selectedIdx]++
21   return curSchedule
```

Figure IV.3: Greedy layer scheduling algorithm.

is $O(kN)$ for N layers.

Profiling. Before launching the scheduler, Layerweaver requires to profile a DNN model to identify computation time, memory usage, and execution order of each layer. This profiling stage simply performs a few inference operations and then records information for each layer L of the model. Specifically, it records the computation time `L[comp]`, and the number of weights that this layer needs to fetch from the memory `L[size]`. Finally, such pairs are stored in a list M following the original execution order. Since most NPUs have a deterministic performance characteristic, offline profiling is sufficient.

Greedy Scheduler. Figure IV.3 shows the working process of the scheduler. We assumed NPU running Layerweaver has `BufSize` sized on-chip buffer and `MemBW` off-chip memory bandwidth. It maintains three auxiliary data

structures during its run. The algorithm selects one layer from the candidate group and append it to the end of `curSchedule` every step, and this data structure is the outcome of Layerweaver once the algorithm completes. `indexWindow` represents the indexes of layers in the candidate group of each model to track the layer execution progress correctly. Lastly, `schedState` keeps several information representing the current schedule.

For each step the algorithm determines the next layer to be scheduled among candidates. First, the algorithm constructs `candidateGroup` from `indexWindow`. Then, for each candidate layer `UpdateSchedule` function computes how the NPU state changes if a candidate layer is scheduled as described in Section IV.3. Updated schedule states are stored in `stateList`. Next, `Select` function examines all schedule states in `stateList` and estimates the idle time of each updated schedule state, which will be further elaborated in Section IV.4. It selects the layer having the shortest idle time as the next scheduled layer, which is appended to `curSchedule`.

At the beginning of every step, `CheckEnd` function checks `indexWindow[]` to see if scheduling of any model is completed (i.e., all of its layers are scheduled). If so, the scheduler is paused and scheduled layers up to this point (as recorded in `curSchedule`) are dispatched to NPU. Just before the completion of execution, the scheduler is awoken and continues from where it left off. `CheckEnd` then probes the request queue in search of any remaining workload. If there is nothing left, the scheduler is terminated. If there are additional requests queued, they are appended to existing requests of the same model, if any. And then the scheduler resumes.

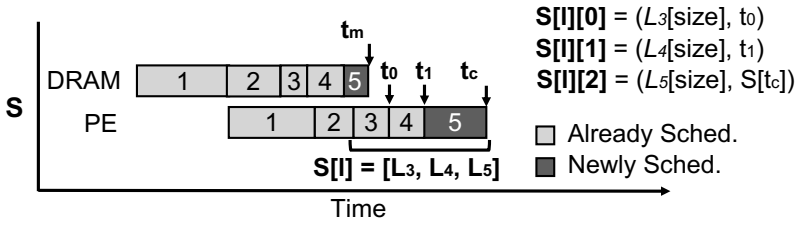


Figure IV.4: Schedule state of an example schedule `S`.

IV.3. Maintaining and Updating Schedule State

This section provides intricate detail of `UpdateSchedule` (Line 15 in Figure IV.3). The function explores the effect of scheduling a layer in the `candidateGroup` on the overall schedule and pass the information to `select`.

Concept of Schedule State. Layerweaver maintains the state for a specific schedule `S`. This state consists of three elements: compute completion timestamp $\texttt{S[t}_\texttt{c}\texttt{]}$, communication completion timestamp $\texttt{S[t}_\texttt{m}\texttt{]}$, and a list `S[l]` containing information about already scheduled layers that are expected to finish in a time interval ($\texttt{S[t}_\texttt{m}\texttt{]}$, $\texttt{S[t}_\texttt{c}\texttt{]}$]. Specifically, the list `S[l]` consists of pairs where each entry (`S[l][j].size`, `S[l][j].completion`) represents the on-chip memory usage and the completion time of the `j`th layer in the list, respectively. Figure IV.4 illustrates an example schedule and elements composing its state. Below, we discuss how scheduling a specific layer L changes the each element of the schedule state.

Scheduling Memory Fetch. Figure IV.5 shows the process of scheduling the memory fetch represented in pseudocode. If the layer L's memory size `L[size]` is smaller than the amount of available on-chip memory at time

```
1  # called by UpdateSchedule
2  def ScheduleMemFetch(S, L):
3    sizeToFetch = L[size]
4    remainingBuf = BufSize − ∑_j S[l][j].size
5    curTime = S[t_m]
6    if sizeToFetch <= remainingBuf:
7      return curTime + sizeToFetch / MemBW
8    else:
9      curTime = curTime + remainingBuf / MemBW
10     sizeToFetch −= remainingBuf
11     for j in range(len(S[l])):
12       if curTime < S[l][j].completion:
13         curTime = S[l][j].completion
14       if sizeToFetch < S[l][j]. size:
15         return curTime + sizeToFetch / MemBW
16       else:
17         sizeToFetch −= S[l][j].size
18         curTime = curTime + S[l][j].size / MemBW
```

Figure IV.5: Scheduling Memory Fetch for Layer L on Schedule S. `BufSize` represents the on-chip buffer capacity, and `MemBW` represents the system's memory bandwidth.

`S[t_m]`, the memory fetch can simply start at `S[t_m]` and finish at `S[t_m] + L[size]/MemBW` (Line 7). This case is shown in Figure IV.6(a). However, if the amount of available on-chip memory at time `S[t_m]` is not sufficient, this becomes trickier, as shown in Figure IV.6(b). First, a portion of `L[size]` that fits the currently remaining on-chip memory capacity is scheduled (Line 9). Then, the remaining amount (i.e., `sizeToFetch` in Line 10) is scheduled when the computation for a layer in list `S[l]` completes and frees the on-chip memory. For this purpose, the code iterates over list `S[l]`. For each iteration, the code checks if the jth layer in the list has completed by the time that previous fetch has completed (Line 12). If so, it immediately schedules a fetch for the freed amount (Line 14-18). If not, it waits until this layer completes and then schedules a fetch (Line 16-18). This process is repeated until `L[size]` amount of data is fetched. The returned value of

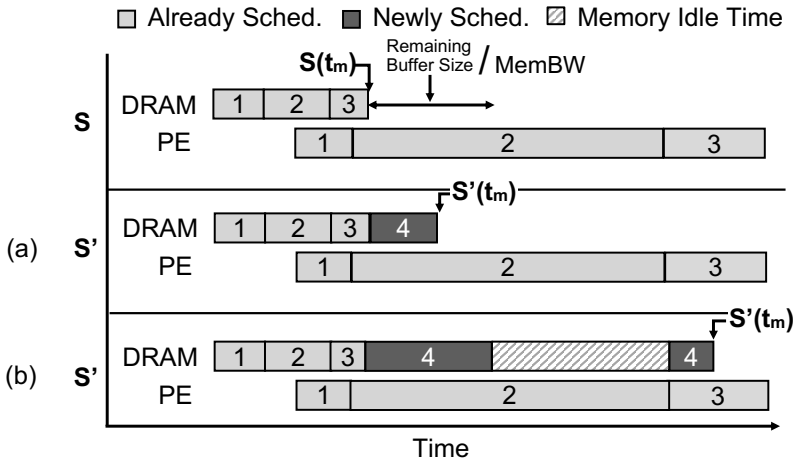


Figure IV.6: Visualization of memory fetch scheduling.

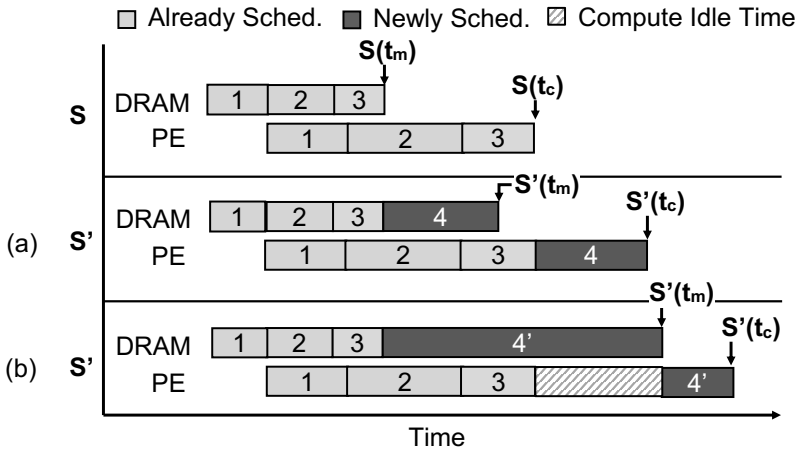


Figure IV.7: Visualization of computation scheduling.

the algorithm is $\mathtt{S'[t_m]}$.

Scheduling Computation. Once the memory fetch ends, the computation for layer L can be scheduled. Here, there are two different cases. In the first case (Figure IV.7(a)), the previous schedule's computation has not ended by the time that memory fetch completes (i.e., $\mathtt{S[t_c]} > \mathtt{S'[t_m]}$). In this case, computation is scheduled on time $\mathtt{S[t_c]}$ and completes at $\mathtt{S[t_c]}$ +

L[comp]. On the other hand, if the previous schedule's computation has ended before the time that memory fetch for the current layer completes (i.e., S[t_c] < S'[t_m] as shown in Figure IV.7(b)), the compute needs to start on time S'[t_m] (since it is dependent on the fetched memory), and completes at S'[t_m] + L[comp]. The following equation summarizes the process of obtaining S'[t_c].

$$S'[t_c] = \max(S[t_c], S'[t_m]) + L[comp]$$

Updating S[l]. Finally, S[l] needs to be updated accordingly. To obtain S'[l], all layer j in S[l] whose completion time is before S'[t_m] are excluded. Then, the current layer L is appended to S[l].

IV.4. Layer Selection to Minimize Resource Idle Time

The goal of Layerweaver scheduler is clear: maximize both the PE utilization, and the DRAM bandwidth utilization (i.e., bandwidth utilization of off-chip memory links). To effectively achieve this goal, the scheduler should execute *the layer that incurs the shortest compute or memory idle time*. Here, we explain how Layerweaver estimates the amount of compute or memory idle time incurred if layer L is scheduled following the current schedule S by Select.

Decoupling Distance. For a schedule S, its decoupling distance is defined as S[t_c] - S[t_m]. And maintaining an appropriate decoupling distance is important. If this distance is too large, it means that the prefetch is happening far before the fetched values are actually used resulting in memory idle time

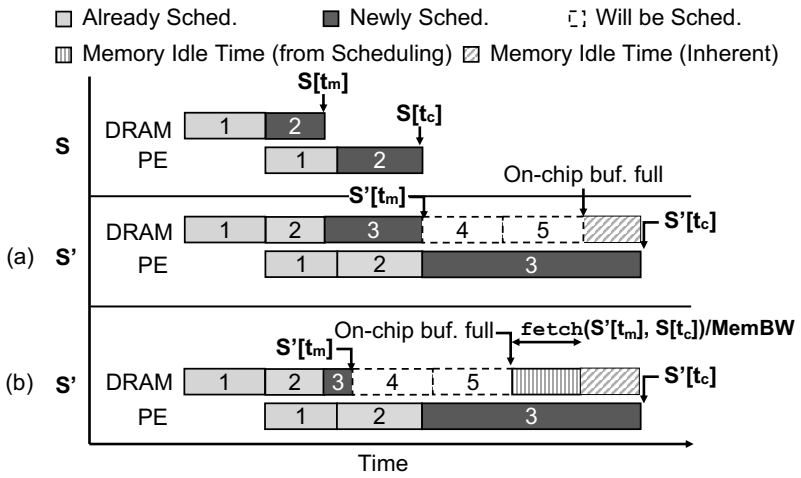


Figure IV.8: Illustration of the memory idle time.

as on-chip buffers have a finite size (Figure IV.6(b)). On the other hand, if this value is too small, it means that the prefetch is happening right before the fetched values are used, leading to compute idle time (Figure IV.7(b)). In what follows we cover both cases in greater details.

Compute Idle Time. This occurs when the decoupling distance (i.e., `S[`t_c`] - S[`t_m`]`) is smaller than `L[size] / MemBW`. In this case, memory fetch cannot finish within the decoupling distance, and PEs should wait until the memory fetch is completed (Figure IV.7(b)). The idle time can be computed as follows.

$$\texttt{L[size] / MemBW} - (\texttt{S[}t_c\texttt{]} - \texttt{S[}t_m\texttt{]})$$

Memory Idle Time. Identifying the amount of memory idle time incurred from a scheduling decision is trickier. This is because the timeline of the current schedule does not actually show the memory idle time. Figure IV.8(a) shows an example case illustrating a scheduling decision that incurs large

memory idle time. In this case, a layer L with the very large compute time (i.e., `L[comp]`) is scheduled. Unfortunately, the resulting decoupling distance is so large and exceeds the amount of time it takes to completely fill the on-chip buffer with the assumed system memory bandwidth. In such a case, regardless of a layer scheduled following the current layer, the hatched area of the timeline (i.e., `L[comp] - (BufSize - L[size]) / MemBW`) remains as memory idle time. However, one should note that this is **not** the result of the scheduling decision. Rather, this is a layer's inherent characteristic because such a memory idle time occurs regardless of the point that this layer is scheduled.

Still, this does not mean that one can schedule such a layer anywhere without any implication. Figure IV.8(b) shows a more general case where $S'[t_m]$ is not equal to $S[t_c]$. In this case, the same memory idle time exists. However, the situation is worse here, because more fetch operations (for the next layers) will be scheduled in a time interval ($S'[t_m]$, $S[t_c]$). By the time $S[t_c]$, the amount of available on-chip buffer may be much lower than that of Figure IV.8(a) (i.e., `BufSize - L[size]`). Say that the amount of memory fetch operations that will be scheduled in a time interval ($S'[t_m]$, $S[t_c]$) is `fetch(`$S'[t_m]$`, `$S[t_c]$`)`. `fetch(`$S'[t_m]$`, `$S[t_c]$`)` can be computed by inspecting `S'[l]` and such a process is similar to Line 10-17 in Figure IV.5. In this case, the remaining on-chip buffer at time $S[t_c]$ is `BufSize - L[size]-fetch(`$S'[t_m]$`, `$S[t_c]$`)`. As a result, the resulting memory idle time is `L[comp] - (BufSize - (L[size] + fetch(`$S'[t_m]$`, `$S[t_c]$`))) / MemBW`. However, note that `L[comp] - (BufSize - L[size]) / MemBW`) is not the result of the scheduling decision. The additional amount

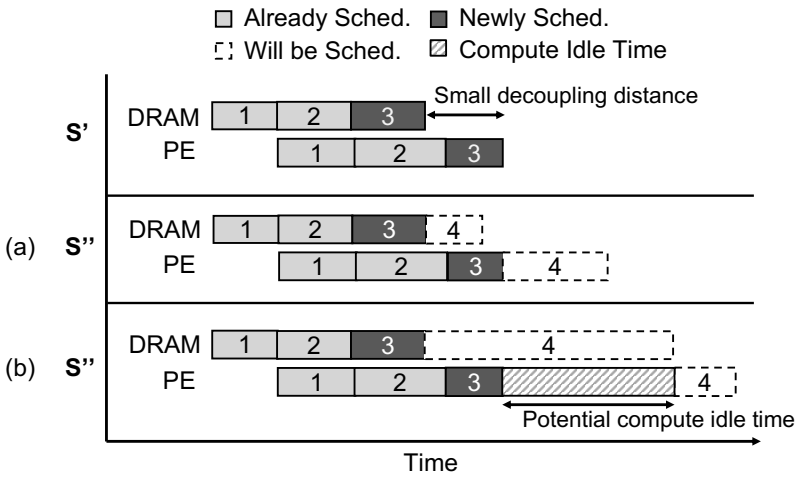


Figure IV.9: Illustration of the potential compute idle time.

of memory idle time resulting from the scheduling decision is as follows.

$$\texttt{fetch(S'[t}_\texttt{m}\texttt{], S[t}_\texttt{c}\texttt{])} \;/\; \texttt{MemBW}$$

Potential Compute Idle Time. There is an additional implication of a scheduling decision. A scheduling decision can potentially incur a compute idle time in the future, depending on the next layer that is scheduled. Figure IV.9 illustrates this case. As shown in the figure, the scheduling of a layer L resulted in a relatively small decoupling distance. This does not incur a compute idle time when the next scheduled layer's `L[size]` is small, as shown in Figure IV.9(a). However, if candidate layers for the next scheduling step have large memory time, it ends up occurring a compute idle time, as shown in Figure IV.9(b). To avoid such a potential compute idle time, it is better to maintain the decoupling distance that is at least as large as the time it takes to fetch the data for the largest layer (i.e., maximum of `L[size]` / `MemBW`

for all L in currently running models). In this case, the amount of potential compute idle time for a scheduling decision is as follows.

$$\texttt{L[size]}_{\text{max}} \texttt{ / MemBW - (S'[t}_{\text{c}}\texttt{] - S'[t}_{\text{m}}\texttt{])}$$

Layer Selection. Given a set of candidate layers to schedule, Layerweaver computes the total idle time that each scheduling decision incurs. Then, the one that incurs the minimum total idle time is selected and scheduled. In a case where multiple candidate layers incur zero total idle time, one that does not incur inherent memory idle time (i.e., `L[comp] - (BufSize - L[size]) / MemBW)` $\langle$ 0) is selected. Finally, for further tie-breaking, one with the largest decoupling distance (i.e., $\texttt{S'[t}_{\text{c}}\texttt{] - S'[t}_{\text{m}}\texttt{]}$) is selected. One drawback of this greedy policy is that it can potentially incur starvation. If a model contains a highly unbalanced layer (e.g., very large memory usage with a very little compute), this layer is likely to be never selected by our scheduler despite the model has layers with more favorable characteristics following that unbalanced layer.

Starvation Prevention. To avoid starvation, Layerweaver sometimes schedules the layer that yields a longer total idle time. Specifically, the first case is where all candidates are incurring compute idle time (excluding potential ones). This indicates that all candidate layers are memory-intensive. This is not the steady-state behavior since Layerweaver only targets scenarios where there exist at least one or more compute-intensive models. However, if a memory-intensive layer from the compute-intensive model is not scheduled, Layerweaver will continue to encounter memory-intensive candidates, and the circumstance can persist. In this case, Layerweaver selects a layer from

the compute-intensive model regardless of the exact size of compute idle time it incurs. By doing so, a candidate layer from the compute-intensive model will eventually be a compute-intensive one and effectively continue operation. Similarly, there is also a case where all candidate layers are incurring memory idle time. For a similar reason, Layerweaver selects a layer from the memory-intensive model. We find that this is sufficient to avoid starvation, considering that all other cases (e.g., some candidate layers are compute-intensive while the others are memory-intensive) cannot starve a single model.

IV.4.A. Discussion

Scheduling Cost. Our scheduler has $O(kN)$ complexity, where k is the number of models, and N is the number of layers to interweave at a single scheduler invocation. We measured the latency of our scheduling algorithm using a single core of Intel i7-7700K CPU @ 4.20GHz. For two models, the measured scheduler throughput is about 15 layers/μs. Considering that the average latency of a single layer in our evaluated workloads (w/ batch size = 1) on our evaluated NPUs (see Section IV.5.A) ranges from 4.7us to 221us, the time spent on scheduling is much smaller than the time spent on DNN models. Furthermore, such scheduling happens off-critical path most of the time using the host CPU. We also verified that the scheduling time scales linearly with the number of models and the number of layers as expected.

Scheduling Granularity. Section IV.1 explains that Layerweaver generates a schedule by interweaving layers from each model. By layer, we meant basic building blocks of neural networks such as convolutional layer and FC

(dense) layer. Both TensorFlow Keras and PyTorch list a set of supported layers in their documentation [111, 112]. However, this is just an example. In fact, the minimum scheduling granularity of Layerweaver is tied with the granularity of instruction that a target NPU supports. For example, if the host processor utilizes a single instruction for a sequence of layers (layer fusion), Layerweaver can interleave the schedule at that granularity. On the other hand, if the host utilizes finer-grained instructions (e.g., different instructions for matrix multiplication and the followed elementwise activation), Layerweaver can operate at that granularity. In general, Layerweaver can perform better with the finer-grained instructions. For platforms that only support very coarse-grained instructions, Layerweaver can be implemented in hardware by extending the NPU design. For example, AI-MT [110] utilizes hardware extensions to enable fine-grained data movements as well as computation. Doing so allows them to minimize the on-chip storage requirements for the weight buffer.

Context Switch Overhead. To minimize the context switch overhead (i.e., the overhead of executing a layer from one model then executing another layer from a different model), Layerweaver requires an activation buffer that can house activations for two models. By doing so, even when executing a layer from a different model, no extra off-chip data transfer needs to happen. Each model's activation buffer is sized to fit the largest activation size of the model for the given batch size. In our evaluation, the model that required the largest activation buffer size was MobileNetV2 (2.2MB) and ResNet50 (1.5MB) for single-batch inference. In other words, Layerweaver requires an additional 1.53MB activation buffer. The storage overhead becomes larger as

the batch size increases, but the relative overhead remains the same since the original activation buffer size increases at the same time. Table IV.1 shows the storage overhead of multi-model execution on two different evaluated NPU configurations.

Failsafe Mechanism. Layerweaver gets very limited or no benefit at all when all tasks are compute-intensive or memory-intensive at the same time. In such a case, it is impossible to achieve decent performance improvement. For example, if all workloads are compute-intensive, the scheduling decision does not really make a difference. All choices will incur memory idle time, and there will be near-zero compute idle time. For this reason, if Layerweaver detects that all provided workloads are compute-intensive or memory-intensive during the profiling run, Layerweaver is disabled.

IV.5. Evaluation

IV.5.A. Methodology

Simulation Setup. To estimate the computation cycles of NPU, we used MAESTRO [113], which analytically estimates computation cycles with various architectural parameters. We set two types of the NPU, NNP-I-like compute-centric NPU [20], and TPUv3-like memory-centric NPU [19, 114]. For dataflow we use weight stationary dataflow [115]. The detailed parameters are summarized in Table IV.1. We built a custom simulator to model the layer-wise execution behavior. We estimated computation cycles from MAESTRO. To estimate memory behavior, we calculate the data transfer

Table IV.1: NPU configuration parameters

	Compute-centric [20]	**Memory-centric** [19, 114]
Peak throughput	92 TOP/s	22.5 TOP/s
PE operating frequency	927 MHz	700 MHz
Memory BW	68 GB/s (12 ICE)	225 GB/s (1 MXU)
On-chip SRAM	48 MB (Wgt) / 2.3 MB (Act)	48 MB (Wgt) / 36 MB (Act)
	1.5 MB (Extra Act Buffer)	24 MB (Extra Act Buffer)
Common parameters		
Arithmetic precision	16 bits	
Dataflow	Weight-stationary	

time from off-chip DRAM for each layer using its tensor dimensions.

Workloads. We select four popular DNNs for each of the two workload groups: compute-intensive and memory-intensive. Thus, the total number of DNN pairs taking one from each group is 16. The four compute-intensive models are InceptionV3 (IC) [65], MobileNetV2 (MN) [66], ResNet50 (RN) [37], and ResNeXt50 (RX) [36]. For memory-intensive models, we use BERT-base (BB), BERT-large (BL) [39], NCF (NCF) [41], and XLNet (XL) [40].

IV.5.B. Evaluation Scenarios

We extend the single-stream scenario of MLPerf inference [2] benchmark to support multiple different kinds of inference requests. Layerweaver can enhance various kinds of NPUs when there is an inherent mismatch between their compute-memory capabilities and arithmetic intensities of the DNNs being served.

Schedulers. For evaluation we compare Layerweaver with three baseline schedulers as well as a (nearly) concurrent work AI-MT [110]. The three

baseline schedulers include: 1) scheduling only computation-intensive models (*Compute-only*), 2) scheduling only memory-intensive models (*Memory-only*), 3) scheduling both computation-intensive and memory-intensive models by bisecting the cycles equally and allocating each half to a specific model (*Fair*). Note that those baselines used for our evaluation substantially outperform layer-wise double buffering [42], which is used as a baseline scheme of AI-MT. We also carefully model the features of AI-MT, including memory block prefetch, compute block merging, and priority mechanism. AI-MT requires setting two user-defined thresholds, and through extensive 2D parameter sweeping we use the setting that yields the best overall throughput for our workloads. All schedulers are run on the same hardware configuration specified in Table IV.1.

Metric. We evaluate Layerweaver by measuring the system throughput (STP) [116], which is a common metric to quantify the performance of multiple workloads running on an NPU. To compute this metric, each query from a model gets the weight that is proportional to its latency in standalone execution. Then, we compare the weighted queries per second using various schedulers. For example, assume that model A takes 5ms and model B 10ms to process a single query. Suppose Scheduler 1 processes four queries of model A within 20ms and Scheduler 2 processes two queries of model B within the same interval. Then, their STPs are equal according to our metric.

Single-Batch Streams with Memory-centric NPU. This scenario reflects a case where a single NPU is asked to handle multiple tasks simultaneously while achieving the maximum throughput. In this scenario, whenever the NPU completes a query for a model, the next query for the same model is

immediately dispatched. Such a single-batch inference is popular for time series or real-time data. For such data, achieving better throughput in this scenario results in a better processing rate (e.g., frames per second, sensor frequency, etc.). Layerweaver can resolve a severe resource under-utilization problem that a memory-centric NPU experiences on single-batch workloads (Figure II.8 in Section II.3).

Multi-Batch Streams with Compute-centric NPU. This scenario is analogous to the previous scenario except that a request is batched with more than one inputs. For evaluation, we use the batch size of 16 unless specified otherwise. This scenario may correspond to a case where the NPU is required to run multiple models, and each model is invoked with inputs collected from multiple sources at a regular interval (e.g., autonomous driving, edge computing). Layerweaver can alleviate the resource under-utilization problem of a compute-centric NPU on multi-batch workloads.

IV.5.C. Results

Throughput. Figure IV.10(a) shows the system throughput (STP) over various model combinations. Layerweaver improves the system throughput on memory-centric NPU for single-batch streams (# streams = 2) by 60.1% on average (up to 75.2%) compared to the three baselines. Note that all three baseline schedulers (Section IV.5.B) have the same system throughput because they simply execute different combinations of two models with no layer-wise interweaving. Similarly, Figure IV.10(b) shows that Layerweaver improves the system throughput on compute-centric NPU for multi-batch

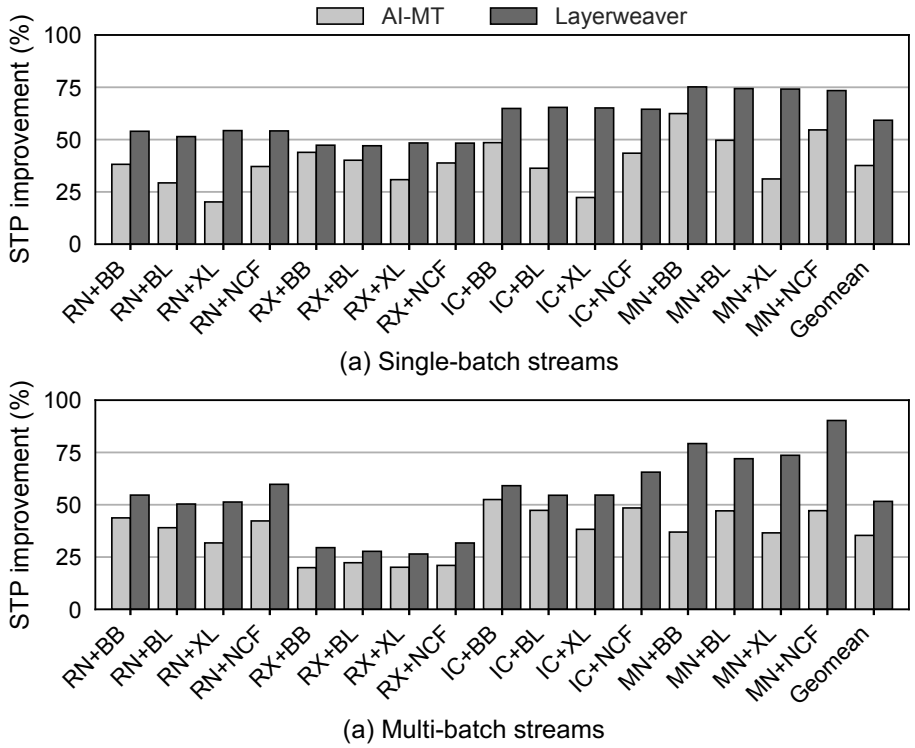


Figure IV.10: System throughput (STP) improvement on (a) a memory-centric NPU for single-batch streams and (b) a compute-centric NPU for multi-batch streams (# streams=2). Higher is better.

streams (# streams = 2) by 53.9% on average (up to 90.2%). These results translate to 21.6% and 16.3% higher geomean throughput than AI-MT for single- and multi-batch streams, respectively. (A more detailed analysis is to be presented later in this section.) Layerweaver effectively interweaves layers to achieve a much higher resource utilization, and hence substantial throughput gains.

Utilization of PE Cycles and DRAM Bandwidth. Figure IV.11(a) shows the portion of active cycles for PEs and memory on the single-batch streams scenario, and Figure IV.11(b) shows the same on the multi-batch streams

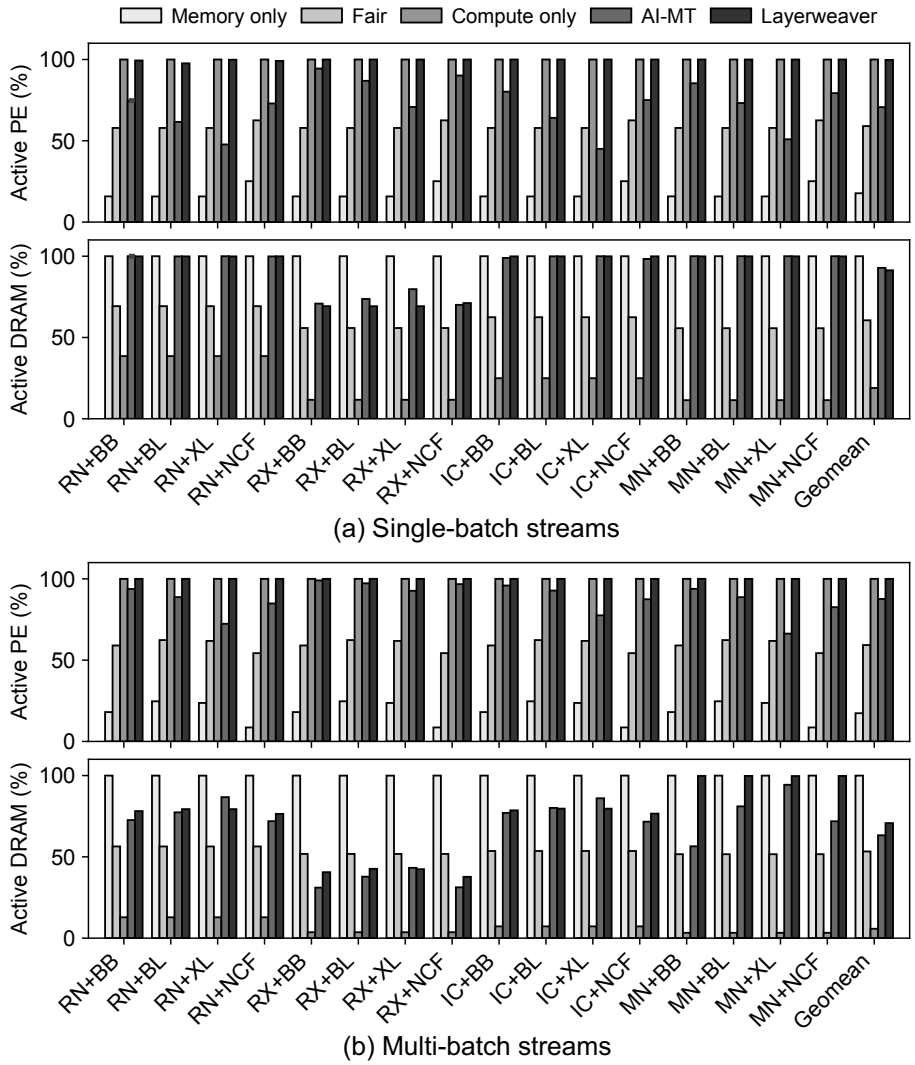


Figure IV.11: Portion of active cycles on (a) a memory-centric NPU for single-batch streams and (b) a compute-centric NPU for multi-batch streams (# streams=2).

scenario. On average, Layerweaver achieves 99.7% and 91.3% utilization of PE cycles and DRAM bandwidth, respectively, for the single-batch streams scenario, and 99.9% and 70.7% for the multi-batch streams scenario. The baseline schedulers share the same resource under-utilization problem. The

Compute-only and *Memory-only* schedulers end up with a low utilization for either DRAM bandwidth or PE cycles. Even if two DNNs are time-multiplexed at a model granularity (i.e., Fair), they end up with a mediocre level of utilization for both resources.

In contrast, Layerweaver improves the resource utilization of both PE cycles and DRAM BW by scheduling layers in a way that minimizes the resource idle time. Note that Layerweaver does not fully utilize DRAM bandwidth in some cases (e.g., combinations containing RX in Figure IV.11(a) and some combinations in Figure IV.11(b)). This is due to an inherent imbalance between compute and memory time leading to memory idle time (i.e., `L[comp] - (BufSize - L[size]) / MemBW`) as discussed with Figure IV.8 in Section IV.4. This problem can be alleviated by increasing the on-chip buffer size (`BufSize`) or adding more PEs to reduce compute time (`L[comp]`).

Impact on Single Request Latency. Figure IV.12 presents the average normalized turnaround time (ANTT) for both single- and multi-batch streams. This metric is an average of latency slowdown compared to standalone execution for each model. By executing multiple, heterogeneous requests concurrently, Layerweaver trades the latency of an individual request for higher system throughput. The ANTTs of Layerweaver are 1.27 and 1.36 for single- and multi-batch streams, respectively, normalized to the standalone execution time. This result is much more favorable than AI-MT [110], whose ANTTs are 1.55 and 1.58. Furthermore, Layerweaver has much smaller variations of ANTT than AI-MT to demonstrate more robust performance. We also checked the geomean of maximum slowdown (i.e., 100% tail) for each work-

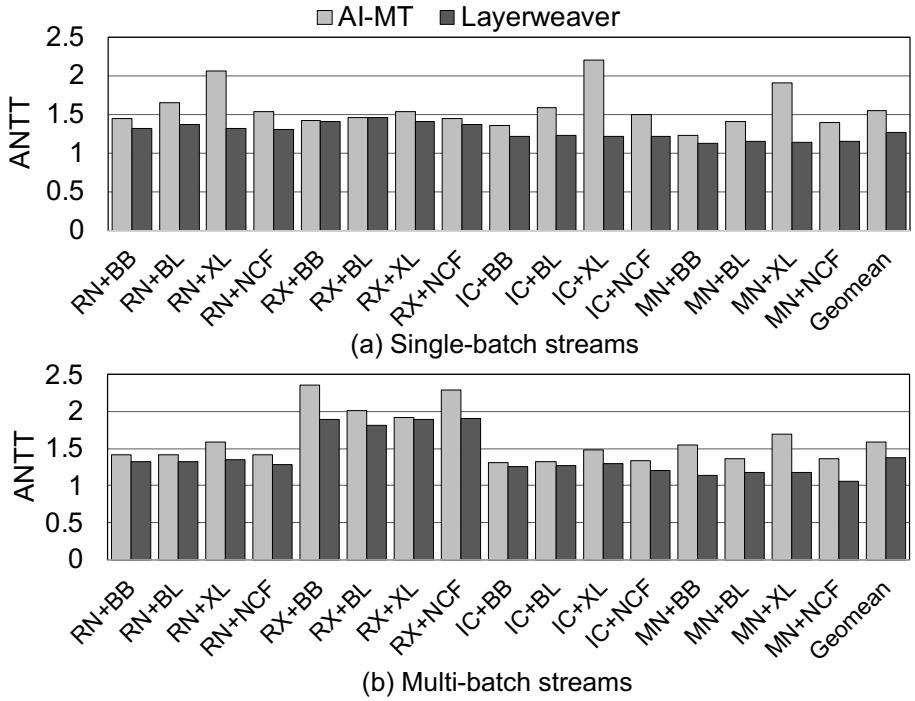


Figure IV.12: Average Normalized Turnaround Time (ANTT) with (a) single-batch streams and (b) multi-batch streams (# streams=2). Lower is better.

load on both AI-MT and Layerweaver. AI-MT exhibited 1.89× maximum slowdown and Layerweaver exhibited 1.40× maximum slowdown on single-batch streams. Similarly, AI-MT showed 1.90× maximum slowdown and Layerweaver exhibited 1.61× maximum slowdown on multi-batch streams. As expected, Layerweaver has much less impact on tail latency than AI-MT. Note that Layerweaver's near-ideal schedule that fully utilizes both DRAM bandwidth and PE still incurs a certain level of slowdown. This is inevitable in cases where a single model was already utilizing more than 50% of the resources. In such a case, an interleaving of two models can never achieve more than 2× speedup, and thus the slowdown is unavoidable.

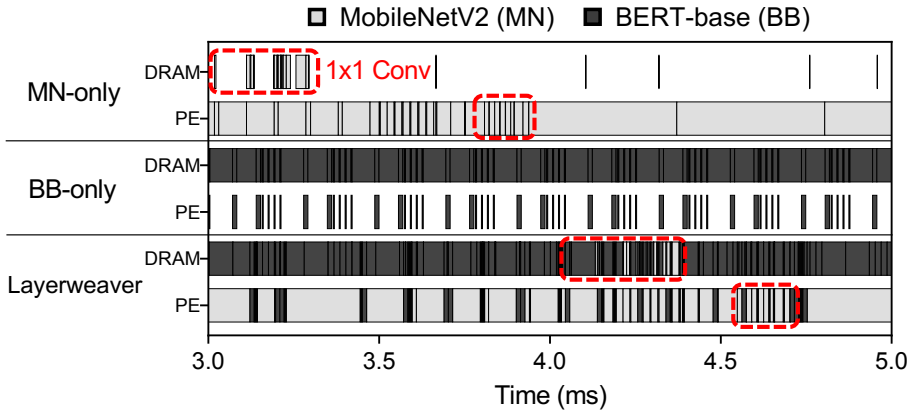


Figure IV.13: Timeline analysis on BERT-base (BB) and MobileNetV2 (MN). Multi-batch streams (# streams = 2) is run on memory-centric device.

Timeline Analysis. As a case study, we present an execution timeline visualizing the busy cycles for PEs and DRAM channels in Figure IV.13 while executing two DNN models (MN and BB). Standalone execution leads to under-utilization of either compute or off-chip DRAM bandwidth resources. However, Layerweaver can successfully balance the memory and compute resource usage to minimize the resource idle time. One interesting observation is that Layerweaver can effectively handle memory-intensive layers in a compute-intensive model. The boxed area in red in the figure corresponds to the 1×1 and grouped convolution layers with very large input channels in MN [38]. These layers are memory-intensive although the whole model is known to be compute-intensive. In such a case, our scheduler prioritizes the compute-intensive model (MN) as described in Section IV.4 so that MN can quickly move to compute-intensive layers to prevent starvation of the model and improve overall resource utilization.

In-depth Comparison to AI-MT. AI-MT [110] is a very recent work that also

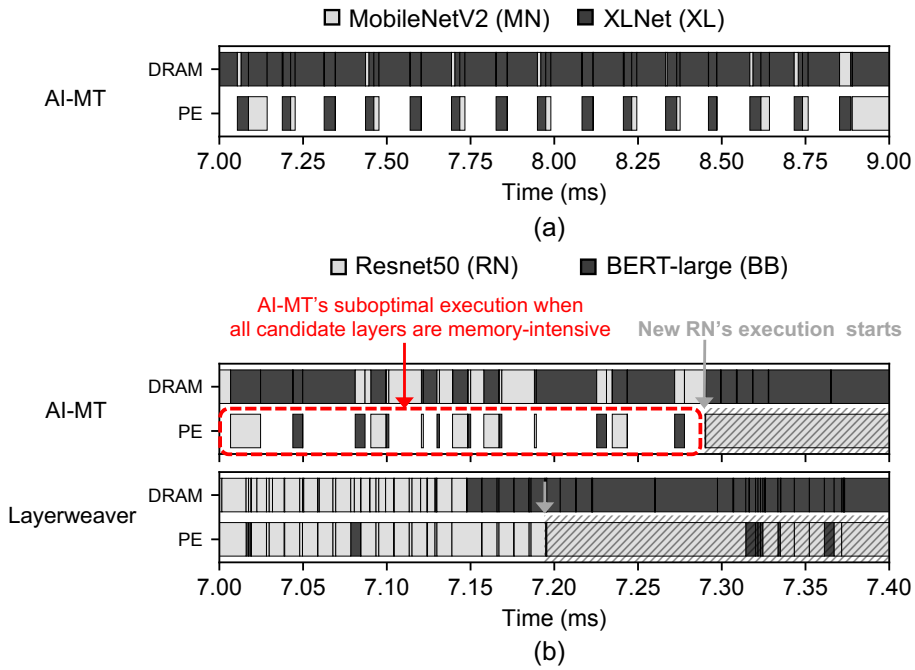


Figure IV.14: Timeline analysis demonstrating (a) a case that AI-MT shows PE under-utilization with MobileNetV2 (MN) and XLNet (XL). And (b) shows a ResNet50 (RN) and BERT-large (BL) case that AI-MT suffer from the starvation originated from memory-intensive layers in compute-intensive models (e.g. FC layers in RN).

utilizes time-multiplexed multi-DNN execution to improve system throughput. It aims to balance both compute and memory resource usage by maintaining the decoupling distance within a desired level. One limitation of AI-MT is that its performance largely depends on the choice of two user-defined thresholds. Even with careful selection of those parameters via extensive 2D parameter sweeping, we find that Layerweaver consistently outperforms AI-MT across all workloads as shown in Figure IV.10 and Figure IV.12.

Figure IV.14(a) shows a case where AI-MT execution results in the PE under-utilization (MN+XL in Figure IV.10(b)). When AI-MT scheduler finds

that the system has little on-chip memory space left, it blindly selects a layer with the least compute time, with an intention of avoiding the memory idle time that can occur from scheduling a layer with a long compute time. Unfortunately, in many cases, such a layer is the one from a memory-intensive model, which also consumes a large amount of on-chip memory space, and thus does not lower the on-chip memory usage. Thus, the problem persists and the system suffers from PE under-utilization since the scheduler favors layers from memory-intensive models. In contrast, Layerweaver achieves high performance for all cases by estimating the cost of a particular scheduling decision in a more formal way and directly aiming to minimize the resource idle time.

Figure IV.14(b) presents another scenario where AI-MT fails to make good scheduling decisions. This is a case where all layers in `candidateGroup` are all memory-intensive, and there exists a sufficient decoupling distance. The particular timeline shown in the figure is from the RN+BL workload where a compute-intensive model (RN) has a memory-intensive layer (e.g., FC or 1x1 Conv). In this case, AI-MT randomly selects a layer to execute. Unfortunately, every time AI-MT schedules a layer from BERT-large (BL) in this situation, a large amount of PE under-utilization happens. On the other hand, Layerweaver intelligently chooses to execute the memory-intensive layer from a compute-intensive model, expecting the compute-intensive layer to follow the current memory-intensive layer (explained in the "Starvation Prevention" paragraph in Section IV.4). As a result, Layerweaver can avoid the unnecessary PE idle time that AI-MT suffers from.

Sensitivity to Changes in Workload Characteristics. Figure IV.15 shows the

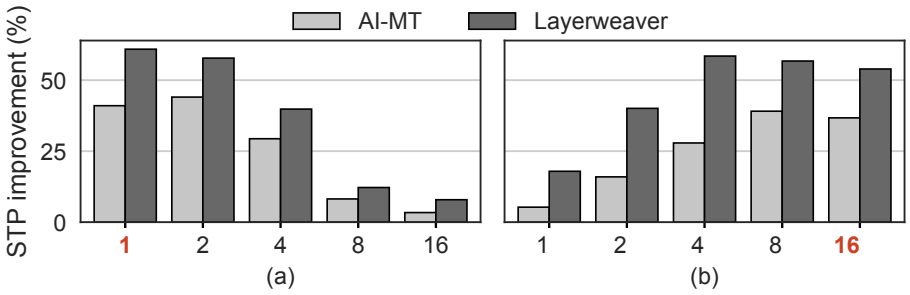


Figure IV.15: Average system throughput (STP) on (a) memory-centric NPU and (b) compute-centric NPU for multi-batch streams (stream # = 2) scenario. Various batch size from 1 to 16 is used to demonstrate its sensitivity. The bold label denotes the selected batch size for workload-specific evaluation (Figure IV.10).

average throughput improvement in N-batch streams for memory-centric and compute-centric NPUs. The setup is similar to Figure IV.10 except that we vary batch sizes for each NPU and report the geomean throughput improvement for each case. Changing the batch size affects the arithmetic intensity of the model. And both Layerweaver or AI-MT [110] benefit the most when one of the models is compute-intensive, and the other is memory-intensive in that particular NPU. However, depending on batch sizes, both models can be *relatively* compute-intensive or memory-intensive as implied in Figure II.7. For example, Figure IV.15(a) shows that the larger batch size makes both workloads to be compute-intensive and lower the benefits of multi-model scheduling on memory-centric NPU. On the other hand, the smaller batch size makes both models to be memory-intensive on compute-centric NPU and lower the benefits of Layerweaver or AI-MT. In all cases, Layerweaver outperforms AI-MT by a significant margin.

Layerweaver with More Than Two Models. Layerweaver can also be utilized with more than two models. Here, we evaluate a case where four models (two

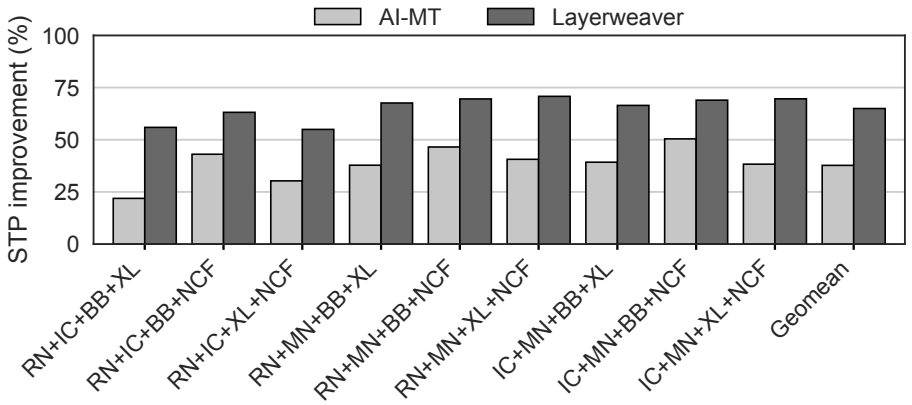


Figure IV.16: Single-batch streams (# streams = 4) system throughput (STP) on a compute-centric NPU.

compute-bound, two memory-bound) are deployed. Specifically, we assume a single-batch streams scenario on compute-centric NPU device. To make the number of combinations manageable, we do not include ResNeXt50 and BERT-large for this experiment to obtain 9 combinations (instead of 36) as shown in Figure IV.16. It shows that Layerweaver can achieve substantial speedups for these workloads as well. Layerweaver demonstrates 27.2% higher throughput improvement than AI-MT. However, we observe that utilizing Layerweaver for more than two models does not bring much additional benefit. Compared to the case of simply utilizing the schedule that stitches two-model-interweaved schedules (e.g., RN + BB schedule followed by IC + XL schedule), the four-model-interweaved schedule resulted in the small geomean STP gain (i.e., $\langle 1$ % average). As long as one model is memory-intensive and the other is compute-intensive, Layerweaver almost fully utilizes resources just with two models.

IV.6. Related Work

Task Multi-tasking on Accelerators. Minimizing performance interference caused by multi-tasking on GPU has been previously studied [117, 118, 119]. Prophet [117] and Baymax [118] acknowledge that resource contention by task co-location and PCIe bandwidth contention caused by data transfers is crucial for multi-tasking performance. Thus, they identify the task co-location performance model to improve compute utilization of GPUs. In contrast, Layerweaver leverages different workload characteristics of DNN models to balance resource utilization on the emerging NPU hardware. AI-MT [110] proposes a TPU extension to support time-multiplexed multi-model execution for optimizing throughput. However, its scheduling algorithm largely relies on heuristics requiring fine tuning of two user-defined thresholds, which is burdensome and suboptimal. We quantitatively compared the quality of scheduling to demonstrate Layerweaver substantially outperforms AI-MT without dedicated hardware support or parameter tuning (Section IV.5.C).

Priority-based Task Preemption. To satisfy the latency constraints of high-priority inference tasks, preemption-based approaches have been proposed [120, 121, 122, 123]. PREMA [120] introduces an effective preemption mechanism that considers the task size and its priority to balance throughput and latency, and a preemptible NPU architecture holding metadata for task switching. TimeGraph [121] proposes a real-time device-driver level scheduler based on two-priority policy using GPU resource usage. Tanasic et al. [123] devise two preemption mechanisms using context switch and GPU

SM draining, respectively, to reduce the performance overhead of preemption. These proposals target to improve QoS for the latency of requests but not increase system throughput. Instead, Layerweaver demonstrates throughput improvement by co-scheduling multiple heterogeneous DNN models.

DNN Serving System Optimization. TensorFlow serving [17] is a production-grade DNN serving system for a serving system. Also, other serving systems such as SageMaker [124], Google AI platform [125], and Azure Machine Learning [126] offer separate online and offline services that automatically scale models based on their load. Clipper [15] targets a low-latency prediction serving system on top of various machine learning frameworks using caching, batching, adaptive model selection. Based on Clipper, Pretzel [16] improves the inference latency by optimizing the serving pipelines. Since these inference serving systems use the only coarse-grained (e.g., model, input batch) scheduling, the results can be suboptimal compared to Layerweaver, which exploits fine-grained scheduling at a layer granularity across different models.

V. Layerweaver+: QoS-aware DNN Scheduler for Multi-tenant NPUs

V.1. Overview

Layerweaver+ replaces the greedy scheduler explained in Chapter IV to achieve the two design goals: maximizing the inference throughput and bounding each request's latency to a given deadline. To this end, we introduce two operation modes in Layerweaver+: 1) *Select the layer that causes the minimum resource idle time first*, and 2) *Select the layer with the minimum QoS slack time*. In the first mode, the QoS slack time is used for a tie-breaker (i.e., prioritizing the layer with the smallest slack time). Layerweaver+ usually operates in Mode 1 to maximize system throughput. However, when QoS violation is imminent for a request, the scheduler switches to Mode 2 to schedule the next layer from that request.

Before describing the intricate details of the QoS-aware scheduling algorithm, Figure V.1 and V.2compare the behaviors of Layerweaver and Layerweaver+ scheduling. In the baseline scheduler executing one request at a time (Top), each request will be executed in the order of arrival. In this case, a QoS violation happens for Request B. Figure V.1(a) and (b) illustrate the scheduling behaviors of Layerweaver with no consideration for QoS, which result in QoS violations for both Request A and B. In contrast, Figure V.2(a) and (b) show the scheduling behaviors of Layerweaver+, which prioritizes a

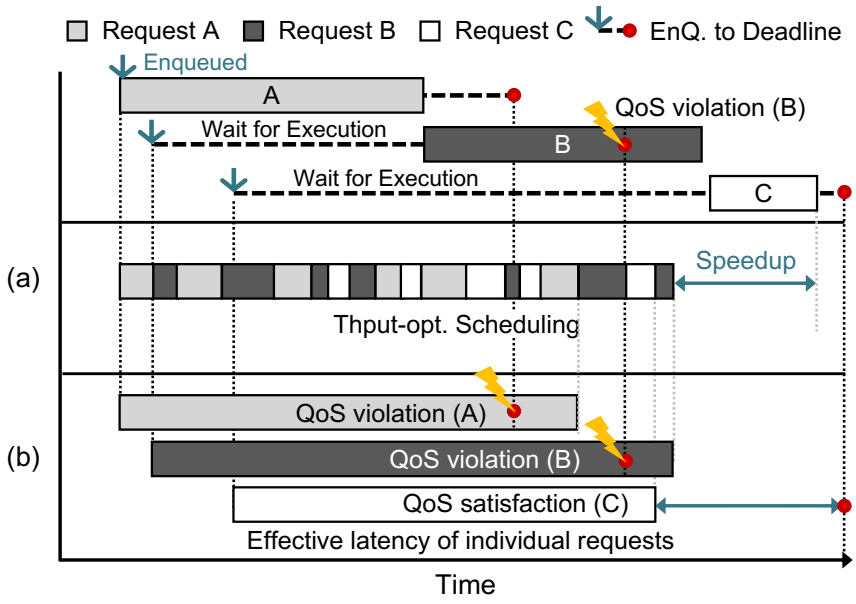


Figure V.1: Illustration of Layerweaver scheduling.

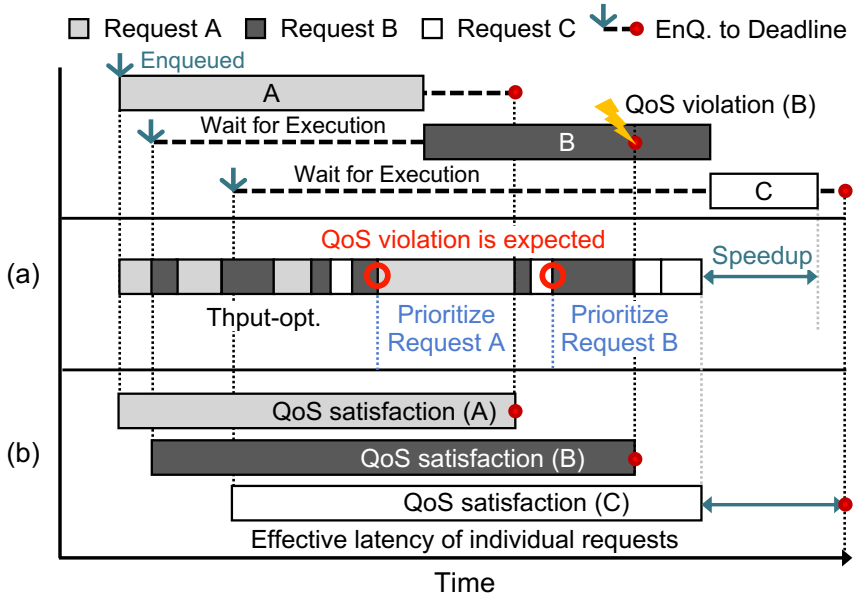


Figure V.2: Illustration of Layerweaver+ scheduling.

```
                          Model info.        QoS target of M_i      Timestamp of enqueue for M_i
1  def Schedule(M_0, ..., M_{k-1}, QoS_0, ..., QoS_{k-1}, EnqTime_0, ..., EnqTime_{k-1}):
2      totalSteps = sum_{i=0,...,k-1}(len(M_i))
3      progress = [0, ..., 0] # length k
4      schedState = [t_m:0, t_c:0, l:[ ]]; schedule = [ ]
5      for step in range(totalSteps):
6          # checks for pause
7          for i in range(k):
8              CheckEnd(progress[i], M_i)
9          # examine all candidate layers
10         candidateGroup = [M_i[idx] for (i, idx) in enumerate(progress)]
11         stateList = [ ]; slackTimeList = [ ]
12         for i in range(k):
13             # schedule state update
14             newSchedState = UpdateSchedule(schedState, candidateGroup[i])
15             stateList.append(newSchedState)
16             slackTimeList.append(EnqTime_i + QoS_i − schedState[t_c])
17         # schedule a layer with minimum idle time for throughput
18         schedState_thput, tidx = SelectMinIdleTime(candidateGroup, stateList, slackTimeList)
19         # schedule a layer with minimum QoS slack time (Most imminent for QoS violation)
20         schedState_urgent, uidx = SelectMinSlackTime(slackTimeList, stateList)
21         # choose trade−off between throughput and latency
22         StnTime = GetStandaloneTime(M_uidx[progress[uidx]:-1])
23         if EnqTime_uidx + QoS_uidx − schedState_thput[t_c] > StnTime:
24             selectedIdx = uidx
25             schedState = schedState_urgent
26         else:
27             selectedIdx = tidx
28             schedState = schedState_thput
29         schedule.append(candidateGroup[selectedIdx])
30         progress[selectedIdx]++
31     return schedule
```

Figure V.3: Layerweaver+ scheduling algorithm. The newly augmented parts for QoS-aware scheduling are highlighted in gray.

request whose deadline is imminent. With this mechanism, Layerweaver+ generates a schedule that meets the deadlines of all three requests at the expense of slight degradation in system throughput due to prioritization without interleaving multiple models.

V.2. Two-mode Scheduling Algorithm

Figure V.3 shows a pseudocode of the Layerweaver+ scheduling algorithm. The scheduling function now takes the QoS target (deadline) (QoS_i) and the timestamp of the arrival time ($EnqTime_i$) for each request as input. The algorithm then computes and maintains the amount of QoS slack time (i.e., the time margin until QoS violation) in `slackTimeList` data structure (Line 16).

Mode 1: Select the layer with the minimum resource idle time. As with the original greedy algorithm, the function `SelectMinIdleTime()` finds the layer that has the minimum resource idle time among `candidateGroup` (Line 18). If there are multiple such layers, the algorithm selects the next layer from the request that has the least QoS slack time as a tie-breaker. This tie-breaking rule helps reduce the chances for QoS violation while maintaining high resource utilization and hence system throughput.

Mode 2: Select the layer with the minimum QoS slack time. Simply utilizing the QoS slack time as a tie-breaker is often not sufficient to prevent QoS violation. In some cases, the scheduler needs to schedule the layer from a request whose QoS violation is imminent. For this, we newly introduce the `SelectMinSlackTime()` function that selects the next layer from the request having the minimum QoS slack time (Line 20). This is a relatively simple function that looks up `slackTimeList` to identify the request with the minimum QoS slack time.

Putting It All Together. The QoS-aware scheduler operates in Mode 1 for most of the time. However, whenever it finds a request which may potentially fail

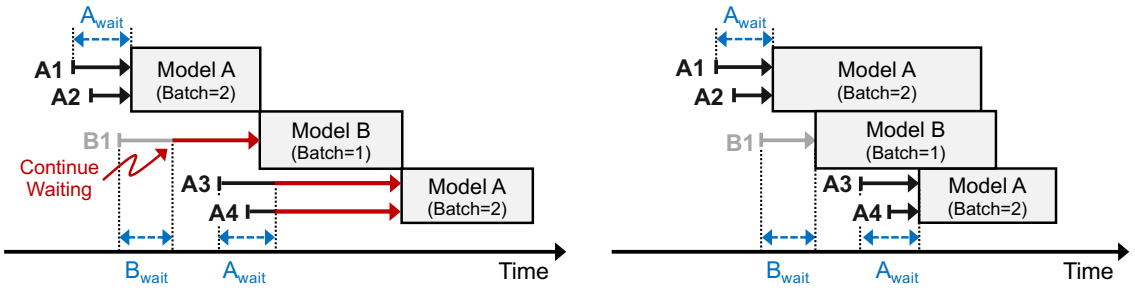


Figure V.4: The example timeline of two different batching schemes (# streams = 2). A_{wait} and B_{wait} denote the waiting time of Model A and B, respectively, to construct a batch.

to satisfy the QoS constraints, it switches to Mode 2 to prioritize layers from that particular request for scheduling. Specifically, our scheduler determines that a request may potentially fail to satisfy the QoS constraints when the estimated remaining standalone execution time for the request exceeds the estimated QoS slack time (Line 22). For the estimation of the remaining standalone execution time for a request, we utilize the following equation: $\sum_{L=progress,\ldots,end} \max(m_L, c_L)$ where m_L and c_L denote memory access time and computation time of layer L, respectively. This equation approximates the remaining standalone execution time of the model by assuming that each layer's execution time can be approximated as the larger of its memory access time and compute time. This simple equation allows the scheduler to calculate the slack time quickly and effectively prevents QoS violation according to our evaluation in Section V.3.B.

Request Batching. Many inference serving systems utilize request batching [15, 17, 18, 61]. For example, if multiple heterogeneous requests are

serviced by a single NPU, the conventional request batching enforces the very first request to wait for a pre-determined time interval and form a batch during the interval. Any requests that arrive after the end of this interval are serviced later. Since the baseline scheduler executes one model at a time, the model heterogeneity only makes the problem worse. If some requests of Model A have been batched before, a request of Model B should wait until the batched inference for Model A completes even if Model B request arrived within the same time interval. With Layerweaver+ (and Layerweaver), it is possible to execute two or more models simultaneously in a time-multiplexed manner. The overall operation is similar to that of the conventional request batching, but the difference is that requests from both Model A and B (or the same model) can be executed concurrently.

V.2.A. Discussion

Scheduling Cost. Layerweaver+ scheduler has the same complexity as the previous one in Chapter IV, which is $O(kN)$ where k is the number of requests residing in a service queue, and N is the number of layers to interweave at a single scheduler invocation. However, if there is no approximation when estimating the remaining standalone execution time (Section V.2), the complexity proportionally increases to the number of remaining layers. However, the scheduler in the host CPU can perform the estimation with parallelism thanks to the approximation with negligible overheads. In addition, there could be too many requests in the service queue (k) compared to the dequeuing speed of the scheduler. In that case, the scheduler can output a batch of

schedules to mitigate scheduling overhead and manage the average number of requests in the service queue.

V.3. Evaluation

V.3.A. Methodology

Simulation Setup. To model cycle-level behaviors of an NPU, we have extended MAESTRO [113] to estimate the computation and data transfer time while considering the effect of data prefetch and random query arrivals. We used an NPU accelerator featuring 128 TOP/s of computation, 100 GB/s off-chip memory bandwidth and 50MB on-chip buffer size supposing inference-only NPU. We select three compute-intensive models: InceptionV3 (IC), MobileNetV2 (MN) and three ResNet50 (RN); and memory-intensive models: BERT-base (BB), BERT-large (BL), and XLNet (XL).

Service Scenario. We extend the server scenario of MLPerf inference [2] benchmark to support multiple different kinds of inference requests. It models a case where a single NPU serves randomly arrived requests for different DNN services. When assuming two models are offloaded into a NPU, the interval between consecutive requests follows the Poisson distribution. We controlled the request arrival ratio between model A and model B by manipulating the Poisson parameter of model A (λ_A) and B (λ_B). The exact relation among

these parameters could be summarized as follows.

$$QPS_{total} = QPS_A + QPS_B = \frac{1}{\lambda_A} + \frac{1}{\lambda_B}$$
$$K = \frac{\lambda_B}{\lambda_A}, \quad \lambda_A = \frac{1}{QPS_{total}}(\frac{K+1}{K}), \quad \lambda_B = \frac{1}{QPS_{total}}(K+1)$$

Each request has a specific QoS constraint which represents its hard deadline. We set the QoS constraints to be <15ms, <130ms for computer vision tasks (RN, RX, and MN) and NLP tasks (BB, BL, and XL), respectively. We compare the maximum STP (System Throughput) that the underlying system can sustain with less than 1% QoS violation [2]. The metric implies that merely increasing STP would incur a significant amount of QoS violations. And the system that meets the condition would also satisfy 99%-tile latency of requests would not violate QoS constraints. We evaluate three schedulers explained in Section V.2: *Baseline* [18], Layerweaver [1] and Layerweaver+.

V.3.B. Results

Throughput. Figure V.5 shows the maximum sustainable system throughput (STP) normalized to the baseline. Overall, Layerweaver even without considering QoS achieves notably better throughput than the baseline with request batching. This is because Layerweaver can interweave two requests, achieving much higher resource utilization than the baseline which serves one batch of the same model at a time. Layerweaver+ (Section V.2) further improves the throughput as it substantially reduces the number of requests violating QoS constraints. Moreover, it also shows that the approximate scheduler utilizing the simple equation for estimating standalone execution time (*Lay-*

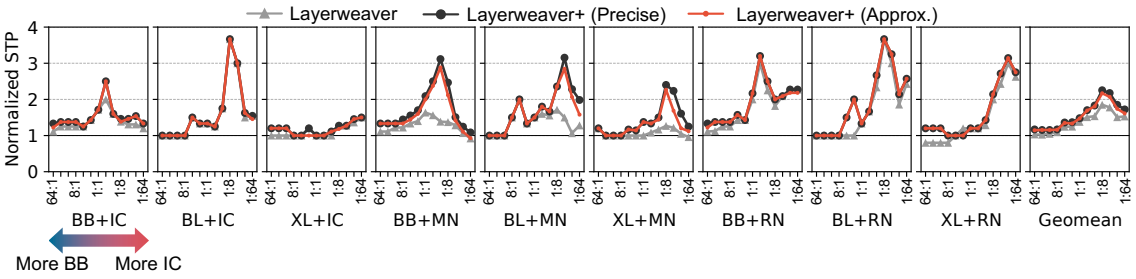


Figure V.5: Maximum sustainable system throughput (STP) normalized to *Baseline* for multi-stream server scenario with two streams. The X-axis represents the varying request arrival ratio between the two streams (with 2× per step). The leftmost tick corresponds to a 64:1 ratio where the memory-intensive model (e.g., BB) has a 64× higher arrival rate than the compute-intensive model (e.g., IC). Layerweaver+ can sustain much higher STP than the baseline for all cases and significantly outperform Layerweaver [1] for some cases.

erweaver+ (*Approx.*)) delivers comparable throughput to a hypothetical precise scheduler utilizing the precise standalone execution time (*Layerweaver+* (*Precise*)). Thus, the approximate scheduler works well with negligible performance degradation while reducing the computation cost of the scheduler significantly.

Tail Latency. Figure V.6 compares the tail latency distribution of the three schedulers in question. Specifically, Figure V.6(a) shows the tail latency of two interleaved models (RN and BB) when requests are injected at the rate of $QPS_{RN} = 800$ and $QPS_{BB} = 200$. For the baseline, about 80% (50%) of the RN (BB) requests fail to meet the 15 ms (130 ms) deadline, mainly because the overall system throughput is too low without layer-wise interleaving. Layerweaver substantially improves the system throughput. However, 10% of the RN requests still violate the 15 ms deadline because Layerweaver schedules BB more frequently to maximize the overall throughput, even though BB has

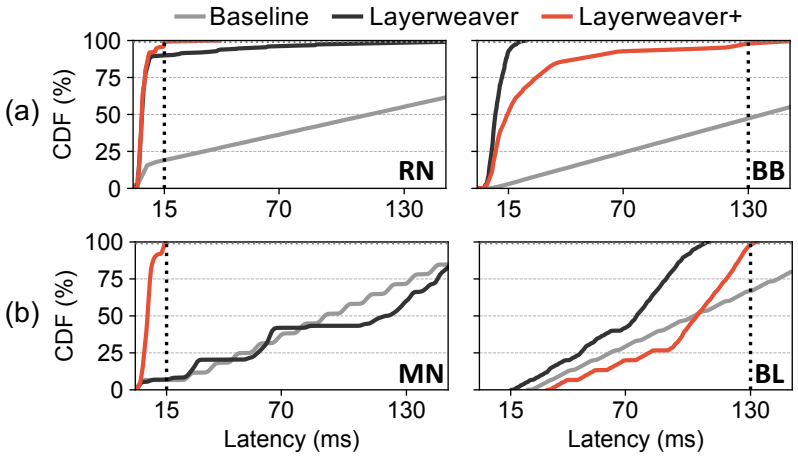


Figure V.6: Tail latency of (a) ResNet50 (RN) and BERT-base (BB); and (b) MobileNetV2 (MN) and BERT-large (BL). The vertical lines represent QoS constraints for vision (15ms) and NLP (130ms) tasks [2].

a relatively loose deadline and thus does not have to be scheduled that frequently. In contrast, Layerweaver+ prioritizes RN requests having a tighter deadline when their QoS slack becomes small. With this mechanism, Layerweaver+ can serve 99%+ requests within the specified deadlines. A similar behavior is observed in Figure V.6(b) at the injection rate of $QPS_{MN} = 7530$, $QPS_{BL} = 470$ as well.

Sensitivity of QoS. Figure V.7 presents a sensitivity study with varying QoS constraints. Overall, the figure shows that increasing the deadline for compute-intensive vision tasks (i.e., RN, MN) or memory-intensive tasks (i.e., BB, BL) leads to higher throughput in some cases. Technically, the baseline and the Layerweaver schedule do not change, but more inference queries arriving on time can improve the effective throughput when the deadline is relaxed. Layerweaver+, on the other hand, adjust the schedule based on the deadline of each task. In some cases, Layerweaver+ throughput does

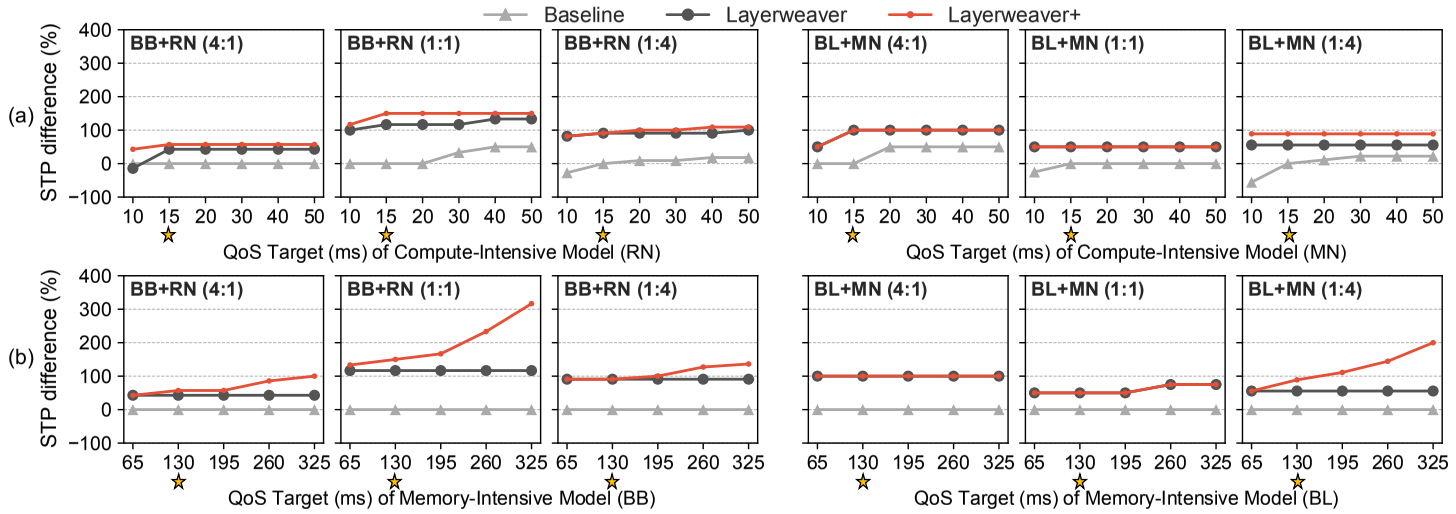


Figure V.7: System throughput changes normalized to the baseline throughput with varying QoS constraints (in milliseconds) for (a) compute-intensive models (RN, MN) and (b) memory-intensive models (BB, BL). The star represents the default QoS constraints in the MLPerf inference (15ms, 130ms). The ratio in parentheses denotes request rate between two models. For example, BB+RN (4:1) means BB has a 4× higher request rate than RN.

not improve; this is mostly because the original Layerweaver without QoS consideration already satisfies QoS target for most of the queries. In cases where Layerweaver+ throughput improves with the increased QoS deadline (i.e., BB+RN (1:1), BL+MN(1:4)), Layerweaver+ can reduce the frequency to switch to Mode 2 (prioritizing requests whose deadline is imminent), which would have negative impact on system throughput. By avoiding prioritizing a specific model, Layerweaver+ can operate in Mode 1 (throughput-oriented mode) to maximize the overall resource utilization, and hence throughput.

VI. Conclusion and Future Work

VI.1. Conclusion

This thesis introduces three software frameworks designed for improving the resource efficiency of NPUs. Firstly, we present *libnumber*, a portable C++ API and auto-tuning framework that optimizes number representation for each layer of a DNN. By introducing the `Number` ADT, *libnumber* encapsulates the internal representation of a number, thus separating the concern for developing an effective DNN from the concern of optimizing the number representation at a bit level. While the task of quantization has been performed in an ad hoc manner, *libnumber* proposes a systematic approach to it by providing a common API and a flexible auto-tuner. We also propose Layerweaver, a DNN inference serving system with a novel multi-model scheduler, which eliminates temporal waste in compute and memory bandwidth resources via layer-wise time-multiplexing of two or more DNN models with different characteristics. Layerweaver+ extends Layerweaver scheduling algorithm to support request prioritization, thereby preventing QoS violation without losing benefits from throughput-oriented scheduling. The resulting quantization framework is easy to use, requiring minimal programmer effort, while producing a high-quality search for an optimal number representation for each layer of a DNN. And we show that our layer-wise scheduling framework achieves near perfect resource utilization on industry standard MLPerf benchmark with minmal runtime search costs.

VI.2. Future Work

The NPU scheduler runtime can be further extended to efficiently partition NPU resources while the schedulers in this thesis are designed only for temporal resource utilization. There already exist some previous works that propose spatial partitioning NPU resources [127, 128] to prevent fragmentation of computing units. However, how dynamically scheduling NPU resources with the balanced throughput-accuracy-latency trade-off on runtime is still in question. Because the auto-tuning process of *libnumber* is designed for offline search requiring substantial tuning time, there should be an approximation to determine the efficiently quantized representation and its mapping to NPU at runtime. One viable approach to achieve this goal is the vertical integration of optimization techniques in this thesis. Preparing some set of differently quantized DNNs or designing DNN models with dynamically tunable precisions [129] would also be helpful by providing knobs to NPU scheudling system.

References

[1] Y. H. Oh, S. Kim, Y. Jin, S. Son, J. Bae, J. Lee, Y. Park, D. U. Kim, T. J. Ham, and J. W. Lee, "Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling," in *The 27th IEEE International Symposium on High Performance Computer Architecture* (*HPCA*), 2021.

[2] V. J. Reddi *et al.*, "MLPerf inference: A benchmarking methodology for machine learning inference systems," in *The 47th International Symposium on Computer Architecture* (*ISCA*), Association for Computing Machinery, 2020.

[3] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Morgan & Claypool Publishers, 2013.

[4] P. Li, "The DQN model based on the dual network for direct marketing," in *The 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems* (*HPCC/SmartCity/DSS*), pp. 1088–1093, 2018.

[5] L. M. Matos, P. Cortez, R. Mendes, and A. Moreau, "Using deep learning for mobile marketing user conversion prediction," in *The 2019 International Joint Conference on Neural Networks* (*IJCNN*), pp. 1–8, 2019.

[6] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The architectural implications of facebook's DNN-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 488–501, 2020.

[7] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *The 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 620–629, Feb 2018.

[8] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, p. 701–710, Association for Computing Machinery, 2014.

[9] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *The 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 855—864, Association for Computing Machinery, 2016.

[10] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars, "Sirius: An open end-to-end voice and vision personal assistant and

its implications for future warehouse scale computers," in *The Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS*), p. 223–238, 2015.

[11] Google, "Google now." `http://www.google.com/landing/now/`.

[12] Y. Xiang and H. Kim, "Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference," in *The IEEE Real-Time Systems Symposium* (*RTSS*), 2019.

[13] M. Yan, A. Li, M. Kalakrishnan, and P. Pastor, "Learning probabilistic multi-modal actor models for vision-based robotic grasping," in *The 2019 International Conference on Robotics and Automation* (*ICRA*), pp. 4804–4810, May 2019.

[14] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[15] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *The 14th USENIX Symposium on Networked Systems Design and Implementation* (*NSDI*), 2017.

[16] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi, "PRETZEL: Opening the black box of machine

learning prediction serving systems," in *The 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 611–626, Oct. 2018.

[17] "TensorFlow serving models." `https://www.tensorflow.org/tfx/guide/serving`.

[18] "NVIDIA Triton Inference Server." `https://developer.nvidia.com/nvidia-triton-inference-server`.

[19] Google, "Cloud TPU at Google Cloud.." `https://cloud.google.com/tpu/docs/system-architecture`, 2018.

[20] O. Wechsler, M. Behar, and B. Daga, "Spring Hill (NNP-I 1000), Intel's Data Center Inference Chip," in *A Symposium on High Performance Chips (Hot Chips)*, 2019.

[21] Cambricon, "Cambricon MLU100." `http://www.cambricon.com`.

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

[23] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," 2017.

[24] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *The 33rd International Conference on Machine Learning*, ICML '16, 2016.

[25] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," 2016.

[26] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. E. Husseini, T. Juhasz, K. Kagi, R. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Y. Xiao, D. Zhang, R. Zhao, and D. Burger, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, 2018.

[27] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, 2008.

[28] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos, "Proteus: Exploiting numerical precision variability in deep neural networks," in *The 2016 International Conference on Supercomputing*, ICS '16, 2016.

[29] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, N. E. Jerger, R. Urtasun, and A. Moshovos, "Reduced-precision strategies for bounded memory in deep neural nets," 2015.

[30] M. Courbariaux, Y. Bengio, and J. David, "Training deep neural networks with low precision multiplications," 2014.

[31] J. t. Qiu, "Going deeper with embedded fpga platform for convolutional neural network," in *The 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, 2016.

[32] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable fpgas," in *The 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, 2017.

[33] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," 2016.

[34] TensorFlow™, "Tensorflow lite." `https://www.tensorflow.org/lite`, 2021.

[35] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," 2017.

[36] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," *CoRR:1611.05431*, 2016.

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2016.

[38] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," *CoRR:1801.04381*, 2018.

[39] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *CoRR:1810.04805*, 2018.

[40] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized autoregressive pretraining for language understanding," *CoRR:1906.08237*, 2019.

[41] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T. Chua, "Neural collaborative filtering," *CoRR:1708.05031*, 2017.

[42] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *The International Symposium on Field-Programmable Gate Arrays* (*FPGA*), 2015.

[43] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *The 49th Annual International Symposium on Microarchitecture* (*MICRO*), 2016.

[44] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, "TANGRAM: Optimized coarse-grained dataflow for scalable NN accelerators," in *The 24th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS*), 2019.

[45] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *The 52nd Annual International Symposium on Microarchitecture* (*MICRO*), 2019.

[46] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "SCALEDEEP: A scalable compute architecture for learning and evaluating deep networks," in *The 44th Annual International Symposium on Computer Architecture* (*ISCA*), 2017.

[47] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *The 24th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS*), 2019.

[48] Y. H. Oh, Q. Quan, D. Kim, S. Kim, J. Heo, S. Jung, J. Jang, and J. W. Lee, "A portable, automatic data qantizer for deep neural networks," in *The 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, 2018.

[49] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee, and D.-K. Jeong, "A^3: Accelerating attention mechanisms in neural networks with approximation," in

The 26th IEEE International Symposium on High Performance Computer Architecture, HPCA, 2020.

[50] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *The 47th Annual International Symposium on Microarchitecture* (*MICRO*), 2014.

[51] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: DNN as a service and its implications for future warehouse scale computers," in *The 42nd Annual International Symposium on Computer Architecture* (*ISCA*), p. 27–40, 2015.

[52] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *The Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS*), p. 615–629, 2017.

[53] "Samsung Exynos 9 Series 9820." `https://www.tensorflow.org/tfx/guide/serving`.

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

[55] Google Cloud, "Edge TPU: Run inference at the edge." https://cloud.google.com/edge-tpu.

[56] A. Krizhevsky, "cuda-convnet: High-performance c++/cuda implementation of convolutional neural networks." https://code.google.com/p/cuda-convnet/, 2012.

[57] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[58] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," 2015.

[59] P. Micikevicius *et al.*, "Mixed precision training," 2017.

[60] D. Burger, "Microsoft unveils Project Brainwave for real-time AI." https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/, 2017.

[61] Y. Choi, Y. Kim and M. Rhu, "LazyBatching: An SLA-aware batching system for cloud machine learning inference," in *The 27th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

[62] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt,

A. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale dnn processor for real-time ai," in *The 45th International Symposium on Computer Architecture*, ISCA '18, 2018.

[63] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," 2016.

[64] J. Redmon, "Darknet: Open source neural networks in c," 2013-2016.

[65] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015.

[66] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[67] Habana, "Habana Goya." `https://habana.ai`.

[68] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *The 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[69] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '16, 2016.

[70] S. *et al.*, "Going deeper with convolutions," in *The IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '15, 2015.

[71] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '16, 2016.

[72] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *The 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, 2014.

[73] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August, "Parallelism orchestration using dope: the degree of parallelism executive," in *The 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, 2011.

[74] A. Raman, A. Zaks, J. W. Lee, and D. I. August, "Parcae: a system for flexible parallel execution," in *The 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, 2012.

[75] T. Chen and C. Guestrin, "Xgboost," *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[76] M. Hopkins, M. Mitzenmacher, and S. Wagner-Carena, "Simulated annealing for jpeg quantization," *arXiv preprint arXiv:1709.00649*, 2017.

[77] B. Wu, Y. Wang, P. Zhang, Y. Tian, P. Vajda, and K. Keutzer, "Mixed precision quantization of convnets via differentiable neural architecture search," *arXiv preprint arXiv:1812.00090*, 2018.

[78] L. Yang and Q. Jin, "Fracbits: Mixed precision quantization via fractional bit-widths," *arXiv preprint arXiv:2007.02017*, 2020.

[79] K. Chitta, J. M. Alvarez, E. Haussmann, and C. Farabet, "Training data subset search with ensemble active learning," 2020.

[80] TensorFlow™, "Tensorflow mechanics 101." `https://github.com/tensorflow/tensorflow/tree/r1.2/tensorflow/examples/tutorials/mnist`, 2017.

[81] Y. LeCun, "The mnist database of handwritten digits," *http://yann.lecun.com/exdb/mnist/.*, 1998.

[82] R. Vasudevan, "Cifar-10 classifier." `https://github.com/vrakesh/CIFAR-10-Classifier`, 2017.

[83] Y. LeCun *et al.*, "Lenet-5, convolutional neural networks," 1998.

[84] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012.

[85] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *The IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '09, 2009.

[86] M. Lin, Q. Chen, and S. Yan, "Network in network," 2013.

[87] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014.

[88] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," 2016.

[89] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision.*, 2010.

[90] Y. t. Jia, "Caffe: Convolutional architecture for fast feature embedding," in *The 22nd ACM International Conference on Multimedia*, ACMMM '14, 2014.

[91] C. A. Coleman *et al.*, "Dawnbench: An end-to-end deep learning benchmark and competition." `https://github.com/stanford-futuredata/dawn-bench-entries`, 2017.

[92] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," 2016.

[93] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi, "Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks," *IEEE Transactions on Neural Networks and Learning Systems.*, 2018.

[94] X. INC., "Xilinx kintex ultrascale fpga family." https://www.xilinx.com/products/silicon-devices/fpga/kintex-ultrascale.html, 2018.

[95] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *The 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, 2015.

[96] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '17, 2017.

[97] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '14, 2014.

[98] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," 2016.

[99] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," 2015.

[100] P. V. Kotipalli, R. Singh, P. Wood, I. Laguna, and S. Bagchi, "Ampt-ga: Automatic mixed precision floating point tuning for gpu applica-

tions," in *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, p. 160–170, 2019.

[101] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated End-to-End optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

[102] Z. Wang, D. Grewe, and M. F. P. O'boyle, "Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems," *ACM Transaction on Architecture and Code Optimization.*, 2014.

[103] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *The 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, 2008.

[104] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques*, PACT '17, 2017.

[105] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in

The 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, 2013.

[106] "AWS Elastic Load Balancing." `https://aws.amazon.com/ko/elasticloadbalancing`.

[107] "Load Balancing on IBM Cloud." `https://www.ibm.com/cloud/learn/load-balancing`.

[108] "Kubeflow: The Machine Learning Toolkit for Kubernetes." `https://www.kubeflow.org`.

[109] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *The 43rd International Symposium on Computer Architecture* (*ISCA*), 2016.

[110] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," in *The 47th Annual International Symposium on Computer Architecture* (*ISCA*), 2020.

[111] "TensorFlow Core v2.3.0. Keras API docs.." `https://www.tensorflow.org/api_docs/python/tf/keras/layers`, 2020.

[112] "PyTorch 1.6.0 documentation.." `https://pytorch.org/docs/stable/nn.html`, 2020.

[113] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of DNN

dataflow: A data-centric approach," in *The 52nd Annual International Symposium on Microarchitecture* (*MICRO*), 2019.

[114] Y. Wang, G.-Y. Wei, and D. Brooks, "A systematic methodology for analysis of deep learning hardware and software platforms," in *Machine Learning and Systems* (*MLSys*), pp. 30–43, 2020.

[115] G. Zhou, J. Zhou, and H. Lin, "Research on NVIDIA deep learning accelerator," in *The 12th International Conference on Anti-counterfeiting, Security, and Identification* (*ASID*), 2018.

[116] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[117] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," in *The 22nd International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS*), 2017.

[118] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," in *The 21st International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS*), 2016.

[119] P. Yu and M. Chowdhury, "Fine-grained GPU sharing primitives for deep learning applications," in *Machine Learning and Systems 2020* (*MLSys*), pp. 98–111, 2020.

[120] Y. Choi and M. Rhu, "PREMA: A predictive multi-task scheduling algorithm for preemptible neural processing units," in *The 26th IEEE International Symposium on High Performance Computer Architecture* (*HPCA*), pp. 220–233, 2020.

[121] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: GPU scheduling for real-time multi-tasking environments," in *The 2011 USENIX Conference on USENIX Annual Technical Conference* (*ATC*), 2011.

[122] G. A. Elliott, B. C. Ward, and J. H. Anderson, "Gpusync: A framework for real-time GPU management," in *The 2013 IEEE 34th Real-Time Systems Symposium* (*RTSS*), pp. 33–44, 2013.

[123] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *The 41st International Symposium on Computer Architecture* (*ISCA*), pp. 193–204, 2014.

[124] Amazon, "Amazon SageMaker." `https://aws.amazon.com/sagemaker/`.

[125] Google, "Google AI platform." `https://cloud.google.com/ai-platform`.

[126] Microsoft, "Microsoft azure machine learning." https://docs.microsoft.com/en-us/azure/machine-learning/.

[127] H. Kwon, L. Lai, M. Pellauer, T. Krishna, Y.-H. Chen, and V. Chandra, "Heterogeneous dataflow accelerators for multi-dnn workloads," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 71–83, 2021.

[128] B. Fang, X. Zeng, and M. Zhang, "Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, p. 115–127, 2018.

[129] H. Yu, H. Li, H. Shi, T. S. Huang, and G. Hua, "Any-precision deep neural networks," *arXiv preprint arXiv:1911.07346*, 2019.

논문요약

NPU 리소스 활용률 개선을 위한 하드웨어 및 소프트웨어 기법

성균관대학교 일반대학원
전자전기컴퓨터공학과
오영환

최근 AI기반 응용 서비스들이 급증함에 따라 DNN을 효율적으로 수행할 수 있는 시스템이 크게 각광받고 있다. 이러한 딥러닝 기반의 서비스들은 기존의 응용 대비 연산 및 메모리 집약적인 특성을 동시에 가지고 있으며, 따라서 시스템에 막대한 양의 연산 및 메모리 리소스를 필요로 하게 된다. 따라서 딥러닝 기반의 서비스를 효율적으로 처리하기위해 모바일에서 데이터센터 등의 대규모 시스템까지 다양한 환경에서 뉴럴 프로세싱 유닛(NPU)이 널리 활용되고 있다. 하지만, NPU가 다양한 조직에서 독자적으로 개발됨에 따라 각 NPU에서 사용되는 데이터 표현이나 연산-메모리대역폭 비율(Compute-to-Memory Bandwidth Ratio) 등의 특성은 매우 다르게 나타난다. 따라서, 다양한 종류의 DNN을 하는 NPU 시스템에서는 가장 최적의 데이터 표현 및 실행 스케줄을 고정적으로 결정하는 것이 매우 어렵다.

구체적으로 본 논문에서는 현재 NPU에서 널리 사용되는 데이터 양자화 및 리소스 스케줄링 기법들이 다음과 같이 중요한 한계점들이 있음을 밝힌다. 첫 번째로, 특정한 DNN 모델을 표현하는 최적의 데이터 표현을 찾기 위해서, 현재의 양자화 기법들은 탐색을 수행하는 조건을 크게 제약한다. 이는 DNN의 모든 레이어에 대하여 다양한 비트 수를 탐색하려면 방대한 양의 탐색 시간을 요구하기 때문인데, 현재의 탐색 알고리즘은 실현 가능한 시간 내에 최적의 데이터 표현을 결정하기 어렵다. 뿐만 아니라 그 기법들은 대부분 특정 DNN 프레임워크(예, TensorFlow) 혹은 특정 하드웨어 플랫폼을 가정하고 구현되었는데, 현재의

다양한 NPU 기반 플랫폼들에 일관적으로 적용되기 어렵다. 두 번째로 현재의 NPU 리소스 스케줄링 기법은 NPU의 연산 및 메모리 리소스를 충분히 활용할 수 없어서 스루풋 성능에 제약이 발생한다. 이는 NPU의 연산-메모리대역폭 비율과 DNN의 연산집약도(Arithmetic Intensity)가 큰 차이를 보이기 때문인데 이를 보완할 방법은 현재 전무한 상황이다. 더구나 데이터센터에서는 사용자에게 고품질의 서비스를 제공하기 위하여 일반적으로 서비스 품질(혹은 최대지연시간)에 조건을 두기 때문에 스케줄링 문제가 더욱 복잡해진다.

따라서, 이러한 문제를 해결하기위해 본 논문은 NPU기반 시스템을 위한 세 가지 기법을 고안하였다. 첫 번째는 *libnumber* 로 손쉽게 이식이 가능하면서도 자동으로 양자화된 데이터 표현을 검색하는 프레임워크를 제안한다. *libnumber* 는 사용자로부터 내부의 데이터 표현(Type, Bit width, Bias 등)을 추상화시키며, 다른 어떤 알고리즘보다도 넓은 범위의 표현을 효율적이고 체계적인 방법으로 검색한다. 두 번째 기법은 Layerweaver 로 서로 다른 특성의 DNN 모델을 교차 실행하여 시간적으로 NPU 리소스가 낭비되는 것을 최소화한다. 마지막으로 Layerweaver+ 는 Layerweaver 를 확장하여 서비스 품질을 고려한 스케줄링이 가능하도록 설계하였다. 본 논문에서는 제안한 자동 양자화 및 리소스 스케줄링 기법을 NPU기반 시스템의 시뮬레이션을 통해 평가하였다. 이를 통해서, Layerweaver 와 Layerweaver+ 는 다양한 시나리오 및 다양한 NPU에서 거의 완벽한 NPU 리소스 활용률을 달성하였으며, *libnumber* 는 DNN의 파라미터 사이즈와 중간 데이터를 획기적으로 줄일 수 있다는 사실을 실험적으로 입증하였다.

주제어: 뉴럴 네트워크, 모델 서빙 시스템, 양자화, 리소스 스케줄링, 딥러닝 가속기

Ph.D. Dissertation

Hardware and Software Techniques for Improving NPU Resource Utilization

2022

Young H. Oh