TO BRIDGE NEURAL NETWORK DESIGN AND REAL-WORLD PERFORMANCE: A BEHAVIOUR STUDY FOR NEURAL NETWORKS

Xiaohu Tang [1 2] **Shihao Han** [3 2] **Li Lyna Zhang** [2] **Ting Cao** [2] **Yunxin Liu** [2]

ABSTRACT

The boom of edge AI applications has spawned a great many neural network (NN) algorithms and inference platforms. Unfortunately, the fast pace of development in their fields have magnified the gaps between them. A well-designed NN algorithm with reduced number of computation operations and memory accesses can easily result in increased inference latency in real-world deployment, due to a mismatch between the algorithm and the features of target platforms.

Therefore, it is critical to understand the behaviour characteristics of NN design space on target platforms. However, none of existing NN benchmarking or characterization studies can serve this purpose. They only evaluate some sparse configurations in the design space for the purpose of platform optimization rather than the scaling in every design dimension for NN algorithm efficiency. This paper presents the first empirical study on the NN design space to learn NN behaviour characteristics on different inference platforms. The revealed characteristics can be used as guidelines to design efficient NN algorithms. We profile ten-thousand configurations from a cutting-edge NN design space on seven industrial edge AI platforms. Seven key findings as well as their causes and implications for efficient NN design are highlighted.

1 INTRODUCTION

Numerous edge devices such as mobile phones, cameras and speakers provide real-world usage scenarios for NN technology. To enable affordable NN inference on edge devices, remarkable innovations have been achieved on the design of efficient NN algorithms and the development of inference platforms (including hardware accelerators and the corresponding inference frameworks on top).

However, there is a gap between NN algorithm design and inference platforms. Current NN design has no consideration of the features of target platforms but only aims to reduce the number of computation operations (OPs[1]) for efficiency. Given an AI task, as shown in Fig. 1, NN experts carefully configure each dimension in a huge design space by manual or Neural Architecture Search (NAS) (Tan et al., 2019) techniques, to find the model with a tradeoff between high accuracy and low OPs. The designed model is then deployed to various inference platforms.

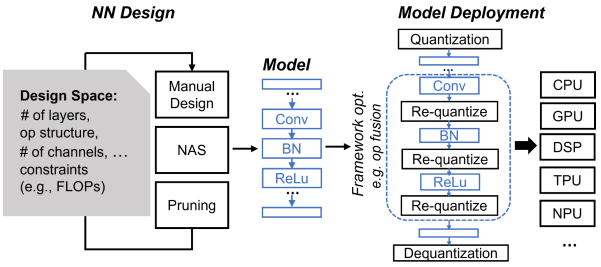

Figure 1. The standard process from NN design to on-device model deployment.

Unfortunately, just reducing OPs or memory accesses does not always reduce inference latency, but easily increases latency in real-world deployment. For example, the well-known MobileNetV3 (219 MOPs, 8 MB memory footprint) has much less OPs and memory footprint than MobileNetV2 (300 MOPs, 11 MB memory footprint). Its inference runs 25% faster than MobileNetV2 in TFLite (Google, 2020b) on ARM Cortex A76 CPU. However, it runs 71% slower than MobileNetV2 in OpenVINO (Intel, 2020) framework on an Movidius VPU chip (Movidius, 2020) due to a mismatched design with platform implementation.

Consequently, it is critical to understand the behaviour characteristics of the NN design space on target platforms to

[1] Zhejiang University (Work was done as an intern at Microsoft Research) [2]Microsoft Research [3]Rose-Hulman Institute of Technology. Correspondence to: Li Lyna Zhang, Ting Cao, Yunxin Liu <lzhani, ting.cao, yunxin.liu@microsoft.com>.

Proceedings of the 4^{th} MLSys Conference, San Jose, CA, USA, 2021.

[1]This paper uses OPs rather than FLOPs due to different precisions of AI chips. This definition also follows (Zhang et al., 2018), *i.e.,* the number of multiply-adds.

find efficient configurations. However, they are still unclear by now. This is partially because of the intrinsic separation of NN designers and platform developers. Besides, there are more and more inference platforms released frequently. Many of them are closed source. It is challenging to understand the behaviours on these platforms.

Although there are NN characterization or benchmarking works before, such as BenchIP (Tao et al., 2018), DeepBench (Baidu, 2020), MLPerf (Reddi et al., 2020), ParaDnn (Wang et al., 2020a) and others (Gao et al., 2020; Bianco et al., 2018; Zhang et al., 2019; Turner et al., 2018; Hadidi et al., 2019; Wu et al., 2019b), they cannot be used to guide efficient configurations for NN design. On the other hand, they aim at performance comparison of NN platforms and expose platform optimization opportunities. Therefore, only some sparse points of a NN design space (*i.e.,* some popular NN models and operators) are included in the evaluation dataset rather than the scaling of each dimension of the NN design space.

This paper presents the first comprehensive empirical study on NN design space to learn its behaviour characteristics on inference platforms. The revealed characteristics can thus be used as guidelines to design fast NNs in real-world deployment, as well as accelerate the design process by excluding inefficient configurations. We evaluate the latency response to the scaling of each dimension of cutting-edge NN design space, including building blocks, the number of channels, input size, kernel size, activation function, and data precision. The size of study dataset is in the order of ten thousands. Rather than for a specific model, the study results can benefit various model design for different AI tasks on a target platform. The paper currently focuses on *CNNs* (Convolutional Neural Networks) due to their high design complexity.

Particularly, this study aims to answer the following questions. *(i)* What are the behaviour characteristics that show an inconsistent latency response to the change of OPs and memory accesses of a configuration in the design space? For example, latency increase is much bigger than OPs increase. These cases should be specifically avoided in the design. *(ii)* What are the root causes for these unexpected characteristics? *(iii)* What are the implications of these characteristics for efficient-NN design?

Surprisingly, our results show that many of latency responses are against the change of OPs and memory accesses, advocating the necessities of this empirical study. By now, we profile the design space on seven representative industrial AI platforms such as Edge TPU (Google, 2020a), Rockchip NPU (Rockchip, 2019) and Kendryte KPU (Kendryte, 2020a) with their corresponding frameworks (a whole list in Table 2). For each design dimension, this paper highlights one finding from the data. Implications for NN design are also described. All the findings are thoroughly explained through hardware features, framework implementations and NN structures.

To demonstrate the value of the findings, we conduct two case studies of NN design: channel pruning and hardware-aware neural network search. By applying these findings to eliminate the inefficient configurations, the design space sizes in the two case studies can be reduced by $10^{12}\times$ and $32\times$, respectively.

Major conclusions from the findings are: *(i)* The use of more convolution channels and bigger filter kernels does not necessarily increase latency. For example, the latency of convolution increases with the number of output channels in a step pattern rather than linear on every platform, except for the KPU. *(ii)* The use of non-Conv operators can largely increase latency except for the ARM CPU. For example, adding Squeeze&Excitation (SE) block to the MobileNetV2 block barely adds time cost on the CPU, but adds $15\times$ time cost on the KPU, and $51\times$ on the Edge TPU. *(iii)* The use of INT8 can achieve $> 11\times$ speedup compared to FP16 on the Rockchip NPU. *(iv)* Considering robust support for various NN models, the ARM CPU is the best platform. Considering the latency, power and energy cost, Edge TPU and the Rockchip NPU are the best platforms. For example, the energy cost of MobileNetV1 inference on the Edge TPU is only 4% of that on the ARM CPU.

To sum up, the key contributions of this paper are as follows. We perform the first comprehensive empirical study on NN design space to learn NN behaviour characteristics for efficient NN algorithm design. We highlight seven findings, and provide thorough explanation as well as the implication for NN design for each finding.

2 Background and Related Work

2.1 Efficient NN design

Discovering efficient NN models remains a laborious task due to the huge design space, and the tension between improving accuracy and reducing inference latency. This section introduces the dataset our paper study on *i.e.,* NN design space, and current techniques for efficient NN design.

2.1.1 Design space

Regarding CNN models, we categorize the design space into layer type and layer hyperparameters.

Layer type. A CNN is composed of sequential layers, where each layer (notated as O) can be a primitive *operator* (*e.g.,* convolution layer) or a *building block* (*e.g.,* separable depthwise convolution layer). Primitive operators mainly include: fully-connected operator, element-wise operator, activation function and convolution operator. A building

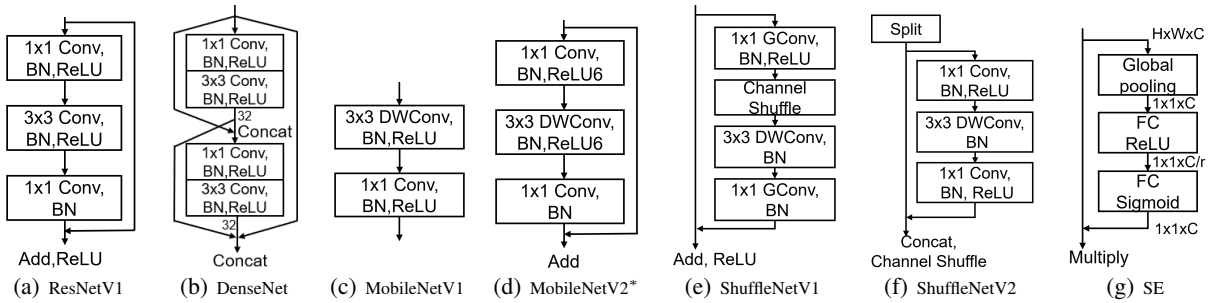

Figure 2. Representative building blocks from state-of-the-art CNN models. *: We also consider MobileNetV2+SE block, we follow (Howard et al., 2019) and insert SE between the DWConv and last 1×1 Conv.

block consists of multiple primitive operators. Fig. 2 shows the commonly used building blocks in efficient CNN design.

Layer hyperparameters. Once the layer type is decided, NN designers need to carefully configure layer hyperparameters to find the tradeoff between accuracy and efficiency. Layer hyperparameters mainly include *(i)* basic parameters: input height (H) and width (W), input channel number (C_{in}) and data precision (P); *(ii)* operator parameters: kernel size (K), stride (S), and number of filters (C_{out}).

In summary, for each layer, the major design space is a 8-dimension vector: $(O, K, S, H, W, C_{in}, C_{out}, P)$. The size of the design space is the multiplication of the size of every dimension and can easily reach billions.

2.1.2 *Design space exploration*

To feasibly reduce design cost from the huge space, NN designers normally tune only one or a subset of dimensions to explore the best configurations, and fix other dimensions to pre-defined configurations. Current exploration approaches are based on human heuristics (He et al., 2016; Sandler et al., 2018) and NAS (Tan et al., 2019; Cai et al., 2019). We now explain the exploration of two design dimensions as examples.

For layer type (O) configuration, the common heuristic is to select an operator or a building block and then repeat it for every layer. Early CNN models frequently apply the 2D convolution operator as the core component of a layer (Krizhevsky et al., 2012). Nowadays, new blocks (He et al., 2016; Huang et al., 2017; Howard et al., 2017; Ma et al., 2018) with reduced cost and higher accuracy have been invented. For example, MobileNetV1 (Howard et al., 2017) introduces depthwise separable convolution to reduce OPs. These cutting-edge blocks are used as candidates for layer type in NAS-based model design (Zhang et al., 2020).

For channel number configuration, a general heuristic is "*half size, double channel*", which was introduced in VGG (Simonyan & Zisserman, 2014). The rule is that when the feature map size is halved, the number of filters (C_{out}) is doubled. Apart from such human-designed heuristics, many pruning-based and NAS-based approaches (Li et al., 2017; He et al., 2018; Liu et al., 2019) are proposed to explore optimal channel number from a candidate integer set.

2.1.3 *Unawareness of deployment platforms*

Current NN design mostly focuses on model accuracy and rarely considers inference efficiency on various edge platforms. The common efforts for efficiency are to limit the OPs of the model or replace the computation-intensive convolution with more memory-intensive operators (Howard et al., 2019; Ma et al., 2018). These efforts can gain latency reduction on CPUs. However, current NN platforms, mainly designed for convolution, hardly adapt to these efforts and thus performance is degraded (analysis in Sec. 4). Furthermore, NN platforms have different features. There is no one-size-fits-all model design for all platforms.

Some NN researchers notice these issues. However, due to the unawareness of behaviour characteristics of the design space, they use the measured (Dai et al., 2019; Yang et al., 2018; Tan et al., 2019) or predicted latency (Yang et al., 2017; Qi et al., 2017) for every configuration in the space to search for the accuracy and efficiency tradeoff. However, considering the huge design space, it is unbearable to conduct real measurements for every model design. Latency prediction can avoid the costly measurements, but it is difficult to build prediction models since many platforms are blackbox and updated frequently.

Some efficient NN designs closely cooperate with platform development teams, such as EfficientNet-EdgeTPU (Suyog Gupta, 2019) and MobileNetEdgeTPU (Andrew Howard, 2019). The two works aim to design efficient models for Edge TPU by augmenting the design space with expert-selected building blocks that run efficiently on Edge TPU. Except for building blocks, other design dimensions are exhaustively searched. However, general NN designers hardly get support from platform teams.

Table 1. Configurations selected for each dimension of the CNN design space. * DFP: dynamic fixed point

	Layer type (O)		Kernel size (K)	Stride (S)	Input $H \times W$ (H=W)	Channel in (C_{in})	Channel out (C_{out})	Precision* (P)
Channel	Conv, DWConv		1,3,5,7	1,2	224,112,56 28, 14,7	3 to 1000	3 to 1000	FP32 FP16 DFP16 INT8 UINT8 DFP8
Operator	Fully connect	-	-	-	-	1024,1280	1000	
	Element-wise	Pooling	7	-	224,112,56 28,14,7,3	32,64,96 128,160,240 256,320,480 512,640	32,64,96 128,160,240 256,320,480 512,640	
		Add, Concat, Multiply, Shuffle	-	-				
	Activation	ReLU, ReLU6, Sigmoid, HardSwish, Swish	-	-				
	Convolution	Conv, DWConv Dilated, Group, MixConv	1,3,5,7	1,2				
Block	MobileNetV1, ShuffleNetV1/V2, DenseNet MobileNetV2 with/without SE, ResNetV1		3,5,7	1,2				

Our work profiles the NN design space to expose behaviour characteristics, which can guide general NN design. The methodology and tool can apply to different platforms.

2.2 NN inference on edge platforms

After NN experts finish model design, an offline deployment process is then executed to enable the model run on target platforms. The software framework of a platform such as TFLite normally conduct two major steps for deployment. *(i)* Convert the original model format to the current framework format. For example, TFLite requires the graph to be converted from the TensorFlow protobuf (*.pb* file) to its FlatBuffer-based format (*.tflite* file); *(ii)* Optimize the model graph, plan model execution, or conduct code generation. The most important graph optimization is *operator fusion* (Chen et al., 2018). It fuses the computation of adjacent operators together and avoids saving intermediate results in memory, which can greatly reduce latency. The memory-intensive operators which cannot be fused into convolution will cause major overhead (analysis in Sec. 4).

2.3 Limitations of existing works for NN design

Our paper aims at efficient NN algorithm design. It studies the scaling of each dimension of the whole design space to tell the efficient configurations.

Existing benchmarking works, on the other hand, are for the purpose of cross-platform comparison to expose NN platform optimization opportunities. Representative works include BenchIP (Tao et al., 2018), DeepBench (Baidu, 2020), AIBench (Gao et al., 2020), ParaDnn (Wang et al., 2020b), MLPerf (Reddi et al., 2020), and NNBench-X (Xie et al., 2019). According to this purpose, they just select some typical points of the design space *i.e.,* mature models or operators, in the profiling dataset. To better expose platform limitations, they also include configurations suitable for hardware evaluation but rarely considered in NN design. For example, BenchIP also covers configurations with diverse hardware behaviours, such as various branch prediction rate and data reuse distances. ParaDnn generates parameterized models, such as varying the batch size to challenge the bounds of the underlying hardware.

3 Evaluation Methodology

In this paper, we present the first study on NN design space. There is no existing public dataset or measurement tool for this purpose. This section elaborates our dataset and tool which automates dataset generation and profiling for latency, energy and model accuracy on target platforms.

3.1 Dataset selection

For the dataset, it is impossible to profile full combinations over all design dimensions in a feasible time. Fortunately, we observe the general practice for NN design is to explore only one design dimension (*e.g.,* C_{out}) at one time to restrain design cost (He et al., 2017; Liu et al., 2019; Tan et al., 2019; Cai et al., 2019). Inspired by this, we fix the configurations of the other dimensions and only vary one dimension at one time to generate the characterization dataset. By this method, the insights for one dimension can be directly applied to NN design process, and the dataset is also constrained to a practical size.

Table 1 lists configurations of each dimension in our dataset, which are from state-of-the-art design space used by the current efficient models for edge platforms (as introduced in Sec. 2.1). Particularly, input height H and width W are set to equal, as this is the most common setting in vision tasks. The C_{out} and C_{in} are only evaluated for convolution (Conv) and depthwise convolution (DWConv) operators as they dominate the CNN inference latency.

The dataset includes six precisions supported by our seven evaluation platforms. Besides latency, data precision also impacts inference accuracy. To evaluate the accuracy impact, our dataset collects 14 representative models from both manually-designed and NAS-searched ones with different levels of computation and memory cost. The model list together with the evaluation results are in Appendix B.4.

3.2 Evaluated edge platforms

We have measured the dataset on seven typical industrial edge platforms listed in Table 2. The abbreviations will be used to refer to each platform. Unless specifically stated, **the hardware processor and its bundled software frame-**

Table 2. Measured platforms. ∗: we measure Adreno GPU's peak performance by the Clpeak (Bhat, 2020) since no official number can be found. Other peak performance numbers are claimed by producers. We only profile one CPU core to avoid scheduling interference.

Processor	Precision	Peak perf. per sec.	Framework	Device	Abbr.
Cortex A76 CPU	FP32/INT8	23 GOPs / core (FP32)	TFLite v2.1	Snapdragon 855 SoC on Mi 9	CPU
Adreno 640 GPU	FP32/FP16	840 GOPs (FP16)∗	TFLite v2.1	Snapdragon 855 SoC on Mi 9	GPU
Hexagon 685 DSP	INT8	256 GOPs	SNPE v1.30(Qualcomm, 2020)	Snapdragon 845 SoC on Pixel3XL	DSP
Movidius Myrid X VPU	FP16	5 TOPs	OpenVINO 2019R2(Intel, 2020)	Intel Neural Compute Stick 2	VPU
Edge TPU	INT8	4 TOPs	TFLite-based	Coral USB Accelerator	TPU
Kendryte KPU	INT8	0.5 TOPs	NNCASE (Kendryte, 2020b)	Cannon Kendryte K210 SoC dev. board	KPU
Rockchip NPU	FP16/DFP16/DFP8/UINT8	3 TOPs (UINT8)	RKNN v1.2.1(Rockchip, 2020b)	RK3399Pro SoC on Toybrick SBC	NPU

work are treated as a whole platform in our analysis. This is because 1) except for the CPU, GPU, and DSP, the other accelerators and frameworks are all blackbox. It is hard to attribute the NN behaviour to the framework or the accelerator; 2) our goal is to guide efficient NN design. To this end, it is not essential to analyze them separately.

We will briefly introduce key features of the evaluated AI accelerators to help understand the analysis in Sec. 4. The main idea of NN accelerators is to support vector or matrix computation units of different width to increase data-level parallelism. It is hard to get more details because these accelerators are proprietary with limited published documents.

Qualcomm Hexagon DSP 600 series (BDTi, 2015) feature a Hexagon vector extension (HVX) for image and computer vision tasks. The width of vector registers is 1024 bits and each VLIW instruction supports four vector slots.

Intel Movidius Myriad X VPU integrates 16 SHAVE (Streaming Hybrid Architecture Vector Engine) cores and a neural compute engine. The SHAVE core (WikiChip, 2018) uses a hybrid RISC/DSP/GPU architecture. The vector register is 128-bit wide. A VLIW instruction supports one to four vector slots.

Rockchip NPU (Rockchip, 2020a) features a NN specific engine and a vector processing unit as the supplement (Rockchip, 2019). The NN engine can run 1920 multiply-add operations in INT8, 192 in INT16, and 64 in FP16 per cycle. The vector unit can perform one multiply/add operation per cycle. Most element-wise and matrix operations are processed in this vector unit.

CNN structure is hardwired in Kendryte KPU as Conv + batch normalization (BN) + activation + pooling. It is not tolerant to other NN structures.

No official design document can be found for Edge TPU.

Frameworks. Except for TFLite, all the other frameworks are proprietary software developed by the processor manufacturer. SNPE's Hexagon NN library for DSP is open source but other parts are closed-source. There are quite a few open-source frameworks for the CPU and GPU. We pick TFLite due to its wide usage in the real-world mobile applications (Xu et al., 2019).

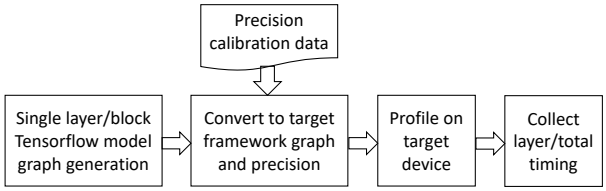

Figure 3. The measurement tool.

Table 3. Operator OPs and mac calculation.

	Conv	Group Conv	DWConv	Ele-wise
OPs	$H^2C_{in}C_{out}K^2$	$H^2C_{in}C_{out}K^2/G$	$H^2C_{in}K^2$	nH^2C_{in}
mac	$H^2C_{in}+H^2C_{out}$ $+K^2C_{in}C_{out}$	$H^2C_{in}+H^2C_{out}$ $+K^2C_{in}C_{out}/G$	$2H^2C_{in}$ $+K^2C_{in}$	H^2C_{in}

3.3 Measurement tool

Fig. 3 shows the high-level working process of our measurement tool. For each configuration in the dataset, it first generates a single-layer model graph in TensorFlow *protobuf* format, which is generally-supported by every NN platform. The graph is then converted to the format and data precision *e.g.,* INT8 of the target platform by invoking the platform's conversion API such as SNPE's *snpe-tensorflow-to-dlc*. The generated graph is then pushed to the target platform for latency, energy, and accuracy measurement.

It is non-trivial to accurately profile latency for a single layer on all platforms. The goal of the single layer measurement is for NN design. **The measured latency of a single layer should be the same as when it is within a complete model**. For AI accelerators, there is normally a host-accelerator data transfer before and after the inference execution. This cost needs to be excluded in the single layer profiling. However, not every framework supports fine-grained timing. For example, TFLite only provides an end-to-end latency on mobile GPU and Edge TPU, which includes the operator/block execution time, as well as the data layout conversion and transfer time. We implement operator-level profiling in TFLite for mobile GPU. For the close-source TFLite backend for Edge TPU, we pile the same operator/block into two multi-layer models, and use their latency difference to calculate a single layer's latency.

More implementation details on latency, accuracy, and en-

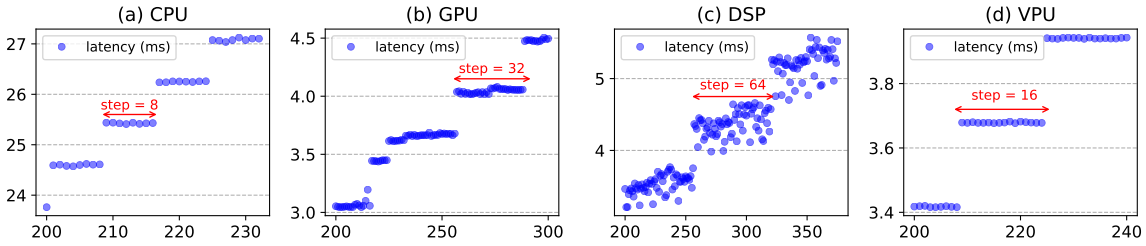

Figure 4. The latency of Conv shows a step/staircase pattern with C_{out}. Configuration: $H \times W = 28 \times 28$, $C_{in} = 320$, $K = 3$, $S = 1$. The X-axis is C_{out} (different intervals to better show the pattern) while the Y-axis is the latency in milliseconds.

ergy measurement are in Appendix A.

OPs and memory access calculation. We are particularly interested in the behaviour characteristics that show an inconsistent latency response to the number of computation operations (OPs) and memory accesses (mac). Table 3 shows formulas to calculate them for typical operators. Clearly, Conv has the highest data reuse rate (*i.e.,* operational intensity, calculated as OPs/mac), and thus is the most computation-intensive. For Group Conv, filters are divided by groups (G) to reduce OPs, and thus the data reuse is reduced. DWConv can be assumed as $G = C_{in}$. Its data reuse is dozens of times less than Conv. Element-wise operators and activation functions execute computation operations to each element of a tensor. They are the most memory-intensive operators, very sensitive to hardware memory bandwidth.

We profile the whole dataset on all platforms. For each design dimension, only one example set of hyperparameter configurations will be analyzed in Sec. 4. For other configurations, the basic tendencies are similar but certainly with variations. For data precision, unless the quantization analysis, FP32 data is used for the CPU as a baseline. The lowest precision data is used for other platforms to show their best performance.

4 NN Behaviour Characteristics

Out of the many findings, we highlight one finding for each major design dimension [2]. For each finding, the three research questions will be answered: (1) the characteristic that shows an inconsistent latency response to OPs and mac change of a configuration; (2) the reason for this characteristic; and (3) the implication for NN design.

4.1 Do more Conv channels increase latency?

The number of channels is an important hyperparameter to tune for efficient-NN design. This section analyzes the latency response to C_{out}. When other hyperparameters are fixed and only C_{out} varies, the computation and memory complexity of a Conv operator is linear $O(C_{out})$ (refer to Table 3 for calculation formulas). Therefore, the measured latency should also have a linear relationship with C_{out}. However, the measured Conv latency is against this expectation. Fig. 4 shows the latency responses on four of our platforms. Results for other platforms are in Appendix B.1.

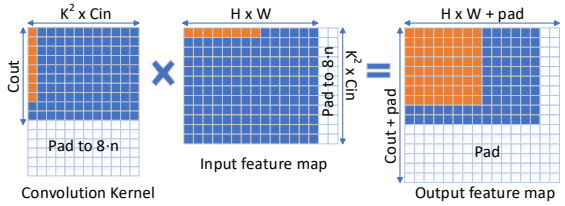

Figure 5. Matrix multiplication of TFLite *ruy*. The orange color shows the elementary block. Data is padded to be a multiple of 8.

Finding 1. The latency of Conv increases in a step pattern rather than linear with the number of output channels on the NN platforms, except for KPU.
Cause: The input tensors are padded to fully utilize the hardware data-level parallelism.
Implication: For potential higher accuracy, it is encouraged to keep the largest number of channels in each latency step in the NN design space and skip the other ones.

Hardware processors employ vectorization/matrix computing units to accelerate tensor computation. To fully utilize these units, the corresponding software frameworks pad the input tensors accordingly, which results in the step-pattern latency response.

We will take the CPU and DSP results as examples to elaborate this phenomenon since their frameworks are open source (GPU analysis is provided in Appendix B.1). For the other processors with closed-source frameworks, we expect that similar explanations apply too.

CPU. The evaluated inference framework on the CPU is TFLite v2.1. It uses the popular *im2col* (expand image

[2]Please refer to Appendix B for additional analysis for other design dimensions.

into a column matrix) (Vasudevan et al.) data transformation to convert Conv to matrix multiplication. TFLite's *ruy* library (TensorFlow, 2020) is then invoked for matrix multiplication execution.

To use all the 128-bit SIMD (*i.e.,* Neon) registers (ARM, 2019) on the ARM CPU, as shown in Fig. 5, *ruy* sets the elementary matrix multiplication block to be $(8,1) \times (1,8) \rightarrow (8,8)$ for FP32 data type ((x,y) denotes a $x \times y$ matrix). To adapt this elementary block, the two input matrices are padded to be a multiple of 8. For this reason, when other hyperparameters are fixed and only C_{out} varies, the latency shows a step pattern in the width of 8 as shown in Fig. 4(a).

DSP. To utilize the vector registers of 1024 bits width, the Hexagon NN library designs the data format as *Depth32* (Kuo, 2016), which sets the basic data block to be (1,4,32) for a 3D tensor (H,W,C). Each block is exactly $32 \times 4 \times 8\,bits = 1024\,bits$ for INT8 precision. To accommodate this data format, all the tensors are padded as a multiple of 4 in the width dimension W and a multiple of 32 in the channel dimension C. Two basic block inputs are packed into a pair and fed into two streams for execution. This is why Fig. 4(c) shows that the DSP latency has a 64-width step (32×2) as C_{out} increases.

For similar reasons, other platforms in Fig. 4 also show a step pattern. KPU shows a linear relationship between channel number and latency (refer to Fig. B.1). We assume the reason is that it calculates each channel one by one, and thus there is no need for padding.

The detailed analysis demonstrates again that understanding NN behaviour is costly, which demands deep knowledge in NN platforms. This highly motivates our characterization to fill the gap between NN design and the underlying platform. We also provide analysis for DWConv in Appendix B.2.

4.2 Does a building block have similar relative latency on different NN platforms?

NNs typically are composed of repeated building blocks. The selection of appropriate blocks is an essential design consideration. Intuitively, the relative latency of a building block should be similar on different platforms, since the OPs and mac are the same. However, the measured relative latency varies greatly on each platform and is mostly against the computation and memory complexity.

We pick four building blocks as examples and list their relative latencies on each platform, as well as their OPs and mac in Fig. 6(a). The values are all referenced to the ones of MobileNetV1Block on the same platform. Except for the CPU, the platforms have some unsupported building blocks, such as the ShuffleNetV2Block on the GPU and Edge TPU. Their results are thus missing in the figure.

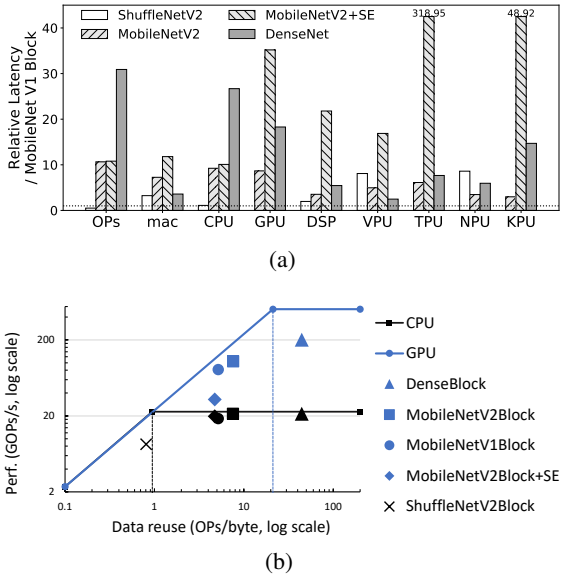

Figure 6. (a) Latency characteristics of blocks vary greatly on each processor. Configuration: $H \times W = 56 \times 56$, C_{in}=C_{out}=32, $K = 3$, $S = 1$. (b) Memory and computation rooflines for the CPU and GPU. The upper and lower markers for each block are the measured performance on the GPU and CPU, respectively.

Finding 2. The relative latency of a building block varies greatly on different platforms. The non-Conv operators can largely increase latency on NN platforms except CPU.
Cause: The mismatch of computation and memory bandwidth is severe and the support for non-Conv operators is weak on the NN platforms except CPU.
Implication: It is encouraged to customize the set of candidate blocks in the NN design space for each platform.

Bandwidth impact. Fig. 6(a) shows that **only on the CPU, the relative latency and OPs of building blocks have a direct relationship.** The reason is explained in Fig. 6(b). The figure shows the ideal CPU and GPU performance rooflines (Williams et al., 2009) as well as the measured performance (OPs / latency) of each block. If the data reuse rate of a block is lower than the ridge point, the performance is bounded by memory bandwidth, otherwise it is bounded by computation bandwidth. Note that the CPU and GPU we use are on the same SoC and share the memory, so the memory bounds in the figure are aligned.

On the single-core CPU (black color in the figure), since the memory and computation bandwidth are similar, the data reuse rate at the ridge point is low. Except for the ShuffleNetV2Block, the measured blocks are all computation bound, and the latency is directly related to OPs. By comparison, the mismatch of computation and memory bandwidth on GPU is bigger. Blocks with lower data reuse like MobileNetV2Block+SE become memory bound, which

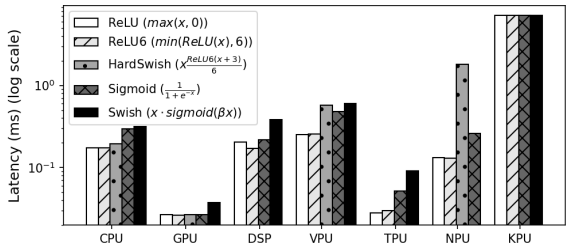

Figure 7. The latency of DWConv+activation functions. The latency of DWConv+ReLU (white bar) equals the DWConv latency. Configuration $H = W = 28$, $C_{in} = C_{out} = 96$, $K = 3$, $S = 1$.

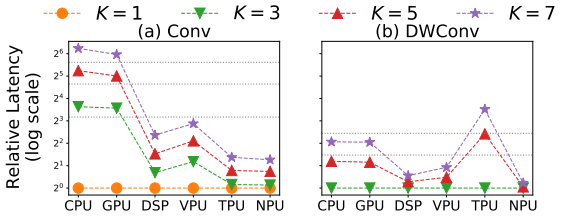

Figure 8. Relative latency of (a) Conv (reference to K=1), and (b) DWConv (reference to K=3) increases much less than OPs (marked by black dash lines) on accelerators. Configuration: $H = W = 56$, $C_{in} = C_{out} = 32$, $K = 3$, $S = 1$.

restrains the performance. The bandwidth mismatch on NN accelerators is more serious. Thus, the relative latency of low-data-reuse blocks increases on these accelerators. For example, MobileNetV2Block runs 1.89× faster than DenseBlock on the CPU due to much less OPs, 1.13× faster on the GPU, but only 0.25× faster on the Edge TPU.

Weak non-Conv operator support. Bandwidth difference is not the only reason for various relative latencies on the platforms. For example, adding SE to MobileNetV2Block only increases mac by 64% and OPs by 1.4%, but **the use of SE dramatically increases the latency except on the CPU** as shown in Fig. 6(a). The latency of MobileNetV2Block+SE is 3× longer than MobileNetV2Block on the GPU, 2× on the VPU, 5× on the DSP, 51× on the Edge TPU, and 15× on the KPU. This is due to the weak support for non-Conv operators by platforms.

As discussed in Sec. 2.2, operator fusion can greatly reduce memory accesses and so does latency. The fusion is implemented either in frameworks or hardware accelerators (refer to Sec. 3.2). However, they are normally behind NN design innovations. Current fusion is generally implemented for traditional Conv-related structures like Conv+Element-wise, but fails for new structures like the SE block.

Therefore, non-Conv operators of new structures such as Global Pooling and tensor-vector Multiplication in SE (Fig. 2(g)) become latency bottlenecks. For example, Global Pooling takes 71.7% of the latency of MobileNetV2Block+SE on the GPU. Neither the Global Pooling nor Multiplication can run on the KPU. They fall back to run on the RISC-V CPU on the SoC, so the latency is extremely high. The Edge TPU team also mentions that SE is suboptimal on Edge TPU (Andrew Howard, 2019).

4.3 Do activation functions barely increase latency?

A popular belief is that activation functions have marginal impact on model latency. The reason is that Conv/DWConv + activation function is a common structure. For traditional activation function like ReLU, platforms support operator fusion for this structure and thus the cost of activation functions is negligible.

However, the belief does not hold for more recent activation functions. Fig. 7 shows that the latency impact of some activation functions can be significant, and the impact varies greatly on different platforms. We evaluate the structure of DWConv + activation function rather than the activation function itself, since it is normally fused with Conv/DWConv. The latency of DWConv+ReLU (white bar) in the figure basically equals the latency of DWConv. (Missing data is due to the lack of support on platforms.)

Finding 3. Only ReLU and ReLU6 have negligible latency impact on every platform. HardSwish has negligible impact only on the CPU and GPU. Swish largely increases latency on every platform except for the KPU.
Cause: Inference platforms have poor operator-fusion support for novel activation functions.
Implication: It is encouraged to remove the costly activation functions from the NN design space for each platform.

The legend in Fig. 7 shows the computation formula of each activation function. The comparison of OPs is thus ReLU < ReLU6 < HardSwish < Sigmoid < Swish. **Only on the CPU, the latency of activation functions is consistent with the OPs**. On the KPU, the activation functions all have similar latency because each activation function is actually approximated by a piecewise cubic spline and then executed on the same Conv + BN + activation hardware pipeline. Other than the CPU and KPU, the obvious latency increases by activation functions on other platforms are all due to the failure of fusion with DWConv. HardSwish on the NPU is extremely slow (13.7×) since it is split into multiple non-fused operators by its framework. Swish cannot be fused on any platforms in this paper.

4.4 Does smaller Conv kernel size reduce latency?

The general belief is that the increase of Conv kernel size K can largely increase latency. This is because when other hyperparameters are fixed, the computation complexity of Conv and DWConv is $O(K^2)$ (refer to Table 3). Thus, for efficient NN design, the kernel size is normally set as 1×1

for Conv, and 3×3 for DWConv in a block to reduce latency (refer to Fig. 2). However, the measured results in Fig. 8 show that except for the CPU and GPU, the latency increase is much smaller than the OPs, particularly on the NPU, DSP, and VPU for DWConv, and NPU and Edge TPU for Conv.

Finding 4. As kernel size increases, the Conv latency increases much less on the other platforms than the CPU and GPU, except for DWConv on the Edge TPU.
Cause: The Conv and DWConv are memory bound on the platforms except for the CPU and GPU.
Implication: It is encouraged to use bigger kernels rather than just the smallest one on the NPU, DSP, and VPU for DWConv, as well as NPU, DSP, and Edge TPU for Conv.

Conv is computation bound on the CPU and GPU. Thus, the latency increase is consistent with OPs increase. By comparison, the latency increase on the other platforms is much less than OPs because Conv is memory bound rather than computation bound on these platforms. Compared to the quadratic increase of OPs, mac only increases 4% for K=3 and 12% for K=5 compared to K=1. Thus, the latency increase is much less compared to the CPU and GPU.

For DWConv, the latency increase with kernel size is even smaller compared to Conv, particularly on the DSP, VPU, and NPU. This is because the mac increase is more marginal, only 0.4% for K=5 compared to K=3. The behaviour on the Edge TPU is an outlier. The reason is not clear since both the hardware and framework are black box.

4.5 Does low data precision reduce latency with marginal accuracy loss?

Low data precision is recognized as an effective way to speed up inference with marginal accuracy loss. However, measured latency and accuracy do not always meet the expectation. Table 4.4 compares the inference latency and accuracy on the ImageNet 2012 dataset (shown in parentheses) of different precisions on the CPU, GPU, and NPU, since they support multiple precisions. The table only shows the models in our dataset supported by NPU (except for ShuffleNetV1). A complete result on each platform for every model is in Appendix B.4.

Finding 5. The use of INT8 on the NPU achieves $> 11\times$ speedup compared to FP16 except for MobileNetV3 ($2.5\times$), while on the CPU, it shows $< 3.6\times$ speedup and even a slowdown for ShuffleNetV1.
Cause: The NPU features an engine specific for INT8 computation, rather than sharing computation units for different precisions like the CPU and GPU. The INT8 requantization is very costly for low-data-reuse operators on the CPU.
Implication: For optimal latency on the CPU, it is encouraged to select suitable data precision for each operator according to its data reuse.

SIMD units on the CPU support 4 FP32 or 16 INT8 operations at a time. Ideally, INT8 can achieve $4\times$ speedup compared to FP32. However, the speedup is lower than that mainly for two reasons. The first reason is the computation implementation, which is hard to fully utilize all the SIMD units. The second reason is the cost of requantization. To avoid overflow, the output tensor of INT8 operators is in INT32. Requantization is needed to map the INT32 tensor to the range of INT8 (Krishnamoorthi, 2018). For high data-reuse operators, this cost is amortized. For low data-reuse operators, this cost can outweigh the performance gain of quantization. For example, the Add operator in INT8 can run $4\times$ slower than FP32. Thus, MobileNetV3 with SE achieves only $1.54\times$ speedup in INT8 compared to FP32. For ShuffleNetV1, INT8 even increases the latency by 13%.

On the Adreno GPU, the FP16 and FP32 computation also share the same units. Ideally, FP16 can achieve $2\times$ speedup compared to FP32. The output tensor of FP16 computation is still in FP16, so no requantization is needed. The speedup of FP16 is $1.5\times$ to $2.0\times$ compared to FP32.

On the NPU, the use of INT8 achieves $>11\times$ speedup than FP16. This is due to the exceptional 8-bit computation performance of the specific NN engine (refer to Sec. 3.2). Global Pooling and Multiplication in SE are not supported by this engine, so the speedup for MobileNetV3 is only $2.5\times$. The use of DFP16 accelerates model inference by $1.31\times$ to $3\times$ than FP16, due to the better performance of fixed-point 16-bit than float-point 16-bit computation.

For accuracy, we find that INT8 can cause unexpected accuracy loss. For example, Table 4.4 shows a big accuracy drop for MobileNetV3 on the CPU (-58.4%) and NPU (-75.6%). Moreover, we also observe some dramatic accuracy drop on other platforms. The accuracy of MnasNet-A1 is reduced by 64.7% on the Edge TPU. MobileNetV2 has a 70% accuracy drop on the DSP. These are because the ReLU6 and Swish activation functions in these models are not well supported by the quantization algorithms of the frameworks on the Edge TPU and DSP (Liu, 2020; Sheng et al., 2018).

Finding 6. INT8 can dramatically decrease inference accuracy of various models.
Cause: The INT8 quantization algorithms are not robust for every model on some NN platforms.
Implication: If the target data precision is INT8, it is necessary to remove building blocks and operators that will cause accuracy loss from the NN design space.

4.6 An across-platform comparison in latency, energy, and accuracy

We finally conduct an comparison among the seven platforms to see which is the best choice in terms of inference latency, energy, and model accuracy.

Table 4. Latency and ImageNet accuracy comparison of different data precision. INT8 significantly reduces accuracy for MobileNetV3 and ShuffleNetV1. In the form "x(*y*)", "x" means the latency and "*y*" means the accuracy after quantization for pre-trained models.

Model	OPs	Top1-Acc (%)	CPU		GPU		NPU			
			FP32 ms	INT8 ms (%)	FP32 ms	FP16 ms (%)	FP16 ms (%)	DFP16 ms (%)	DFP8 ms (%)	UINT8 ms (%)
MobileNetV1	569M	71.0	36.2	13.5 (*70.4*)	12.1	5.9 (*71.6*)	70.2 (*70.9*)	53.3 (*71.0*)	7.8 (*50.5*)	5.7 (*65.9*)
MobileNetV2	300M	71.8	24.9	12.5 (*71.2*)	9.9	6.2 (*71.8*)	72.7 (*71.8*)	49.5 (*71.8*)	8.8 (*59.1*)	6.2 (*69.3*)
MobileNetV3	219M	75.8	19.9	12.9 (*17.4*)	10.1	6.5 (*75.4*)	105.2 (*75.3*)	72.5 (*75.3*)	39.7 (*0.2*)	41.5 (*0.44*)
ShuffleNetV1	140M	67.6	31.7	35.8 (*0.10*)	-	-	-	-	-	-
InceptionV1	1448M	69.8	89.1	24.9 (*70.3*)	28.8	19.8 (*70.6*)	103.2 (*69.7*)	34.0 (*69.7*)	5.3 (*66.2*)	5.4 (*69.2*)
ResNetV1_50	3485M	75.1	208.5	63.7 (*74.7*)	53.2	33.3 (*75.1*)	348.6 (*75.1*)	145.6 (*75.0*)	13.5 (*72.9*)	13.4 (*74.7*)
ResNetV2_50	6548M	75.9	378.9	136.3 (*75.0*)	120.2	68.9 (*75.8*)	491.6 (*75.5*)	290.5 (*75.5*)	45.1 (*72.0*)	42.7 (*75.2*)

The CPU provides the most robust support for models. By comparison, all other platforms have operators not supported or not well performed, which harms model accuracy.

However, for models that NN platforms provide good support for, substantial improvement in latency and energy can be achieved. The best speedup compared to the CPU in our evaluation is 25× for InceptionV1 on the Edge TPU and ResNetV1 on the NPU. The most energy saving is MobileNetV1 on the Edge TPU and ResNetV1 on the NPU which cost only 4% and 3% of the energy on the CPU, respectively. All the data of the models is in Appendix B.4.

Finding 7. Considering the robust support for NN models, particularly the novel ones, the CPU is the best choice. Considering the latency and energy cost for basic popular NN models, Edge TPU and the NPU are the best choice.

Finally, we raise **a contradiction between the design of edge NN platform and efficient NN algorithm** exposed during our study. Current edge NN platforms are mostly designed for computation-intensive convolution operators. However, the trend in efficient NN design is to introduce more memory-intensive blocks *e.g.,* Shuffle and SE. These blocks become the latency bottlenecks, which shows an opportunity for future platform design.

5 Case Studies on NN Design

This section utilizes channel pruning and NAS as examples to show how the revealed findings can improve the NN design process.

Channel pruning. The layer-wise pruning ratio is difficult to set in channel pruning. This process requires huge time cost to achieve the optimal accuracy and efficiency trade-off. Findings in Sec. 4.1 indicate that the decrease in C_{out} does not always decrease latency. We can thus accelerate the pruning process by keeping the largest number of channels in each latency step for better accuracy, and skip others.

For instance, MetaPruning (Liu et al., 2019) searches for C_{out} from $[int(0.1 \times C^l_{out}), C^l_{out}]$ for layer l of MobileNetV1, with the step of $int(0.03 \times C^l_{out})$, where C^l_{out} indicates the original output channel number for layer l. For a layer with $C_{out} = 32$, the original step size is 1 and the candidate channel selections are 30 (ranging from 3 to 32). The total search space contains 30^{14} channel configurations for MobileNetV1 (14 layers to search), which is huge to explore. Fortunately, findings show that the latency of C_{out} decreases with a step width of 8 for Conv on the CPU (c.f. Fig 4(a)). The candidate channel selections are reduced from 30 to 4 (*i.e.,* 8, 16, 24, 32), and the search space is reduced from 30^{14} to 4^{14}. In total, we reduce the channel selections by 7.5× for each layer, and accelerate MetaPruning with a 10^{12}× search space reduction for MobileNetV1.

Hardware-aware NAS. The unawareness of hardware diversity misleads current NAS works to apply an identical manually elaborated search space based on CPUs for all platforms (Cai et al., 2019; Wu et al., 2019a). Findings in Sec. 4 expose the inefficiencies in such search space and suggest new principles.

Recent hardware-aware NAS methods adopt a layer-level hierarchical search space, where each layer searches for the optimal operator/block from several block choices (*e.g.,* MobileNets block variants with different DWConv kernel sizes). The search space is usually huge and expensive. For example, the search space size of MnasNet (Tan et al., 2019) is 10^{13}, and the search cost is 40,000 GPU hours. As discussed in Sec. 4.2, SE dramatically increases the latency on all other hardware except the CPU. As a result, we suggest removing SE from MnasNet's search space for AI accelerators. It can reduce the search space size by 32×.

For future search space design, we summarize the following guidelines: *(i)* The NN search space needs to be customized for each hardware; *(ii)* CPU prefers blocks with less computations, while AI accelerators prefer blocks with less memory accesses; *(iii)* Novel block design should consider the platform optimizations (*e.g.,* operator fusion) and robustness of quantization.

6 Conclusion

To summarize, we propose a dataset which covers major NN design dimensions. By profiling this dataset on seven representative edge platforms, we highlight seven findings as guidelines to improve efficient NN design. Case study shows that the design space can be largely reduced for channel pruning and NAS by our findings.

REFERENCES

Andrew Howard, S. G. Introducing the next generation of on-device vision models: Mobilenetv3 and mobilenetedgetpu, 2019. URL https://ai.googleblog.com/2019/11/introducing-next-generation-on-device.html.

ARM. Armv8-a instruction set architecture, 2019. URL https://developer.arm.com/architectures/learn-the-architecture/armv8-a-instruction-set-architecture.

Baidu. Benchmarking deep learning operations on different hardware, 2020. URL https://github.com/baidu-research/DeepBench.

BDTi. Qualcomm's qdsp6 v6: Imaging and vision enhancements via vector extensions, 2015. URL https://www.bdti.com/InsideDSP/2015/09/30/Qualcomm.

Bhat, K. A tool which profiles opencl devices to find their peak capacities, 2020. URL https://github.com/krrishnarraj/clpeak.

Bianco, S., Cadène, R., Celona, L., and Napoletano, P. Benchmark analysis of representative deep neural network architectures. In *IEEE Access*, volume 6, 2018.

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

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

Dai, X., Zhang, P., Wu, B., Yin, H., Sun, F., Wang, Y., Dukhan, M., Hu, Y., Wu, Y., Jia, Y., et al. Chamnet: Towards efficient network design through platform-aware model adaptation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

Gao, W., Tang, F., Zhan, J., Lan, C., Luo, C., Wang, L., Dai, J., Cao, Z., Xiong, X., Jiang, Z., Hao, T., Fan, F., Wen, X., Zhang, F., Huang, Y., Chen, J., Du, M., Ren, R., Zheng, C., Zheng, D., Tang, H., Zhan, K., Wang, B., Kong, D., Yu, M., Tan, C., Li, H., Tian, X., Li, Y., Lu, G., Shao, J., Wang, Z., Wang, X., and Ye, H. Aibench: An agile domain-specific benchmarking methodology and an ai benchmark suite. 2020.

Google. Edge tpu, 2020a. URL https://cloud.google.com/edge-tpu/.

Google. Tensorflow lite: Deploy machine learning models on mobile and iot devices, 2020b. URL https://www.tensorflow.org/lite.

Hadidi, R., Cao, J., Xie, Y., Asgari, B., Krishna, T., and Kim, H. Characterizing the deployment of deep neural networks on commercial edge devices. In *IISWC*, pp. 35–48. IEEE, 2019.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016.

He, Y., Zhang, X., and Sun, J. Channel pruning for accelerating very deep neural networks. In *The IEEE International Conference on Computer Vision (ICCV)*, 2017.

He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J., and Han, S. Amc: Automl for model compression and acceleration on mobile devices. In *European Conference on Computer Vision (ECCV)*, 2018.

Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q. V., and Adam, H. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. 2017.

Huang, G., Liu, Z., and van der Maaten, L. Densely connected convolutional networks. Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR), 2017.

Intel. Openvino deploy high-performance, deep learning inference, 2020. URL https://software.intel.com/en-us/openvino-toolkit.

Kendryte. Kendryte k210, 2020a. URL https://kendryte.com/#products.

Kendryte. Open deep learning compiler stack for kendryte k210 ai accelerator, 2020b. URL https://github.com/kendryte/nncase.

Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. arXiv preprint, arXiv:1806.08342, 2018.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc., 2012.

Kuo, R. Qualcomm hexagon nn offload framework, 2016. URL https://wiki.codeaurora.org/xwiki/bin/+Qualcomm+Hexagon+NN+Offload+Framework/.

Lee, J., Chirkov, N., Ignasheva, E., Pisarchyk, Y., Shieh, M., Riccardi, F., Sarokin, R., Kulik, A., and Grundmann, M. On-device neural net inference with mobile gpus. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2019.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets. The International Conference on Learning Representations (ICLR), 2017.

Liu, R. Higher accuracy on vision models with efficientnet-lite. Tensorflow Blog, 2020.

Liu, Z., Mu, H., Zhang, X., Guo, Z., Xin Yang, T. K.-T. C., and Sun, J. Metapruning: Meta learning for automatic neural architecture channel pruning. Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2019.

Ma, N., Zhang, X., Zheng, H.-T., and Sun, J. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

Movidius, I. Ultimate performance at ultra-low powerintel movidius myriad x vpu, 2020. URL https://www.movidius.com/myriadx.

Qi, H., Sparks, E. R., and Talwalkar, A. Paleo: A performance model for deep neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

Qualcomm. Snapdragon neural processing engine sdk, 2020. URL https://developer.qualcomm.com/docs/snpe/overview.html.

Reddi, V. J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.-J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., Chukka, R., Coleman, C., Davis, S., Deng, P., Diamos, G., Duke, J., Fick, D., Gardner, J. S., Hubara, I., Idgunji, S., Jablin, T. B., Jiao, J., John, T. S., Kanwar, P., Lee, D., Liao, J., Lokhmotov, A., Massa, F., Meng, P., Micikevicius, P., Osborne, C., Pekhimenko, G., Rajan, A. T. R., Sequeira, D., Sirasao, A., Sun, F., Tang, H., Thomson, M., Wei, F., Wu, E., Xu, L., Yamada, K., Yu, B., Yuan, G., Zhong, A., Zhang, P., and Zhou, Y. Mlperf inference benchmark. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

Rockchip. Rockchip rk3399pro trm, 2019. URL http://rockchip.fr/Rockchip%20RK3399Pro%20TRM%20V1.0%20Part1.pdf.

Rockchip. High performance ai development platform, 2020a. URL http://t.rock-chips.com/en/.

Rockchip. Rknn toolkit, 2020b. URL https://github.com/rockchip-linux/rknn-toolkit.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2018.

Sheng, T., Feng, C., Zhuo, S., Zhang, X., Shen, L., and Aleksic, M. A quantization-friendly separable convolution for mobilenets. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, 2018.

Simonyan, K. and Zisserman, A. Veep deep convolutional networks for large-scale image recognition. 2014.

Suyog Gupta, M. T. Efficientnet-edgetpu: Creating accelerator-optimized neural networks with automl, 2019. URL https://ai.googleblog.com/2019/08/efficientnet-edgetpu-creating.html.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2015.

Tan, M. and Le, Q. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pp. 6105–6114, 2019.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

Tao, J.-H., Du, Z.-D., Guo, Q., Lan, H.-Y., Zhang, L., Zhou, S.-Y., Liu, C., Liu, H.-F., Tang, S., Rush, A., Chen, W., Liu, S.-L., Chen, Y.-J., and Chen, T.-S. Benchip: Benchmarking intelligence processors. *Journal of Computer Science and Technology*, 33, 2018.

TensorFlow, G. The ruy matrix multiplication library, 2020. URL https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/experimental/ruy.

Turner, J., Cano, J., Radu, V., Crowley, E. J., O'Boyle, M. F. P., and Storkey, A. J. Characterising across-stack optimisations for deep convolutional neural networks. In *IISWC*, pp. 101–110. IEEE Computer Society, 2018.

Vasudevan, A., Anderson, A., and Gregg, D. Parallel multi channel convolution using general matrix multiplication. In *The 28th Annual IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*.

Wang, Y., Wei, G., and Brooks, D. A systematic methodology for analysis of deep learning hardware and software platforms. In *Proceedings of Machine Learning and Systems (MLSys)*, 2020a.

Wang, Y. E., Wei, G.-Y., and Brooks, D. A systematic methodology for analysis of deep learning hardware and software platforms. In *Proceedings of Machine Learning and Systems (MLSys)*, 2020b.

WikiChip. Shave v2.0 - microarchitectures - intel movidius, 2018. URL https://en.wikichip.org/w/index.php?title=movidius%2Fmicroarchitectures%2Fshave_v2.0.

Williams, S., Waterman, A., and Patterson, D. A. Roofline: An Insightful Visual Performance Model for Multicore Architectures, April 2009. URL https://doi.org/10.1145/1498765.1498785.

Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., Tian, Y., Vajda, P., Jia, Y., and Keutzer, K. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019a.

Wu, C.-J., Brooks, D., Chen, K., Chen, D., Choudhury, S., Dukhan, M., Hazelwood, K., Isaac, E., Jia, Y., Jia, B., Leyvand, T., Lun, H., Lu, Y., Qiao, L., Reagen, B., Spisak, J., Sun, F., Tulloch, A., Vajda, P., Wang, X., Wang, Y., Wasti, B., Wu, Y., Xian, R., Yoo, S., and Zhang, P. Machine learning at facebook: Understanding inference at the edge. IEEE International Symposium on High Performance Computer Architecture (HPCA), 2019b.

Xie, X., Hu, X., Gu, P., Li, S., Ji, Y., and Xie, Y. Nnbench-x: Benchmarking and understanding neural network workloads for accelerator designs. IEEE Computer Architecture Letters, 2019.

Xu, M., Liu, J., Liu, Y., Lin, F. X., Liu, Y., and Liu, X. A first look at deep learning apps on smartphones. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 2019. doi: 10.1145/3308558.3313591.

Yang, T.-J., Chen, Y.-H., and Sze, V. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

Yang, T.-J., Howard, A., Chen, B., Zhang, X., Go, A., Sandler, M., Sze, V., and Adam, H. Netadapt: Platform-aware neural network adaptation for mobile applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

Zhang, L. L., Yang, Y., Jiang, Y., Zhu, W., and Liu, Y. Fast hardware-aware neural architecture search. In *Proceedings of the IEEE Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2020.

Zhang, W., Wei, W., Xu, L., Jin, L., and Li, C. Ai matrix: A deep learning benchmark for alibaba data centers, 2019.

Zhang, X., Zhou, X., Lin, M., and Sun, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2018.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

A MEASUREMENT SETTINGS

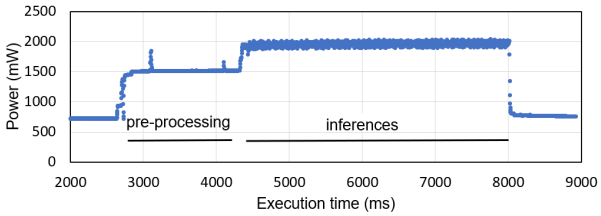

Figure A.1. Power curve of MobileNetV2 execution on VPU, including the pre-processing and 140 inference runs.

Latency measurement. TFLite currently doesn't provide operator-level profiling for accelerators. For GPU backend, we implement an operator profiler by utilizing the OpenCL profiling events, and record the GPU latency of operators and data transfer separately. TFLite has both OpenCL and OpenGL implementations for GPU backend. This paper picks the OpenCL one due to its better performance. Since the TFLite backend for Edge TPU is closed-source and only report end-to-end latency, we pile the same operator/block into two multi-layer models *e.g.,* a 40-layer one and a 10-layer one. The latency is calculated by the latency difference of two models divided by their depth difference.

Except for TFLite, the operator-level profiling of each platform reports the data transfer and operator cost separately. We directly use the reported operator latency. Block latency is calculated as the sum of its constituent operators, due to the serial execution of operators in these frameworks.

For whole model latency, we disable the operator-level profiling and report the end-to-end time cost as in real industry deployment. The reported latency in the paper is the arithmetic mean of 50 runs after 10 warmup runs. The 95% confidence interval is within 2%.

Accuracy evaluation. We evaluate the model accuracy on ImageNet 2012. Pre-processing includes cropping, normalization, and resizing, which are necessary steps before model inference. This is done by TFLite model accuracy tool. Finally, the pre-processed data are used as model inputs for each framework to test the inference accuracy.

Energy measurement. We monitor power trace to calculate energy consumption. For our compute stick devices (*i.e.,* VPU, Edge TPU, NPU, KPU), power trace is sampled by a calibrated power meter at 1 KHz. For android devices (*i.e.,* CPU, GPU, DSP), power trace is collected through system PMIC (power management integrated circuit) at around 800 MHz.

As illustrated previously, model execution time includes not only inference time but also initialization time, data layout conversion time, etc. It's challenging to align the start time of power sampling and inference on our platforms. To locate the inference time, we run hundreds of sequential inference runs. As shown in Fig. A.1, we observe that power fluctuations ahead of actual inference execution, and then shows stable repeating patterns until inference finishes. Therefore, we use the average power of inference stage to compute the energy (*i.e.,* $energy = power \times time$).

Frequency setting. For measurement stability, we disable the OS DVFS and try to fix the processor frequency. We set he CPU frequency to the highest 2.84 GHz. Edge TPU uses the runtime version that operates at the max frequency. KPU is set to 400 MHz although it can be overclocked to 800 MHz. The GPU also provides frequency setting interface for users. However, its DVFS cannot be disabled and still scales frequency according to the workload. We sample the GPU frequency during each inference run. Results show that the GPU frequency is stable at 585 MHz. No available methods to set frequency for the other processors.

B ADDITIONAL RESULTS FOR NN DESIGN SPACE

B.1 Conv output channel numbers analysis

Fig. B.1 shows the latency of Conv with different output channel numbers on other three platforms. Except KPU, the latency shows a step pattern with C_{out} on TPU and NPU.

GPU. In this paper, we utilize TFLite's OpenCL backend on the Adreno GPU for accurate latency measurement. The OpenCL execution model decomposes an index space into *work groups* (*i.e., blocks* in CUDA) (Lee et al., 2019). For Conv, the decomposition is done on the index space of the output tensor as shown in Fig. B.2. A work group is essentially the elementary computation block for GPU SIMT parallelism, since threads (called *work items* in OpenCL) in a work group execute the same shader code and use the same work-group barrier. The output tensor needs to be padded according to the work group size, and thus the latency also shows a step pattern in Fig. 4(b).

However, the step width for the GPU is not constant as the CPU. This is because the work group size is not fixed as the width of the vector registers. The suitable work group size depends on various hardware configurations, such as wave size, number of computing units, and shared memory size. As a result, TFLite conducts an exhaustive search for the best work group size according to the output tensor, which leads to the varied step width.

B.2 Depthwise Conv channel numbers analysis

DWConv is introduced for edge-regime NN models. Each filter only convolves with one input channel rather than all channels to largely reduce computation. The filter kernel number, therefore, equals input channel number *i.e.,* $C_{in} = C_{out}$. The computation complexity is $O(C_{in})$ when only

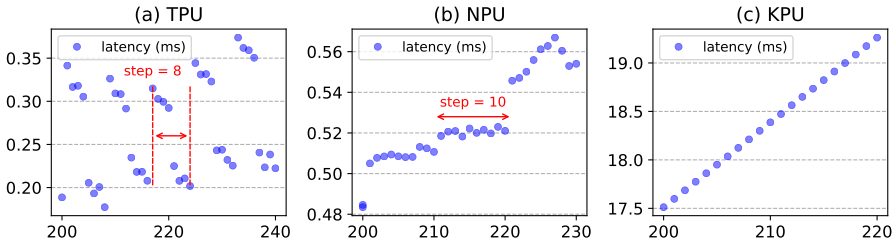

Figure B.1. The latency of Conv with different C_{out} on other three platforms. Configuration: $H \times W = 28 \times 28$, $C_{in} = 320$, $K = 3$, $S = 1$. The X-axis is C_{out} (different intervals to better show the pattern) while the Y-axis is the latency in milliseconds.

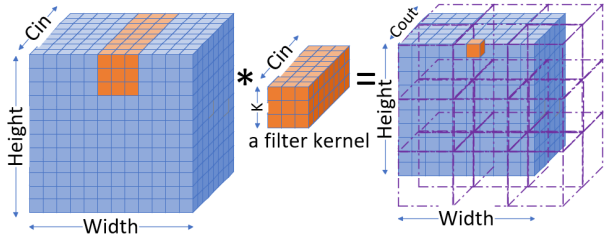

Figure B.2. TFLite OpenCL Conv implementation. Dash line marks work group and padding. Orange color shows the computation of one element in the result tensor.

C_{in} changes (Table 3). The latency has the following findings (figure not shown). Fig. B.3 shows the latency and channel number relationship for DWConv.

Finding 8. The latency of DWConv increases with the channel number in a step pattern for CPU, DSP, TPU and VPU; quadratically for NPU; and linearly for GPU, KPU.

For DWConv on CPU, TFLite uses direct Conv for each channel rather than the *im2col+MM* algorithm. This is because the low computation amount cannot amortize the data transformation cost. For the same reason, the input tensors are not padded either. Instead, TFLite uses three loops with different loop unrolling stride as shown in Fig. B.4. The first loop uses a stride = 16, *i.e.,* each loop iteration convolves 16 channels. The second loop has a stride = 4, and the third stride is 1. The 16 stride has better instruction parallelism, SIMD units utilization, cacheline alignment, and reduced branch cost. Thus, the latency shows a zig-zag pattern in Fig. B.3(a), and achieves the local minimum when the channel number is a multiple of 16.

As is shown in Fig. B.3(a), latency rises from $16k$ to $16k+12$ channels and then drops drastically when reaching $16(k+1)$. We read the corresponding code for depthwise Conv in TensorFlow Lite. And we found that the accumulation loop along the channel axis is manually unrolled to the template of $\lfloor \frac{c}{16} \rfloor \text{loop}_0 + \lfloor \frac{c \bmod 16}{4} \rfloor \text{loop}_1 + (c \bmod 4)\text{loop}_2$.

In this template, loop_0 uses 4×3 (cache for input, filter and result) 128-bit wide registers to operate $\frac{128\text{bit}}{32\text{bit/channel}} \times 4 = 16$ channels at a time. Similarly, loop_1 utilizes 3 NEON registers and Mul-Adds for 4 channels while loop_2 does the plainest operation without SIMD instructions.

Since correctly-implemented loop unrolls can minimize branch penalty and increase parallelism by reducing data dependencies between instructions, fully unrolled accumulation loop is faster than partially unrolled one. Therefore, the latency of loop_0 is smaller than 4 times the latency of loop_1, hence the rise-and-then-fall zig-zag latency diagram of DWConv.

The output feature map of DWConv on GPU is also partitioned into work groups as Conv in Fig. B.2. However, rather than a clear step due to padding, the latency here in Fig. B.3(b) fluctuates a bit. As explained above, the TFLite GPU backend searches for the best work group size. We thus output all the selected work group size to explore the reason. In Fig. 4(b), the selected work group size is relatively stable and the step is clear. However, the selected work group size here varies a lot as the channel number increases, and so thus the latency.

The Rockchip NPU in Fig. B.3(d) shows an unexpected quadratic increase although the complexity is only $O(Cin)$. A very possible explanation is that *the CNN-specific processing units of the NPU is customized only for normal Conv*, and new Conv algorithms have to be converted to normal Conv to run. We deduce this because the computation complexity of normal Conv is $O(C_{in}^2)$ when $C_{in} = C_{out}$ as DWConv, and the latency curve is quadratic. Thus, the quadratic increase for DWConv should be because it runs as a normal Conv on this NPU. To achieve this, each filter kernel should be zero padded to the size of C_{in} and then conducts Conv with the whole input feature map (hardware optimization for computations with zero may exist).

For the DSP, VPU and KPU, they basically have the same latency curve as normal Conv, and we don't include the figure here.

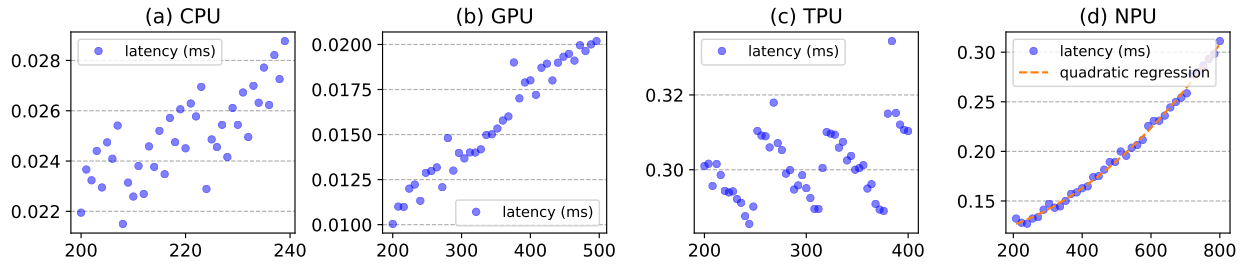

Figure B.3. The latency of DWConv at different channel number. Configuration: $H \times W$=7×7, C_{in}=C_{out}, K=3, S=1. The x-axis is C_{out}, y-axis is the latency in milliseconds.

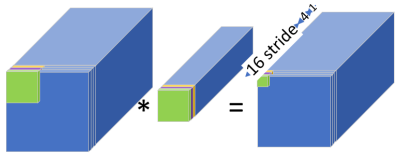

Figure B.4. TFLite CPU DWConv implementation. Color shows each filter only convolves one channel and C_{in}= C_{out}.

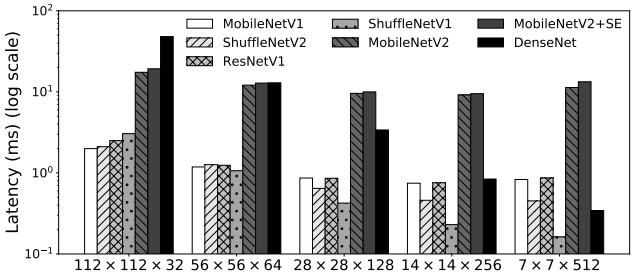

Figure B.5. The fastest block in each layer is different on the CPU. Configuration: the layers vary in $H \times W \times C_{in}$, $C_{in} = C_{out}$, $K = 3$, $S = 1$.

B.3 Input size impacts on latency analysis

Current NN designs normally use the same building block for every layer in a model, assuming one block can perform well on every layer. However, the measured latency in Fig. B.5 shows that there is no one block that runs the fastest in every layer on the CPU. The configurations of input size and number of channels in this evaluation follow the "half size, double channel" heuristic rule discussed in Sec. 2.1, and set $H = W$ and $C_{in} = C_{out}$ as it is the most common setting for CNN models.

Finding 9. No one block is the fastest in every layer on the CPU, while on the other platforms MobileNetV1Block is the fastest in every layer.
Cause: The OPs and mac of each block change differently with layers. The latency is consistent with the change on the CPU, while inconsistent on the other accelerators because of their weak support for non-Conv operators.
Implication: It is encouraged for NN design to select different building blocks for each layer on the CPU.

As shown in Fig. B.5, blocks have different latency response as the layer goes deeper (smaller input and more channels). The latency of ShuffleNetV1Block and DenseBlock keeps reducing with the layers, while the latency reduction stops for the other building blocks in deeper layers. This is because the OPs and mac of operators change differently with layers.

According to the formulas in Table 3, under the "half size double channel setting" rule, the OPs keep the same with layers for Conv and Group Conv (*i.e.,* $(1/2H)^2 \times (2C_{in})^2K^2$), and get halved for DWConv and Element-wise operators (*i.e.,* $(1/2H)^2 \times 2C_{in}K^2$). The mac of an operator is composed by input and output tensors and the filter kernels. The mac of input and output tensors gets halved with layers (*i.e.,* $(1/2H)^2 \times 2C_{in}$). The mac of filter kernels increases $4\times$ (*i.e.,* $(2C_{in})^2K^2$) for Conv and Group Conv, and $2\times$ (*i.e.,* $2C_{in}K^2$) for DWConv with layers.

For MobileNet blocks which are composed by Conv and DWConv (see Fig. 2), the mac reduction of input/output tensors outweighs the mac increase of filter kernels until the 14×14 layer, which has the minimum mac and latency as shown in Fig. B.5. This is also why the latency reduction stops for other building blocks in deeper layers. For ShuffleNetV1Block composed by Group Conv, DWConv, and Channel Shuffle, the mac of filer kernels is much smaller than the input/output tensors. The total mac keeps reducing with layers, so does the latency. ShuffleNetV2Block replaces the Group Conv in ShuffleNetV1Block by Conv, and the mac reduction is much smaller. For DenseBlock, albeit composed by Conv, the two Conv operators have fixed C_{out} or C_{in}. Hence, the OPs keep decreasing with layers, and so does the latency.

In total, MobileNetV1Block runs fastest in the first two layers, but beaten by ShuffleNetV1/V2Block afterwards. It even runs $> 1\times$ slower than the well-known computation-intensive DenseBlock at 7×7. ShuffleNetV1Block is the fastest for the 28×28, 14×14, and 7×7 layers.

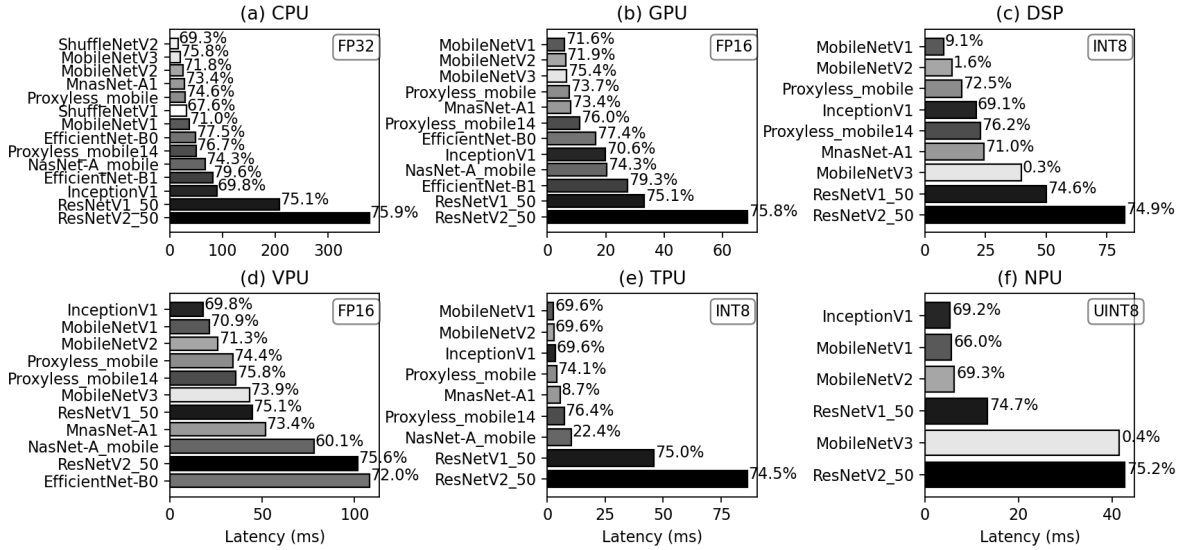

Figure B.6. Regarding the accuracy and latency trade-off, the optimal models are different on each platform. Top-1 accuracy on ImageNet is annotated on the bar. A darker color indicates a larger OPs.

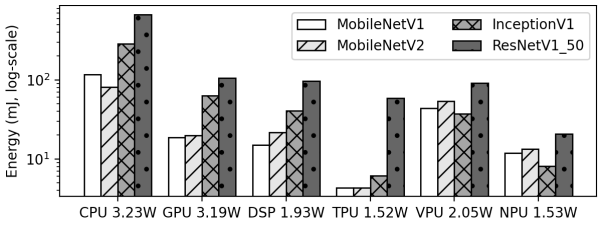

Figure B.7. Energy consumption of one inference run across different platforms. The number followed by each platform on the x-axis is the average power cost.

For other platforms, MobileNetV1Block is always the fastest in every layer due to their poor support for non-Conv operators like Shuffle as discussed in Sec. 4.2.

B.4 Model energy cost, accuracy, and latency on different platforms

For accuracy evaluation, our dataset collects 14 representative models from both manually-designed and NAS-searched ones: 1) Manually designed: MobileNetV1/V2, ShuffleNetV1/V2 for light-weight models, and ResNetV1/V2 (He et al., 2016), InceptionV1 (Szegedy et al., 2015) for large models; 2) NAS searched: MobileNetV3-Large1.0 (MobileNetV3) (Howard et al., 2019), Proxyless_mobile/mobile14 (Cai et al., 2019), MnasNet-A1 (Tan et al., 2019), NasNet-A_mobile (Zoph et al., 2018), and EfficientNet-B0/B1 (Tan & Le, 2019). These models are with different levels of computation and memory cost. OPs range from 140 M (ShuffleNetV1) to 6.55 G (ResNetV2). Model size varies from 9 MB (ShuffleNetV2) to 99 MB (ResNetV2).

Fig. B.6 shows the latency and accuracy of all the models in our dataset on every platform (the missing data is due to lack of support). We can see that every model can run successfully on the CPU with no accuracy loss. By comparison, the other platforms all have operators or models not supported or not well performed, especially for more recently-designed models. MobileNetV3 runs the fastest on the CPU due to the poor support of SE block on the other platforms, although it is only a single-core ARM CPU using FP32 precision. In terms of accuracy, MnasNet-A1 and NasNet-A_mobile on the Edge TPU, MobileNet series on the DSP, and MobileNetV3 on the NPU all experience dramatic accuracy loss.

Fig. B.7 compares the energy cost of four well-supported models to conduct one inference execution on every platform. The Edge TPU and NPU show the least power and energy cost. The energy of MobileNetV1 inference on the Edge TPU is only 4% of the energy cost on the CPU. The energy of ResNetV1 on the NPU is only 3% of the cost on the CPU.

The KPU result is missing in this section because the input shape of the models is over the KPU limitation. It can run blocks or shrank models but not default model settings.