CPU-GPU Layer-Switched Low Latency CNN Inference

Ehsan Aghapour, Dolly Sapra, Andy Pimentel, and Anuj Pathania
University of Amsterdam
e.aghapour@uva.nl, d.sapra@uva.nl, a.d.pimentel@uva.nl, a.pathania@uva.nl

***Abstract*—Convolutional Neural Networks (CNNs) inference on Heterogeneous Multi-Processor System-on-Chips (HMPSoCs) in edge devices represent cutting-edge embedded machine learning. Embedded CPU and GPU within an HMPSoC can both perform inference using CNNs. However, common practice is to run a CNN on the HMPSoC component (CPU or GPU) provides the best performance (lowest latency) for that CNN. CNNs are not monolithic and are composed of several layers of different types. Some of these layers have lower latency on the CPU, while others execute faster on the GPU. In this work, we investigate the reason behind this observation. We also propose an execution of CNN that switches between CPU and GPU at the layer granularity, wherein a CNN layer executes on the component that provides it with the lowest latency. Switching between the CPU and the GPU back and forth mid-inference introduces additional overhead (delay) in the inference. Regardless of overhead, we show in this work that a CPU-GPU layer switched execution results in, on average, having 4.72% lower CNN inference latency on the *Khadas VIM 3* board with *Amlogic A311D* HMPSoC.**

***Index Terms*—On-Chip Inference, Edge Computing**

I. Introduction

Pattern recognition problems originate in many embedded applications in various domains, such as autonomous driving [14], intelligent robotics [4], image classification [7], object detection [15], and semantic segmentation [19]. It is now commonplace to solve (inference) these problems using Convolutional Neural Networks (CNNs), known for their high accuracy in differentiating between patterns. The time-sensitive nature of these applications requires CNN inference to occur on edge devices that run the embedded applications themselves [16]. Heterogeneous Multi-Processor System-on-Chips (HMPSoC) powering the edge devices make on-device inference possible. However, CNN kernels within the embedded applications project significant resource requirements on the underlying HMPSoCs. Consequently, HMPSoCs often struggle to provide low latency embedded CNN inference needed for high-end embedded applications.

An HMPSoC tightly integrates an embedded CPU and a GPU on a single chip. Figure 1 shows an abstract block diagram for the state-of-the-art *Amlogic A311D* HMPSoC within the *Khadas Vim 3* embedded platform. It contains a Hexa-core *ARM big.Little* asymmetric multi-core CPU and a dual-core *Mali* GPU. The *ARM big.Little* CPU further consists of two CPU clusters – a high-performance, high-power quad-core *big* CPU cluster and a low-performance, low-power dual-core *Little* CPU cluster. CPU and GPU within the HMPSoC can both perform inferencing [22].

It is common in non-embedded platforms for GPUs to significantly outperform the CPUs in inference. However, in embedded platforms, the CPU and GPU performance is comparable. It is even possible for a CPU to outperform a GPU for some given CNNs. Consequently, CPUs are still relevant for inference in embedded platforms [24]. Figure 2 shows the latency of different CNNs on different HMPSoC components (CPU or GPU). The *big* CPU cluster always provides lower latency than the *small* CPU cluster. It is not feasible to use the *big* and *small* CPU clusters simultaneously to reduce latency [21]. Therefore, we limit ourselves to only the *big* CPU cluster in this work. When we mention the CPU again in this work, it refers to the *big* CPU cluster. Figure 2 shows that the CPU outperforms the GPU for *MobileNet* and *ResNet50*, while GPU outperforms the CPU for *AlexNet*, *GoogleNet*, and *SqueezeNet*.

It is common to run CNN kernels in an embedded application on the HMPSoC component (CPU or GPU) that provides the lowest latency. However, a CNN is not one monolithic execution block. A CNN comprises several layers that execute sequentially to generate an output from a given input. In this work, we observe that some of these layers execute faster on the CPU while the other layers execute faster on the GPU. To our knowledge, no one has made or explained this observation. Reducing the latency for a CNN inference appears straightforward by executing a given CNN layer on the HMPSoC component, where it performs the fastest. CPUs and GPUs, however, have different Instruction Set Architecture (ISA) and execution models. Therefore, achieving

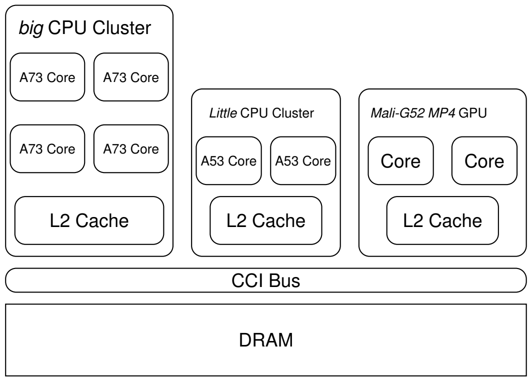


Fig. 1: An abstract block diagram of *Amlogic A311D* HMPSoC in *Khadas Vim 3* embedded platform.

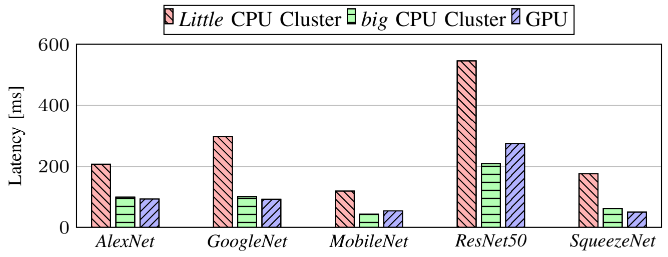


Fig. 2: The inference latency for different inference capable components for different CNNs on *Khadas Vim 3*.

TABLE I: Processing time of *AlexNet* layers on CPU and GPU. The last column shows the time with the best component.

Layer	CPU (ms)	GPU (ms)	Best (ms)
1	11.56	9.69	9.69
2	15.05	14.37	14.37
3	5.15	8.62	5.15
4	4.48	5.33	4.48
5	3.41	4.15	3.41
6	38.51	35.89	35.89
7	17.02	10.06	10.06
8	4.19	3.44	3.44
Total	99.38	91.55	86.50

a CPU-GPU layer switch execution in practice is technically challenging on a real platform.

Motivational Example: *AlexNet* is a popular CNN used for image classification. The *AlexNet* contains 11 layers – five convolution, three max-pooling, and three fully connected layers. Max-pooling layers are too small for us to measure and work with individually. Therefore, we club them with the preceding convolution layers to reduce the number of operatable layers in *AlexNet* to eight. Table I shows the split of *AlexNet* latency on the CPU and the GPU in terms of its layers. Layers 1, 2, 6, 7, and 8 execute faster on GPU. Layers 3, 4, and 5 execute faster on the CPU. Table I also shows the hypothetical latency of *AlexNet*, assuming we execute each layer on the fastest HMPSoC component (CPU or GPU). Latency of *AlexNet* in such a hypothetical CPU-GPU layer switched execution is 12.96% and 5.51% lower than CPU-only and GPU-only execution, respectively. In practice, a CPU-GPU layer switched execution will also inherently have additional overheads of switching between CPU and GPU and vice versa. This overhead will reduce latency gains.

Our Novel Contributions: We make the following novel contributions within the scope of this work.

- Based on their properties, we provide reasoning for a layer executing faster on the CPU than GPU and vice versa.
- We show a CPU-GPU layer switched execution can reduce the latency of CNN in practice on a real-world embedded platform, even with the overheads involved.

Open Source Contribution: The code for the CPU-GPU layer switched execution is publicly available for download at *https://github.com/Ehsan-aghapour/ARMCL-pipe-all ("n-pipe-1" branch)* under *MIT* license.

II. Related Work

In research, there are several optimizations, such as model architecture search [20], quantization [2], weight compression [3], and graph pruning [25], to execute CNNs entirely at the edge. On the other hand, there is research to run CNNs on resource constraint edge devices in their original form. Authors of [17] and [13], [23] propose an efficient hardware design for CNN inference on FPGAs and CGRAs, respectively. However, most edge devices in practice utilize CPUs and GPUS within of-the-shelf HMPSoCs for CNN inference [22].

Most current ML frameworks on devices use embedded CPUs rather than GPUs [21], mainly because previously embedded GPU performance was insufficient for edge inferencing [24]. However, embedded GPUs have significantly improved performance since then, but they are still nowhere near their non-embedded counterparts. Therefore, several efforts have been made to synergistically employ both CPUs and GPUs within HMPSoCs for high-performance edge inference with CNNs [5], [8], [10]–[12], [18], [24]. The authors of [21] were the first to create a pipeline between the CPU clusters of an asymmetric multi-core for CNN inference to improve throughput. Authors of [9] propose a CNN inference pipeline between the CPU and GPU to improve throughput. However, a pipeline design can only improve inference throughput, not latency.

MOSAIC [5] and *DeepX* [12] propose CNN model partitioning techniques that map the sliced model shards onto multiple HMPSoC components. *DeepMon* [8] partially offloads computation for convolution operations to GPU and thereby utilizes both CPU and GPU to minimize the inference latency. *μLayer* [11] intelligently maps ML inference tasks to CPU and GPU on edge devices, leveraging layer distribution and processor-specific quantization techniques. The authors of [6] propose a CNN inference latency prediction model for GPU and design multipath neural networks, enabling the runtime to choose a path that meets latency constraints. *However, to the best of our knowledge, no work exists that proposes a CPU-GPU layer switched execution to reduce the latency of CNN inference.*

III. Experimental Setup

We use *Khadas Vim 3* embedded platform in this work. An *Amlogic A311D* HMPSoC, as shown in Figure 1, powers the *Khadas Vim 3* platform. The platform has a Hexa-core asymmetric *ARM big.Little* multi-core CPU with two CPU clusters, *big* and *Little*. This work uses only the *big* CPU cluster in this work. The quad-core *big* CPU cluster contains four *A73* cores. The HMPSoC contains a dual-core *Mali G52 MP4* GPU. The operational (maximum available) frequency for the *big* CPU cluster and GPU are 2.2 GHz and 0.8 GHz, respectively. A 4 GB LPDDR4 is the main memory for the HMPSoC. The platform is running *Android* v9.0 with kernel v4.9. On top of that, we use *ARM-CL* v21.02 in this work for CNN inference. It is important to note that even though we presented the results on one specific board, the proposed methodology can be applied to conventional embedded devices wherein the embedded GPU has the processing power comparable to Multi-core CPUs.

We use multiple CNN models, namely *AlexNet*, *GoogleNet*, *MobileNet*, *ResNet50*, and *Squeezenet*, as the application kernels. These CNNs perform image classification on the *ImageNet* dataset for 1000 image classes. The input to these models is an image of size (224×224) with three channels (RGB), and the output is a tensor of size 1000 that predicts the input image class.

IV. Inference With *ARM-CL*

We implement CPU-GPU layer switched inference using the *ARM-CL* framework. We first describe the default implementation of CNN inference in *ARM-CL*. We then describe the modifications that enable CPU-GPU layer switched inference.

The *ARM-CL Framework:* A typical CNN consists of a chain of hidden layers between the input and output. Each hidden layer consumes input from the previous layer for the process, producing an output for the subsequent layer. Convolutional layers are the most common (and dominant) layers in CNNs [22]. A convolutional layer has kernels to perform matrix operations with layer-specific weights and biases on its input data. Figure 3 illustrates the CNN architecture for *AlexNet* using an abstract block diagram.

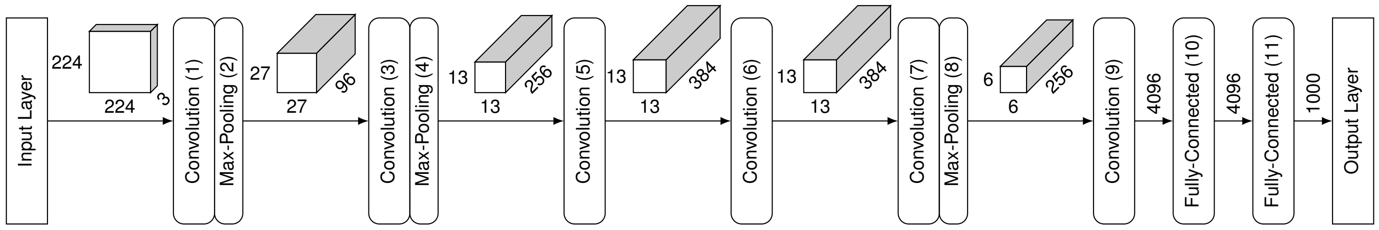


Fig. 3: The CNN architecture for *Alexnet*.

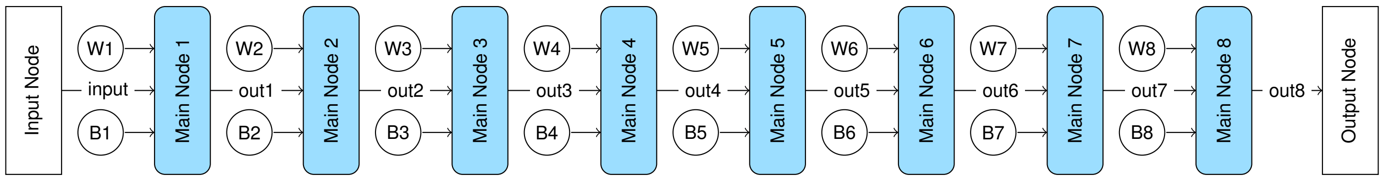


Fig. 4: The graph for *Alexnet* in *ARM-CL* corresponding to its CNN architecture.

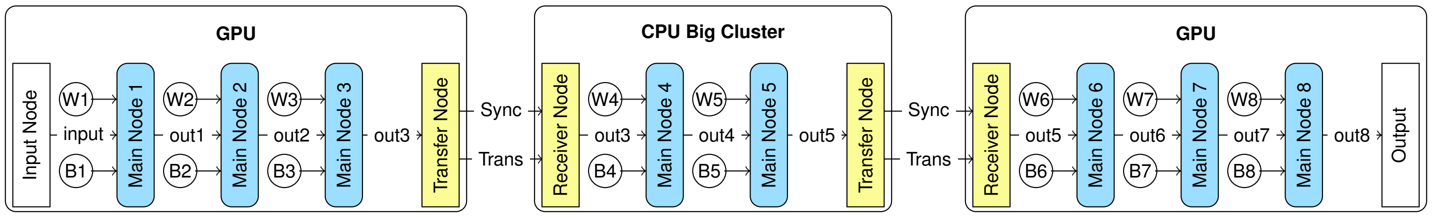


Fig. 5: The three sub-graphs for *Alexnet* obtained from partitioning it into three sub-networks, mapped on GPU, CPU Big cluster and GPU, respectively.

ARM-CL is a collection of low-level Machine Learning (ML) functions optimized for the *ARM Cortex-A* CPU and *Mali* GPU cores. The library provides ML acceleration on *ARM Cortex-A* CPUs through *Neon* (or *SVE*) and acceleration on the *ARM Mali* GPUs through *OpenCL* [1]. The *ARM-CL* represents CNN as a graph for execution on the underlying hardware. Figure 4 shows the *ARM-CL* graph corresponding to the *Alexnet* CNN architecture. In the *ARM-CL* graph, an `Input` and `Output` node represents CNN's input and output layer, respectively. The graph connects the `Input` and `Output` nodes through a series of sequentially connected `Main` nodes. There exists a `Main` node for each hidden layer in the graph. It is important to note that the number of `Main` nodes is not necessarily equal to the number of layers in the CNN. For instance, the graph in Figure 4 subsumes max-pooling layers in *AlexNet* in the `Main` nodes corresponding to the preceding convolutional layer. The graph also connects each `Main` node with its two exclusive `Weight` and `Bias` nodes. `Weight` and `Bias` nodes provide (to the `Main` node) the weights and biases input, respectively. The primary input for the `Main` node comes from the preceding `Main` (or `Input`) node in the graph. Therefore, the graph binds all nodes in a chain of consumer-producer relationships.

Environment Setup: *ARM-CL* begins by generating a graph corresponding to the user-defined CNN to initialize the execution environment. It then sets up the back-end context on the target component (CPU and GPU). The setup for the CPU includes generating worker threads either automatically based on the number of CPU cores or as per a user-defined number of requested threads. For the GPU, it extracts details such as the number of cores and the model number. It creates an *OpenCL* context with a `CLScheduler` optimized for the detected GPU device.

As the next step, the *ARM-CL* determines the features of tensors (such as their shape and data types) based on layer inputs. *ARM-CL* provides a highly optimized implementation of kernel functions to support its execution for each `Main` node type. Finding the optimal implementation and configuration for each node is based on specifications of the underlying hardware and dimensions of the operands. Consequently, it assigns memory to the tensors corresponding to the weights and biases. It loads them with values, serializes the kernels, and prepares them for execution in the correct sequence on the target processor.

Running the Graph: The *ARM-CL* sends the frame (initial input) to the `Input` node to trigger graph processing. After the frame is loaded, the kernels start processing the data. When the target processor is a CPU, the *ARM-CL* partitions the computations within the matrix operation and distributes them between the CPU worker threads. On completion of the computation, these threads fill in the results to the corresponding output tensor. The process continues until all kernels (`Main` nodes) have finished execution. When the target processor is the GPU, *ARM-CL* pushes the kernels to the *OpenCL* queue instead of the CPU worker threads. The *OpenCL* takes over the task of kernel executions on the GPU cores. The *ARM-CL* puts the output from the last `Main` node in the input tensor of the `Output` node.

Partitioning the Graph: We split the original *ARM-CL* graph into sub-graphs to enable CPU-GPU layer-switched execution. We create a sub-graph for each set of consecutive layers that execute on the same component. New `Transfer` and `Receiver` nodes perform synchronized data transfer between sub-graphs to enable component switching mid-inference. The `Receiver` node waits on the `Transfer` node of the preceding sub-graph using the *wait_queue* and releasing the computing resources. As soon as a subgraph's process finishes, its `Transfer` node interrupts the corresponding `Receiver` node and transfers the data. Figure 5 shows one possible sub-graph formulation for *AlexNet*. The formulation allows inference to begin on the GPU, which then switches to the CPU for a few layers before switching back to the GPU for the remaining layers.

V. Layer Latency Analysis

A CNN is a sequence of layers that process a given input consecutively to generate an output. Each layer of the neural network processes the input from the preceding layer in the form of a tensor. Layers have associated trainable parameters (weights and biases) that remain unchanged during the inference and load at the set-up time to process this data. The output of the layers is a tensor, computed by the multiplication operation between its input and weight tensors, followed by an addition operation with the bias matrix. Each layer loads its input data into the memory of the target processor (CPU or GPU) and produces the output tensor for the next layer.

There are four types of layers in a CNN – convolution, pooling, normalization, and fully connected. There are two main parts in a CNN wherein these different layers exist. The first part consists of convolution, pooling, and normalization layers that extract the features from the input and then feed these feature maps into the second part. The second part consists of fully connected layers that predict the output based on the feature maps. The amount of computation within a layer and the communication between layers depend on the layer parameters' type and size.

We start by providing a brief introduction to layer-specific parameters, paying particular attention to the size and dimensions of the data involved during the layer operations. We then analyze the impact of the size of the parameters on the layer's execution time on CPU and GPU. We then reason why a given layer executes faster on the CPU than GPU and vice versa based on the size of the layer parameters.

A. CNN Layers

Convolution Layer. In a convolution layer, the first input (from the preceding layer) is a tensor of the shape: $(input\_height) \times (input\_width) \times (input\_channels)$. The second input is a filter consisting of kernels with the same number of input channels. All filters independently operate on the input data. Since every kernel has an identical shape within a layer, a filter is a tensor with the shape: $(kernel\_height) \times (kernel\_width) \times (input\_channels)$. A filter performs the convolution operation and generates a middle tensor with the shape: $(output\_height) \times (output\_width)$ for each input channel. Subsequently, these middle tensors are combined to generate the final output tensor of the shape: $(output\_height) \times (output\_width) \times (number\_of\_filters = output\_channels)$.

Figure 6 illustrates the input data structure and filters for a convolution layer. This layer has two filters and generates two channels for the output tensor. The number of kernels in each filter equals the number of input channels and is three for this example layer. The shape of each output channel (output height and output width) depends on the shape of the input and kernel, stride, and padding.

Normalization Layer. These layers normalize their output using a moving average of the mean and standard deviation of the batches they have seen during training. This layer is non-trainable, and its input and output sizes are the same.

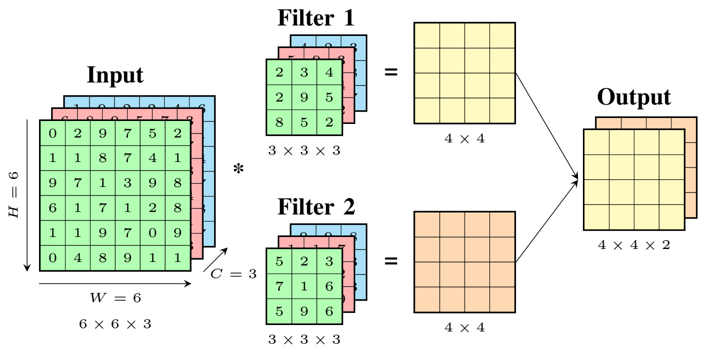


Fig. 6: Data structure for a convolution layer. This layer has an input tensor with three channels. It has two filters processing the input tensor, producing two channels of the output tensor.

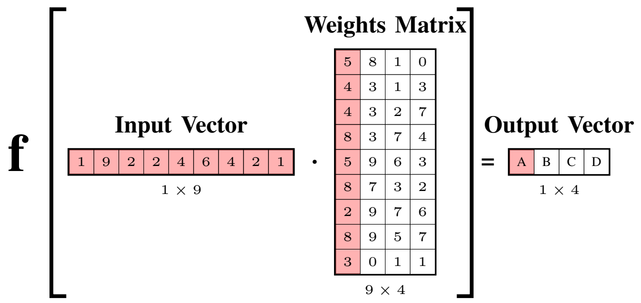


Fig. 7: Data structure for a fully-connected layer. This layer has a flattened input tensor (one dimension). The weight tensor is a 2D tensor with the shape: $(input\_size) \times (number\_of\_neurons)$. The output is a flat tensor with a size of $(number\_of\_neurons)$.

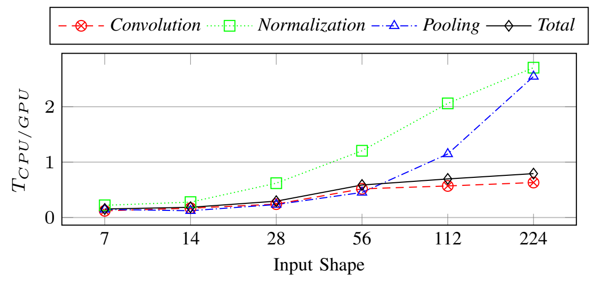
Pooling Layer. The pooling layers reduce the dimensions of the feature maps obtained from convolutions. This layer computes (and replaces) the maximum/average of neighboring neurons for neurons in a feature map, reducing the total number of inputs into subsequent layers. These layers also do not have trainable parameters. The output size is always smaller than its input size, depending on how many points in the input frame are replaced by one value.

Fully-Connected Layers. The output of the last convolution layer is flattened and sent to the first fully-connected layer. So the input tensor to the fully-connected layer is a flattened tensor with the size: $(input\_height \times input\_width \times input\_channels)$. The input tensor connects to all neurons of the fully-connected layer with a weight parameter. For each neuron, the output is the weighted sum of all inputs, and the shape of the weight tensor is $(input\_size) \times (number\_of\_neurons)$.

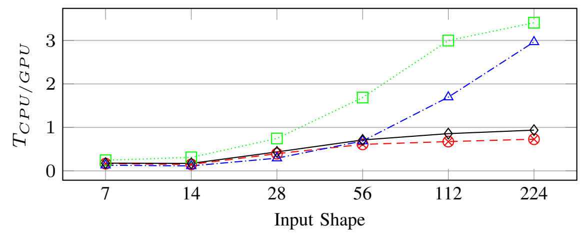
Figure 7 illustrates a fully connected layer's input, weight, and output. This structure is a significantly dense connection. The number of mathematical operations in a fully-connected layer is much more than in a convolution layer, even though the input and output tensors are much smaller than in a convolution layer.

B. Effect of Parameters Size on Latency

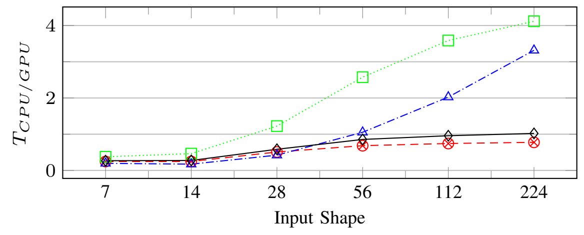
This work observes that some layers within a CNN execute faster on the CPU while other layers perform better on the GPU. We aim to find the reason behind the best HMPSoC component (CPU or GPU) for layers considering their operation types and the size of their parameters. For this purpose, we compare the CPU and GPU layer processing time with different parameter



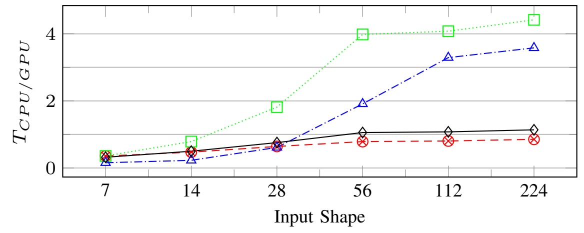
(a) Filters=32



(b) Filters=64



(c) Filters=128



(d) Filters=256

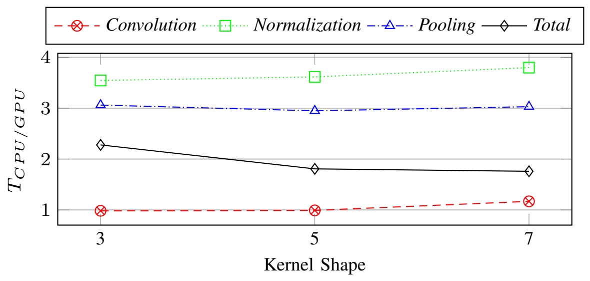
Fig. 8: Exploring the relation of $t_{CPU/GPU}$ with different input shapes and the number of filters. Input channels and kernel shapes are 64 and 3, respectively.

TABLE II: The explored values for convolution, normalization, and pooling layer size variables.

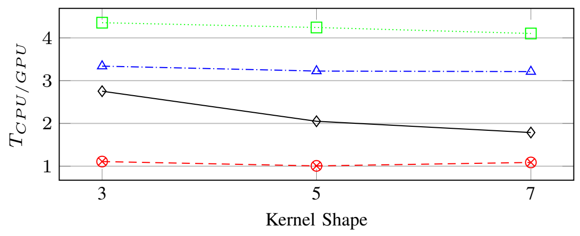
Size Variable	Explored Values
Input Shape	7, 14, 28, 56, 112, 224
Input Channels	1, 2, 32, 64, 128, 256, 512, 1024
Number of Filters	16, 32, 64, 128, 256, 512, 1024
Kernel Shape	3, 5, 7

sizes. First, we measure the layer processing time with the CPU (T_{CPU}) and the GPU (T_{GPU}). Then, we use the measurements to calculate the ratio of T_{CPU} to T_{GPU} ($T_{CPU/GPU}$). When the value of this ratio is less than one, the CPU is faster than GPU in processing the layer. When it is bigger than one, GPU is faster than the CPU.

GPU is always a more efficient processor regardless of the size of tensors for fully-connected layers. However, for other layers, the more efficient processor depends on the size of the tensors involved in the processing. Therefore, we limit our analysis of $T_{CPU/GPU}$ to different parameter sizes in convolution,



(a) Filters=64



(b) Filters=128

Fig. 9: Exploring the relation of $T_{CPU/GPU}$ with kernel shape for first convolution layer with input shape 224 and three channels. The kernel shape has no significant effect on the value of $T_{CPU/GPU}$ for convolution layer (blue line).

normalization, and pooling layers. The normalization and pooling layers follow a convolution layer and must execute on the same component (to avoid significant data switching overhead) as the convolution layer. Therefore, we are most interested in changes in the total execution time (the sum of convolution, normalization, and pooling layer execution) with changes in the parameter sizes.

There are four size variables of interest for a convolution, normalization, and pooling layer: input shape, input channels, kernel shape, and the number of filters. Table II indicates the size variable values we explored in this work. We determined these values based on the layers' parameters in real-world CNNs. On this account, we limit our exploration to realistic values of the size of the layer parameters. We computed $T_{CPU/GPU}$ for different values of various size variables to analyze their effect on execution latency.

Input Shape Analysis. In conventional CNNs, the input of the first layer (image shape) is a tensor with shape 224 or 227 and decreases as we go deeper in the network. Figure 8 shows the value of $T_{CPU/GPU}$ by changing the input shape for a different number of filters. The input channels and the kernel shape are 64 and 3, respectively. We analyze it for other values of input channels and kernel shapes and observe similar behavior. Experiments demonstrate that by increasing the input shape, the $T_{CPU/GPU}$ also increases. *Therefore, increasing input shape makes the convolution, pooling, and normalization layers execute faster on the GPU than on the CPU.*

Kernel Shape Analysis. Most layers in CNNs have a kernel shape of 3. Although the kernel's shape in the initial layers may be more extensive (5 or 7). We explore the effect of changing the kernel shape for the first layer with input shape 224 and three input channels. Figure 9 shows the value of $T_{CPU/GPU}$ for convolution, pooling, normalization, and total timing by increasing the kernel shape for the different number of filters. The value of $T_{CPU/GPU}$ for pooling and normalization layers is much higher than one (GPU is preferred), and for a convolution

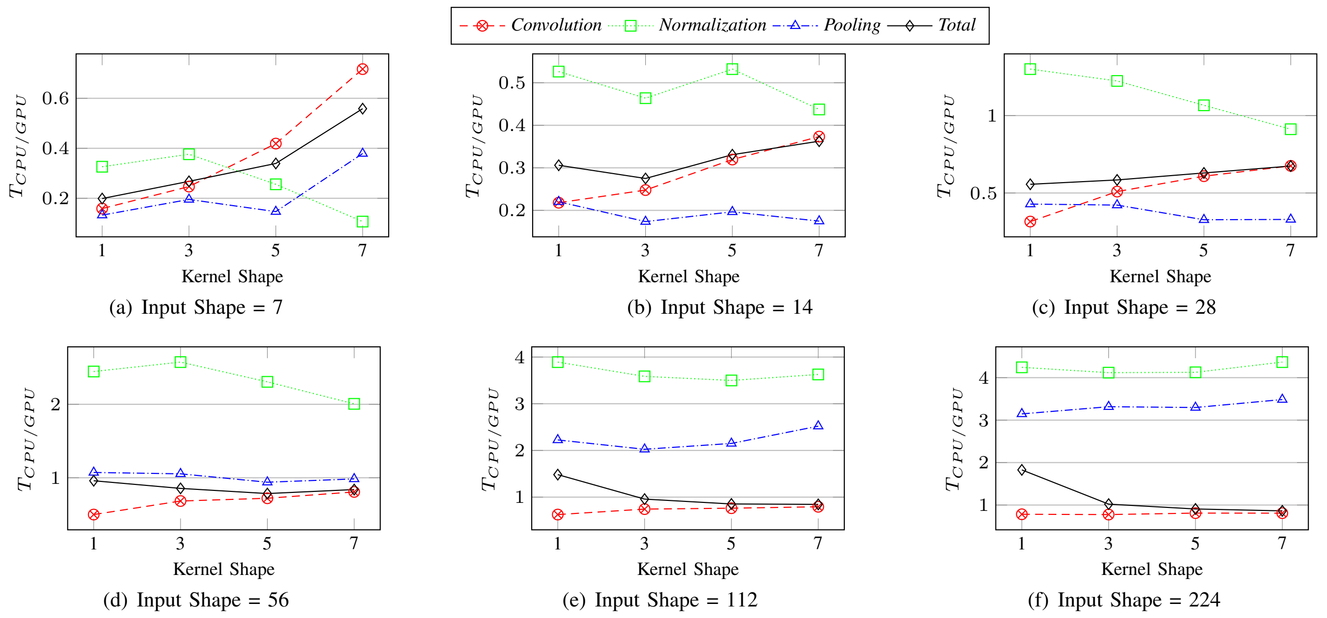


Fig. 10: Exploring the relation of $t_{CPU/GPU}$ with kernel shape. We show this exploration for different input shapes. The number of filters and input channels is 128 and 64, respectively.

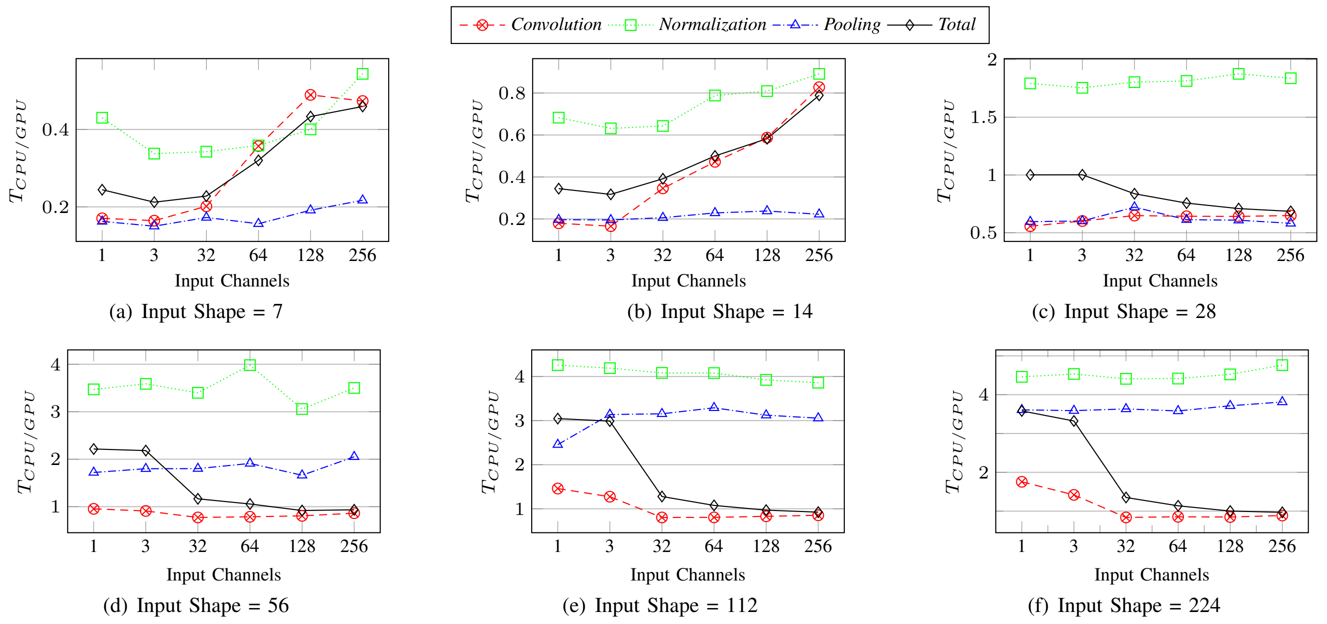


Fig. 11: Exploring the relation of $T_{CPU/GPU}$ with the number of input channels. We show this exploration for different input shapes. The number of filters and kernel shapes are 256 and 3, respectively.

layer, it is close to one. For smaller kernels, the contribution of normalization and pooling layers towards the total time is higher. We ascribe this observation to the fact that small kernels result in fewer operations in the convolution layer but do not help reduce them in the normalization and pooling layer. The input and output sizes for layers remain almost the same as the kernel shape increases or decreases. We observe that the value of $T_{CPU/GPU}$ for convolution is not changing significantly by increasing kernel shape. However, an increase in the kernel shape increases the contribution of the convolution layer in the total time, so the total time is close to the convolution time. Our analysis shows that the value of $T_{CPU/GPU}$ is always larger than one (GPU is preferred) for the first convolution layer.

There are convolution layers with a kernel shape of 1 in some models. We explored the effect of kernel size on the value of $T_{CPU/GPU}$ for these layers. Figure 10 shows the value of $T_{CPU/GPU}$ by increasing the kernel shape for different input sizes. The number of input channels and filters is 64 and 128, respectively. This figure shows that for smaller input shapes (7 and 14), the value of $T_{CPU/GPU}$ for total time increases with increasing kernel shape. On the contrary, for larger input shapes (56, 112, and 224), the value of $T_{CPU/GPU}$ does not increase significantly. We analyze it for other values for the number of

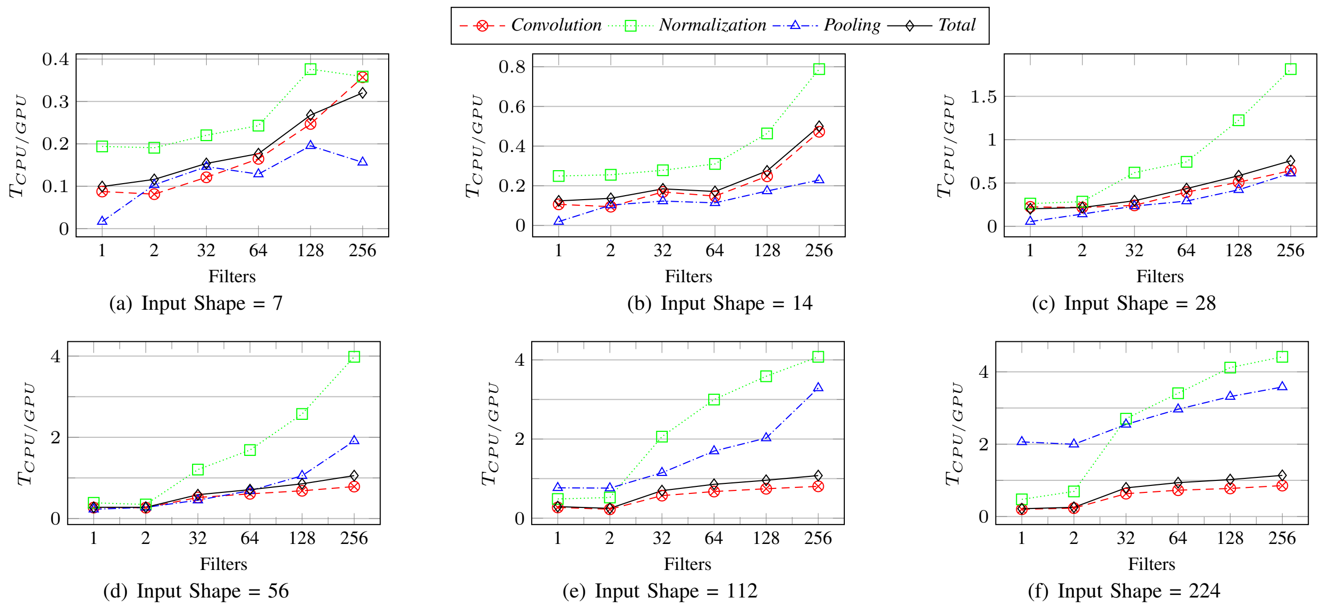


Fig. 12: Exploring the relation of $t_{CPU/GPU}$ with the number of filters. The figure shows this relation for different input shapes. The number of input channels and kernel shape is 64 and 3, respectively.

TABLE III: The preferred processing component based on size variables.

Input Shape	224	112			56			28	14	7
Input Channels	3	32		64-128	32		64	128-256	96-512	384-1024
Filters	32-96	32	64	32-64	128-256	128	256	96-512	384-1024	512-1024
Preferred Component	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	CPU	CPU

input channels and filters and see similar behavior. *Therefore, the effect of kernel shape on the value of $T_{CPU/GPU}$ depends on the other variables such as input shape and number of filters.*

Input Channels Analysis. The input tensor of CNNs starts with three channels, and the number increases by going deeper. Figure 11 shows the value of $T_{CPU/GPU}$ by changing the number of input channels for different input sizes. The number of filters and kernel size are 256 and 3, respectively. The analysis for other values of kernel size and the number of filters exhibits the same behavior as here. Our experiments show that for small input shapes (less than 28), the increase in the number of input channels increases the values of $T_{CPU/GPU}$. On the other hand, for large input shapes (larger than 28), an increase in input channels decreases $T_{CPU/GPU}$. *Therefore, the effect of the number of input channels on $T_{CPU/GPU}$ depends on the input shape.*

Filters Analysis. The filters in a layer construct the input channels for the subsequent layer, and the number of filters usually increases as we go deeper into the network. Figure 12 shows the changes in the value of $T_{CPU/GPU}$ by changing the number of filters for different input sizes. The number of input channels and the kernel shape are 64 and 3, respectively. The analysis for other values of kernel size and the number of input channels demonstrate similar behavior. Our experiments show that by increasing the number of filters, the $T_{CPU/GPU}$ also increases. *Therefore, increasing the number of filters moves the latency in support of the GPU over the CPU.*

Summary: There are many noteworthy observations in these experiments. We observe an inconsistent relationship between the number of channels and kernel shape with the value of $T_{CPU/GPU}$ for the convolution, normalization, and pooling layers. However, there is a consistent relation between the input shape and the number of filters with the $T_{CPU/GPU}$; The input shape decreases (CPU is preferred), and the number of filters increases (GPU is preferred) as we go deeper into the networks.

Table III shows the preferred component for layers based on the size variables. The table shows that for the first layer with input shape 224, layers (with different number of filters) prefer the GPU, and for deeper layers, with input shapes of 28, 14, and 7, layers prefer the CPU. However, for middle layers with input shapes 112 and 56, the preference depends on the other variables: the number of input channels and filters.

VI. Results

We evaluate the inference latency by finding the best mapping of layers to components compared to running the entire network with only the CPU or GPU. First, we measure the processing time of each layer with both the CPU and the GPU. Second, we use the best HMPSoC component for each layer and switch between CPU and GPU. However, each switch comes with an overhead that increases latency. Therefore, we only switch if we expect to improve the latency while considering the switching overhead. We use a regression model to estimate switching overhead between the components based on the involved data transfer size.

We measure the inference latency for different CNNs with the CPU-GPU layer-switched execution on the *Khadas VIM 3* board. This measurement implicitly includes the switching overhead. Figure 13 shows the latency of different CNNs with CPU-GPU

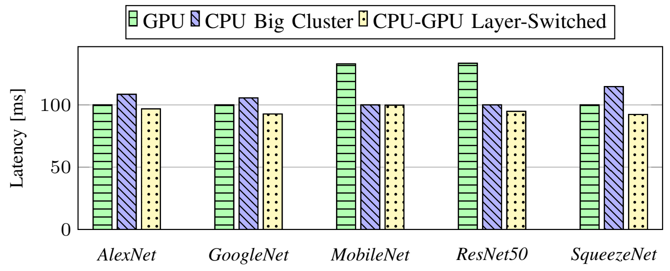


Fig. 13: Inference latency for CPU-GPU layer-switched execution compared to running the whole network with the CPU or GPU.

layer-switched execution compared to running the CNNs on only the CPU or GPU. The results show that the layer-switched execution reduces the latency, on average, by 14.40% and 9.58% compared to CPU-only and GPU-only execution, respectively. The layer-switch execution reduces the latency by 4.72% on average compared to the minimum CPU- or GPU-only execution.

VII. Conclusion

We observe that within a CNN, some layers execute faster on the CPU than on the GPU, while others execute faster on the GPU than on the CPU. We present an analysis explaining this observation based on different size variables that define a layer. Furthermore, we exploit the observation to present the first-of-its-kind concept of CPU-GPU layer-switched execution wherein a CNN layer preferably executes on the component (CPU or GPU) where it runs the fastest. We implement the layer-switched execution on *Amlogic A311D* HMPSoC within *Khadas VIM 3* board. Results show the CPU-GPU layer-switched execution can reduce the latency of CNN inference compared to a CPU-only or GPU-only execution regardless of the additional overhead introduced due to switching between the CPU and GPU within an inference. In the future, we aim to extend this work to consider the trade-off between latency and total energy consumption. The total energy consumption can be computed by analyzing CNN layers for each processing element at different frequency levels and the required energy for switching among processing elements.

References

[1] ARM. Arm compute library. https://developer.arm.com/ip-products/processors/machine-learning/compute-library, 2021. Accessed: May 16, 2021.

[2] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[3] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, et al. Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 395–408, 2017.

[4] Jonatan S Dyrstad and John Reidar Mathiassen. Grasping virtual fish: A step towards robotic deep learning from demonstration in virtual reality. In *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 1181–1187. IEEE, 2017.

[5] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, Jinsu Park, and Woongki Baek. Mosaic: Heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 165–177. IEEE, 2019.

[6] Seonyeong Heo, Sungjun Cho, Youngsok Kim, and Hanjun Kim. Real-time object detection system with multi-path neural networks. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 174–187. IEEE, 2020.

[7] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[8] Loc N Huynh, Youngki Lee, and Rajesh Krishna Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 82–95, 2017.

[9] Duseok Kang, Jinwoo Oh, Jongwoo Choi, Youngmin Yi, and Soonhoi Ha. Scheduling of deep learning applications onto heterogeneous processors in an embedded device. *IEEE Access*, 8:43980–43991, 2020.

[10] Woosung Kang, Kilho Lee, Jinkyu Lee, Insik Shin, and Hoon Sung Chwa. Lalarand: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 329–341. IEEE, 2021.

[11] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[12] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12. IEEE, 2016.

[13] Zhaoying Li, Dhananjaya Wijerathne, Xianzhang Chen, Anuj Pathania, and Tulika Mitra. Chordmap: Automated mapping of streaming applications onto cgra. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[14] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–766, 2018.

[15] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[16] Gopinath Mahale, Pramod Udupa, Kiran Kolar Chandrasekharan, and Sehwan Lee. Windconv: A fused datapath cnn accelerator for power-efficient edge devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4278–4289, 2020.

[17] Yuan Meng, Sanmukh Kuppannagari, Rajgopal Kannan, and Viktor Prasanna. Dynamap: Dynamic algorithm mapping framework for low latency cnn inference. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 183–193, 2021.

[18] Svetlana Minakova, Erqian Tang, and Todor Stefanov. Combining task-and data-level parallelism for high-throughput cnn inference on embedded cpus-gpus mpsocs. In *International Conference on Embedded Computer Systems*, pages 18–35. Springer, 2020.

[19] Tobias Pohlen, Alexander Hermans, Markus Mathias, and Bastian Leibe. Full-resolution residual networks for semantic segmentation in street scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4151–4160, 2017.

[20] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[21] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. High-throughput cnn inference on embedded arm big. little multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2254–2267, 2019.

[22] Siqi Wang, Anuj Pathania, and Tulika Mitra. Neural network inference on mobile socs. *IEEE Design & Test*, 37(5):50–57, 2020.

[23] Dhananjaya Wijerathne, Zhaoying Li, Anuj Pathania, Tulika Mitra, and Lothar Thiele. Himap: Fast and scalable high-quality mapping on cgra via hierarchical abstraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[24] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, qing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[25] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. *ACM SIGARCH Computer Architecture News*, 45(2):548–560, 2017.