High-Throughput CNN Inference on Embedded ARM Big.LITTLE Multicore Processors

Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra

***Abstract*—Internet of Things edge intelligence requires convolutional neural network (CNN) inference to take place in the edge devices itself. *ARM big.LITTLE* architecture is at the heart of prevalent commercial edge devices. It comprises of single-ISA heterogeneous cores grouped into multiple homogeneous clusters that enable power and performance tradeoffs. All cores are expected to be simultaneously employed in inference to attain maximal throughput. However, high communication overhead involved in parallelization of computations from convolution kernels across clusters is detrimental to throughput. We present an alternative framework called *Pipe-it* that employs pipelined design to split convolutional layers across clusters while limiting parallelization of their respective kernels to the assigned cluster. We develop a performance-prediction model that utilizes only the convolutional layer descriptors to predict the execution time of each layer individually on all permitted core configurations (type and count). *Pipe-it* then exploits the predictions to create a balanced pipeline using an efficient design space exploration algorithm. *Pipe-it* on average results in a 39% higher throughput than the highest antecedent throughput.**

***Index Terms*—Asymmetric multicore, convolutional neural network (CNN) performance-prediction, edge inference, heterogeneous multicore.**

I. Introduction

CONVOLUTIONAL neural network (CNN) inference on edge devices has become quintessential for enriched user experience. Continuous vision tasks that use inference to extract high-level semantic information from real-time video streams are paramount in numerous edge application domains, such as advanced driver-assistance systems (ADASs), virtual reality (VR), and augmented reality (AR) [15]. Inference-driven applications project unprecedented computational requirements onto underlying edge devices [34]. Fortunately, there has been tremendous progress to port CNNs to edge devices. Many network models, such as *MobileNet* [9] have been invented specifically for edge to perform high-accuracy classifications with considerably smaller network size. Numerous efficient libraries, such as ARM Compute Library (ARM-CL) [1] and *Tencent NCNN* [3] have been constructed precisely to facilitate efficient CNN implementation for the edge. ARM-CL is highly optimized for edge-specific *ARM* core architectures with inbuilt support for multithreading and acceleration through *ARM NEON* vectorization technology.

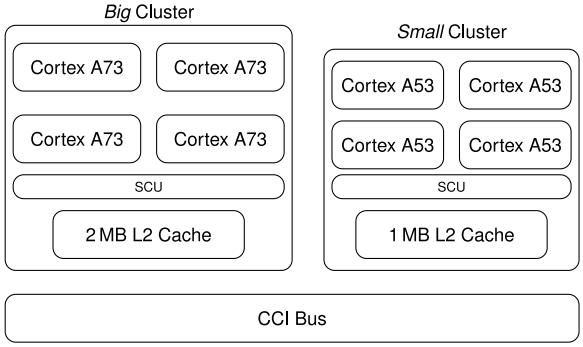


Fig. 1. Abstract block diagram of an eight-core *ARM big.LITTLE* heterogeneous multicore within *Hi3670* SoC [4].

Single-ISA heterogeneous multicores comprise of processing cores that have different power-performance-area characteristics but share the same instruction set architecture (ISA) [16]. Facebook [31] in 2019 reports that about half of the mobile systems on chip (SoCs) in the market adopts such architecture with two CPU clusters: 1) a high-performance cluster and 2) an energy-efficient cluster. This heterogeneous configuration provides higher parallel processing potential than homogeneous multicores within given power and area budget provided all cores can be simultaneously employed productively [23], [27]. Fig. 1 shows an abstract block diagram for the eight-core state-of-the-art *ARM big.LITTLE* heterogeneous multicore in *Hi3670* SoC designed for edge devices. *Hi3670* groups together four high-performance *Big Cortex A73* cores and four low-performance *Small Cortex A53* cores into two clusters alongside L2 caches of size 2 and 1 MB, respectively. Two clusters are kept fully cache-coherent via bus-based cache coherent interconnect (CCI) using snooping broadcast protocol. Cores

Manuscript received March 14, 2019; revised July 25, 2019; accepted September 17, 2019. Date of publication September 30, 2019; date of current version September 18, 2020. This work was supported in part by the Singapore Ministry of Education Academic Research Fund Tier 2 under Grant MOE2015-T2-2-088. This article was recommended by Associate Editor C. Yang. *(Corresponding author: Tulika Mitra.)*

S. Wang, Y. Zeng, A. Pathania, and T. Mitra are with the Department of Computer Science, School of Computing, National University of Singapore, Singapore 117417 (e-mail: wangsq@comp.nus.edu.sg; yifan122@comp.nus.edu.sg; pathania@comp.nus.edu.sg; tulika@comp.nus.edu.sg).

G. Ananthanarayanan is with the Department of Computer Science and Engineering, Indian Institute of Technology Dharwad, Dharwad 580011, India (e-mail: gayathri@iitdh.ac.in).

N. Goel is with the Department of Computer Science and Engineering, Indian Institute of Technology Ropar, Rupnagar 140001, India (e-mail: neeraj@iitrpr.ac.in).

Digital Object Identifier 10.1109/TCAD.2019.2944584

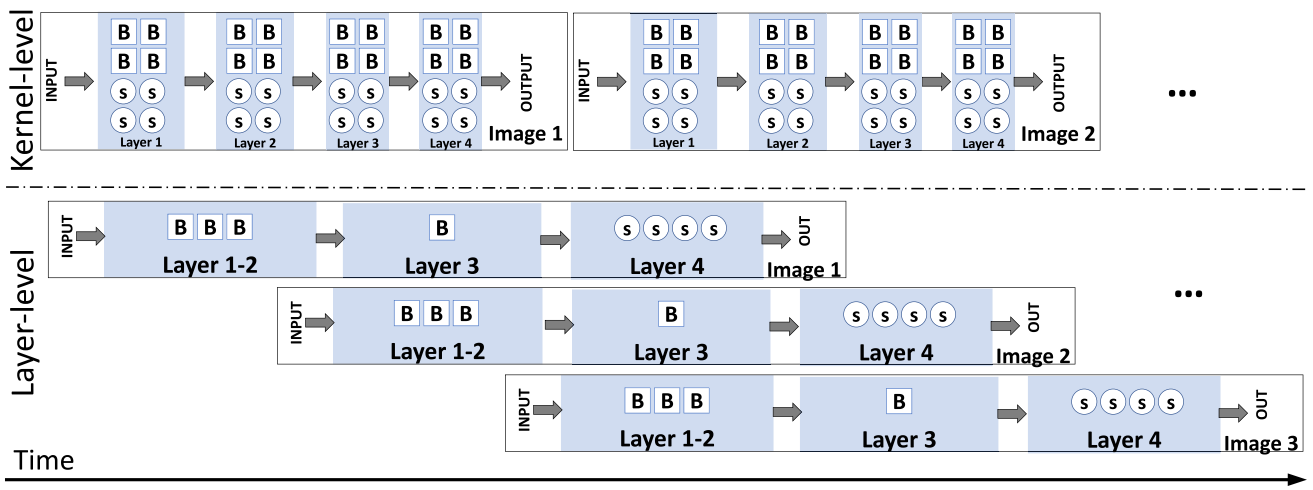


Fig. 2. Visualization of the default *kernel-level* and the proposed *layer-level* splitting with a three-stage pipeline (B3-B1-s4) on heterogeneous multicore with four *Big* (B) cores and four *Small* (s) cores for a representational four-layer CNN.

within a cluster are kept coherent using bus-based snoop control unit (SCU). This raw computational power provided by heterogeneous multicore makes CNN inference on edge device feasible.

Dedicated accelerators, such as GPUs and dedicated IP cores have been proven to be more efficient than CPU for inference. However, their applicability is constrained by the extreme diversity of accelerators and lack of easy programming support. CPU remains the platform of choice for running ML workloads being the most common denominator with high availability in mobile and embedded platforms [21], [30], [31], [36]. In addition, low-cost edge devices may not contain dedicated accelerators, and the performance gap between CPU and GPU is small, making CPU the favorable choice for ML workloads. On the other hand, CNNs are more commonly used as a building block to construct more complex systems. For applications ranging from smart classroom [24] with person and text recognition, to autonomous drones [26] with path planning, object classification and obstacle avoidance, multiple independent inference subtasks are performed concurrently. Such applications require all the available resources to run these inference engines in parallel. Therefore, improving inference throughput on *ARM big.LITTLE* like architectures by itself is a critical problem.

Motivational Example: The layers in a CNN are in a predordained order by design, which is usually in sequential. Their associated convolutional kernels are therefore required to be processed sequentially. Nevertheless, different images from an image stream can potentially be processed in parallel. Unfortunately, existing state-of-the-art deep learning libraries such as ARM-CL is designed to process the image stream sequentially one image at a time. The computation of one kernel at a time is then distributed across all cores with the default parallelization strategy we christen *kernel-level* execution. Fig. 2(top) visualizes the *kernel-level* strategy for a representative four-layer CNN on a eight-core heterogeneous multicore. Section II provides further details on *kernel-level* strategy. The *kernel-level* strategy works

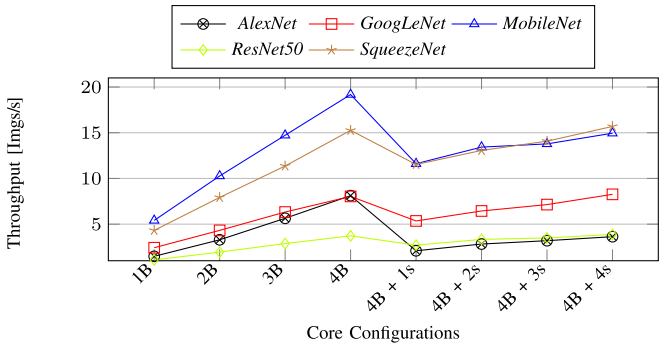


Fig. 3. Throughput of different CNNs with a different number of heterogeneous cores (B: *Big* core, s: *Small* core) using the default *kernel-level* strategy.

for intra-cluster processing but fails to scale to intercluster processing with multiple clusters.

Heterogeneous multiprocessing (HMP) allows execution of kernels using both *Big* and *Small* cores simultaneously. Fig. 3 shows the change in throughput (measured in images per second) of several CNNs with the increase in the number of heterogeneous cores used with *kernel-level* strategy. Throughput increases as we add more *Big* cores but drops sharply on the addition of *Small* cores from another cluster for HMP. Intercluster communication overhead involved in the use of HMP explains the drop. No HMP configuration surpasses the performance of configuration with four *Big* cores. Therefore, Fig. 3 empirically shows that we cannot improve throughput on heterogeneous multicores with default *kernel-level* strategy alone. This limitation originates from the design of *kernel-level* strategy and not from the quality of its implementation.

There are multiple convolutional layers of different dimensions within a CNN that project different resource requirements. Therefore, it is possible to create a processing pipeline with stages composed of only homogeneous cores that still splits CNN processing over different heterogeneous clusters.

Let notation {*core_type*}{*core_count*} denote the core configuration of a pipeline stage. Fig. 2 shows a three-stage pipeline created to process incoming images in a stream using the *layer-level* strategy. Three *Big* cores (B3) construct first pipeline stage processing layers 1 and 2. Remaining one *Big* core (B1) constructs second stage processing layer 3. Four *Small* cores (s4) construct third pipeline stage processing layer 4. This pipeline constructively uses all eight heterogeneous cores in execution by processing multiple images in parallel. Generally, initial layers operating on bigger inputs requires more computational power and memory compared to deeper layers. Therefore, it is intuitive to map initial convolutional layers to more powerful *Big* cluster and deeper layers to less powerful *Small* cluster. However, the design space of mapping layers to core clusters increases exponentially with the increase in the number of layers.

Our Novel Contributions: We propose a framework called *Pipe-it* that partitions CNN layers across heterogeneous cores to improve throughput. *Pipe-it* creates a processing pipeline by splitting layers among heterogeneous core clusters, wherein a given set of homogeneous core(s) always process kernels from a fixed set of layers. Different pipeline stages (and cores within) are responsible for concurrently processing different layers corresponding to consecutive images in a stream. The pipelined execution improves throughput by employing all on-chip memory and processing resources of heterogeneous multicore more effectively than the default approach of splitting individual kernels across all heterogeneous cores.

Pipe-it includes an analytical performance model that predicts the performance of convolutional layer on different core configurations (type and count) from its network structure description. Its design space exploration (DSE) algorithm then uses the predicted performance to locate the best fitting pipeline configuration and respective layer allocation. On average, we get 39% improvement in throughput from entire heterogeneous multicore compared to using only its high-performance homogeneous *Big* cluster.

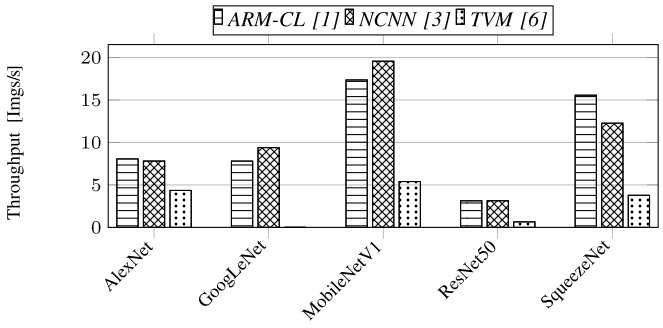
II. Background

ARM-CL [1] is a state-of-the-art framework for implementing CNNs on *ARM* architectures. Fig. 4 shows the throughput of CNN inference implemented with ARM-CL (*version 18.05*), *Tencent NCNN* [3], and *TVM* [6] frameworks running on *Big* cluster using multithreading. Both ARM-CL and *Tencent NCNN* support acceleration through *ARM NEON* vectorization and provides *NEON* assembly implementation for most computationally intensive convolution kernels of CNN. These two frameworks present similar performance and outperform *TVM* implementation without *NEON* acceleration. However, *Tencent NCNN* is not as well maintained or supported as ARM-CL. Therefore, we use ARM-CL as the foundational framework in this article.

ARM-CL is a collection of functions commonly used in machine learning. The functions are infused with hardware-specific optimizations for superior performance on *ARM* architectures. *Graph* API accompanying ARM-CL facilitates the creation of complex networks. The network is written with dedicated API as a graph by the user at the frontend. The execution is automatically handled at the backend. Graph implements the layers as nodes that are connected to other nodes in the CNN sequence as defined by the user. Table I summarizes the architecture of several popular CNNs and their respective implementations in ARM-CL. We count weighted layers (convolutional or fully connected) as major layers because they are, in general, most computationally expensive part of CNNs.

Inside each node, the workload is represented as a series of compute kernels. Runtime scheduler sequentially dispatches the kernels in q node and engages respective processing unit during execution. ARM-CL implements a convolution node with *NEON* acceleration using image to column (*im2col*) and general matrix multiplication (GEMM) kernels. In addition, the parallel nature of the kernels allows their computations to be distributed across multiple cores. This node-level parallelization is implemented in the form of a thread pool that spawns several new threads and distributes the computation of a kernel among them before the scheduler dispatches them for execution.

We extended the default ARM-CL CNN implementations to execute multiple graphs in parallel. The implementation allows



*TVM results are generated with *NNVM-TVM* framework with a pre-trained model from *mxnet.gluon.mode_zoo.vision* model set [2], wherein *GoogLeNet* is not included.

Fig. 4. Throughput of different CNN models on *Big* cluster when implemented in different deep learning frameworks.

TABLE I
Structure of Different CNN Models and the Corresponding Major Layer (Node) Counts in Their Default ARM-CL Implementations

CNN	Major Layers/Modules	*ARM-CL* Major/(Total Node Count)
AlexNet [15]	5 Conv + 3 FC	11* / (21)
GoogLeNet [29]	3 Conv + 9 Inception Modules (6 Conv Each) + 1 FC	58 / (132)
MobileNet [9]	14 Conv + 13 Conv DW + 1 FC	28 / (58)
ResNet50 [8]	1 Conv + 4 Residual Blocks (52 Conv in Total) + 1 FC	54 / (146)
SqueezeNet [12]	2 Conv + 8 Fire Module (3 Conv Each)	26 / (58)

Conv: Convolutional Layers; FC: Fully-connected Layers; Conv DW: Depthwise Convolutional Layers. *Three convolutional layers are implemented as two nodes each for *AlexNet*.

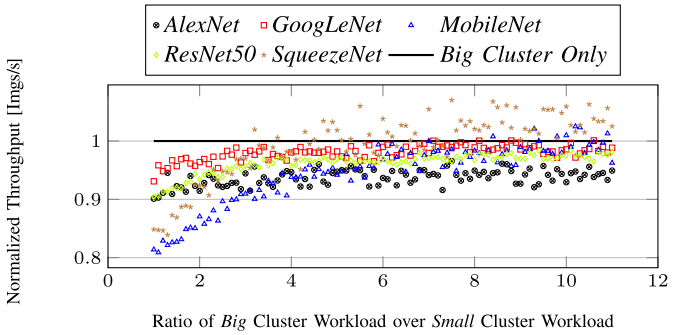


Fig. 5. Throughput of CNN models with disproportionate *kernel-level* workload split between *Big* and *Small* cluster normalized against throughput achieved using *Big* cluster only.

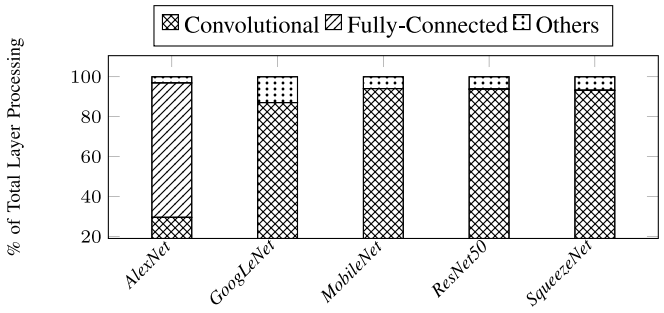


Fig. 6. Breakdown of CNN processing time between different layer types.

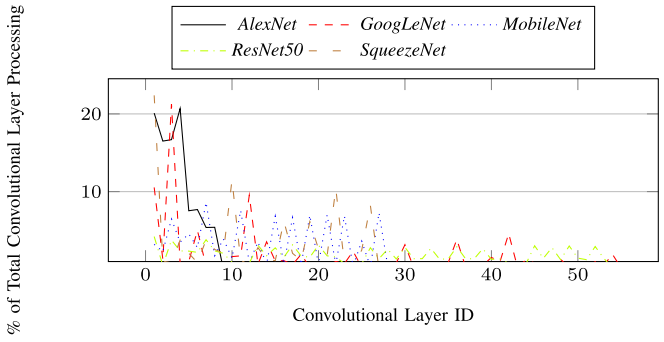


Fig. 7. Distribution of total convolution processing time among convolutional layers for different networks.

the same network to be applied to multiple images concurrently. All graphs share the same copy of read-only parameters (weights and biases) and each graph contains its unique copy of the image CNN needs to classify as we assume images in a stream to be independent. We modify the scheduler to run under a one-thread-per-core model with minimal migrations using thread pinning for faster and predictable execution.

III. Co-Execution at Different Levels

A. Kernel-Level Splitting

We can explore parallelism inherent in kernels by exploiting ARM-CL thread pool implementation to engage all cores. While the parallelization of a kernel across homogeneous cores within a cluster gives performance benefits, further parallelization across heterogeneous clusters does not improve throughput as shown in Fig. 3. Damschen *et al.* [7] made a similar observation for *kernel-level* splitting in the context of CPU-GPU co-execution.

Using multiple cores within the same cluster for processing increases parallel L2 accesses per unit time. Cluster's SCU successfully handles the increased accesses without being overwhelmed and thereby improves performance. However, when additional cores from another cluster are engaged, the working set gets split between the L2 caches of two clusters. Some conflict misses that occur on one cluster now get served by L2 cache of another cluster using CCI increasing average on-chip L2 access latency. Additional L2 cache decreases the number of capacity misses going to main memory. However, the decrease cannot compensate for the increased latency of conflict misses.

Fig. 3 shows the throughput obtained by splitting the computational workload from kernel equally among all threads. However, distributing workload disproportionately does not improve throughput significantly either. Fig. 5 shows through exhaustive search that no ratio of workload split between *Big* and *Small* clusters results in statistically significant higher throughput for most CNNs than when kernels run exclusively only on *Big* cluster. Exhaustive search indicates we must give little or no share of computational work to *Small* cluster for optimal execution.

B. Layer-Level Splitting

Image classification CNNs are made up of multiple layers, which process images sequentially. Fig. 6 shows the share of processing time spent on convolutional layers in different CNNs normalized to total forward pass processing time. Processing of convolutional layers dominates overall time spent for all networks except in relatively older *AlexNet*, wherein fully connected layers dominate.

The convolutional layer at the start of network operates upon the original data of the biggest size (and dimensionality) and produces output data of smaller size due to the application of filters. This shrunken output gets passed on to the subsequent convolutional layer as input, which reduces its convolution processing time. Fig. 7 shows that time taken to process convolutional layers generally decreases as we move deeper into a network.

Observations from Fig. 7 can help us in creating a load-balanced processing split on a heterogeneous multicore. Its high-performance cores can process more processing-intensive initial layers, while low-performance cores can process less processing-intensive deeper layers. Kernels from layers can still get split among all homogeneous cores within a cluster using *kernel-level* splitting. Kernels from nonconvolutional layers are considered part of previous convolutional layers and get processed at the same cluster. We do not explore layer-level splitting of CNN at nonconvolutional layers.

Layer-level splitting between clusters produces a lower number of intercluster L2 conflict misses than *kernel-level* splitting as most layers that feed data into each other are processed on the same cluster reducing the load on CCI. Furthermore, it also allows for multiple images from a stream to be processed

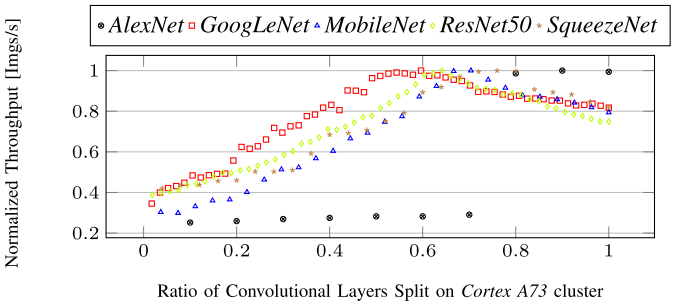


Fig. 8. Throughput of a two-stage pipeline (B4-s4) with workload split at different convolutional layers normalized against the maximum throughput obtained.

in parallel. The *Big* cluster can start processing layers from image $Z+1$, while *Small* cluster is still processing layers from image Z. *Layer-level* splitting, unlike *kernel-level*, also requires less movement of weight and biases between clusters. It processes weights and biases shared between the kernels of different images on the same cluster. This optimization further reduces the amount of conflict misses between clusters and thereby improves L2 cache usage efficiency.

IV. Design Space

A. Split Points at Convolutional Layers

Structure of different convolutional layers can differ significantly from each other within a network. Their performance on *Big* and *Small* clusters with a different number of allocated cores can also be quite different. These differences mandate nontrivial decisions on splitting convolutional layers across pipelines stages of *Pipe-it*.

Consider a basic two-stage layer-level split pipeline (B4-s4) processing a network containing W major layers. The parameters are summarized in Table II. First X layers are processed on *Big* cluster with *kernel-level* split among all four *Big* cores and rest $(W-X)$ layers are processed on *Small* cluster. The challenge is to find an optimal split point X with maximum throughput. There are $\binom{W-1}{1} = (W-1)$ possible split points in this pipeline. Fig. 8 shows throughput for different CNNs with split ratio (X/W) ranging from zero to one. We also include fully connected layers for *AlexNet* as valid points to split. Optimal split ranges from 0.60 for *GoogLeNet* to 0.90 for *AlexNet*.

Design space for a three-stage pipeline is much larger as we need to locate two split points X_1 and X_2. Consider a pipeline configuration (B4-s2-s2). Four *Big* cores, two *Small* cores, and remaining two *Small* cores are used to construct pipeline stages 1–3, respectively. Fig. 9 shows the execution of *ResNet50* with different configurations. The y-axis shows split point X_1, which splits stage 1 (B4) and [stage 2 + 3] (s2-s2). X_1 also splits *Big* and *Small* clusters for this pipeline configuration. The x-axis shows split point X_2, which splits stages 2 (s2) and 3 (s2). The z-axis shows the throughput for a workload split. Throughput peaks at 5.6 Imgs/s with split points X_1 and X_2 at layers 33 and 45, respectively. The optimal three-stage pipeline for *ResNet50* has 7% higher throughput than the corresponding optimal two-stage pipeline.

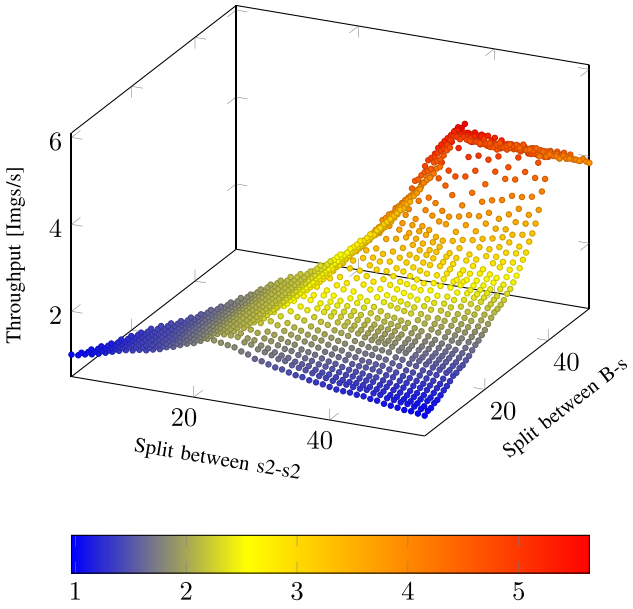


Fig. 9. Throughput of *ResNet50* with a three-stage pipeline (B4-s2-s2) with workload split at different layers.

B. Stages of Pipelines

We can create pipelines with many more stages (up to H on heterogeneous multicore with H cores) in pursuit of higher throughput for CNN inference. We eliminate pipeline designs with heterogeneous core types within pipeline stage as *kernel-level* split between clusters is not helpful (Fig. 3). We only consider pipeline configurations with *Big* cores for initial convolutional layers and *Small* cores for subsequent convolutional layers as CNNs usually have more compute-intensive convolutional kernels at the beginning (Fig. 7).

Equation (1) gives the number of different pipelines possible C_p with p pipeline stages on heterogeneous multicore with H_B *Big* cores and H_s *Small* cores. We use p_B and p_s to denote the number of stages constructed with the *Big* and *Small* clusters, respectively. $\binom{H_B-1}{P_B-1} \times \binom{H_s-1}{P_s-1}$ gives the total number of different pipeline that we can construct. However, the values of p_B and p_s must satisfy the following requirements to construct a meaningful p-stage pipeline:

$$p_B \in [1, H_B],\ \ p_s \in [1, H_s],\ \ p_B + p_s = p.$$

Thus, the minimum value of $\max(1, p-H_s)$ and the maximum value of $\min(H_B, p-1)$ gives a range of p_B. We then go through p_B and calculate the total number of different pipelines possible with p stages using the following equation:

$$C_p = \sum_{P_B=\max(1,p-H_s)}^{\min(H_B,p-1)} \binom{H_B-1}{P_B-1} \times \binom{H_s-1}{(p-P_B)-1}. \quad (1)$$

Equation (2) gives the total number of design points for CNN with W convolutional layers (D_W) in *layer-level* splitting on H-core heterogeneous multicore

$$D_W = \sum_{p=2}^{H} \binom{W-1}{p-1} \times C_p. \quad (2)$$

TABLE II
DESCRIPTION OF PARAMETERS IN CHRONOLOGICAL ORDER

Parameters	Descriptions
W	Workload, number of major layers (convolutional layers, with fully-connected layers for *AlexNet*) in a CNN.
X, X_1, X_2	Split-point of workload, number of layers to be allocated to pipeline stages.
H, H_B, H_s	Number of cores in a heterogeneous multi-core architecture. B: *Big* cores, s: *Small* cores.
p, p_B, p_s	Number of pipeline stages; number of stages on *Big* and *Small* clusters.
C_p	Number of different pipeline configurations for a pipeline with p stages.
D_W	Number of design points for a CNN with W major layers on a H-core heterogeneous multi-core architecture.
I_w, I_h, I_d	Input image tensor dimensions in width, height and depth.
F_w, F_h, F_d, Ofm	Filter dimensions in width, height, depth and number of output feature maps.
Pad, S	Padding, stride information for convolution.
N, K, M	Dimensions of matrices in convolution converted GEMM.
α, β	Regression coefficients.
ts	Tile size for GEMM optimization.
n_{iter}	Number of iterations generated for image tensor with tile size ts.
$iter_t$	Number of iterations allocated to a thread t in multi-threaded execution.
T_{iter}	Execution time of a single iteration.
T_{multi}	Execution time of multi-threaded execution.
$P = \{P_1, P_2, ..., P_p\}$	Representation of a pipeline configuration with p stages
P_i = (type, count)	Representation of the configuration of the i-th stage in a pipeline P. E.g. $(B, 3)$, also written as $B3$ for convenience.
$L = \{L_1, L_2, ..., L_p\}$	Corresponding layer allocation for pipeline P with p stages.
$L_i = \{l_j, ..., l_k\}$	A set of layers in original order allocated to stage P_i, also written as l_{j-k} for convenience.
T, T^{P_i}	Time matrix for execution times of a single layer on different core configurations; Time array of execution times of a set of layers with core configuration P_i.
$T_{l_j}^{P_i}$, $T_{L_i}^{P_i}$	Execution time of layer l_j with pipeline configuration P_i; execution time of a pipeline stage P_i with its corresponding layer allocation L_i.
L_{wl}	A set of layers as defined in the context (workload).

There are in total 64 possible pipelines (with $p = 2$–8) as calculated with (1) for our prototype board with eight-core heterogeneous multicore. Furthermore, there are in total 5 379 616 distinct possible design points for *MobileNet* with its 28 convolutional layers as calculated using (2). Design space gets even larger for bigger CNNs like *GoogLeNet* and *ResNet50* with more layers. Therefore, it is not possible to explore entire *layer-level* splitting design space using exhaustive search in a reasonable amount of time.

C. Pipe-It Framework

We present a two-part *Pipe-it* framework to quickly go through huge design space and locate the best configuration to execute given CNN workload. *Pipe-it* first predicts the execution time of all layers on all possible core configuration from static network-layer configuration descriptors (Section V). *Pipe-it* then goes through design space heuristically using predicted timing information to obtain near-optimal pipeline configuration and corresponding workload allocation (Section VI).

V. LAYER-WISE PERFORMANCE ESTIMATION

The most time-consuming part of CNNs is the execution of convolutional layers. Convolutional layers convolve input tensors with filters to generate respective output tensors, feeding into following layers as inputs. With the extensive calculation requirements, hardware-dependent implementation and optimization techniques are applied to accelerate the execution of convolutions.

GEMM is commonly used to implement convolution executions. ARM-CL first converts input image tensor and filter into a matrix (*Im2col kernel*). It then performs GEMM execution and finally transforms the execution results back into output image tensor format (*Col2Im kernel*). Lu *et al.* [22] showed execution time of convolution correlates linearly to the dimension of matrices. We build on the approach that correlates statically available descriptors of each convolutional layers with layer execution times. We evaluate and model individual convolutional layers with special consideration on the effects of multithreading, whereas Lu *et al.* [22] only considered the overall execution time of the network.

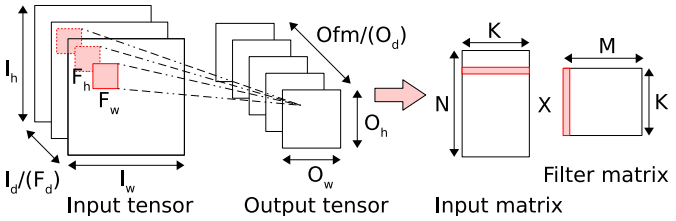


Fig. 10. Visualization of a convolutional layer with input image tensor of size $\{I_w, I_h, I_d\}$ and filter of size $\{F_w, F_h, F_d, Ofm\}$ generating output tensor of size $\{O_w, O_h, O_d\}$. The execution is realized as GEMM of input matrix $[N \times K]$ and filter matrix $[K \times M]$ generates result matrix of size $[N \times M]$.

A. Convolution as GEMM

Fig. 10 visualizes convolution using GEMM. Consider a convolutional layer with input image tensor of size (height, width, and depth) $\{I_w, I_h, I_d\}$ and filter of size (height, width, depth, and number of output feature maps) $\{F_w, F_h, F_d, Ofm\}$, with padding *Pad*, and stride *S*. Convolutional layer generates output tensor $\{O_w, O_h, O_d\}$ of size given by (3). Input tensor and filter are required to have matching depth ($I_d = F_d$) and

are usually square ($I_w = I_d, O_w = O_h$)

$$O_w = \lfloor (I_w - F_w + 2 * \text{Pad})/S \rfloor + 1$$
$$O_h = \lfloor (I_h - F_h + 2 * \text{Pad})/S \rfloor + 1$$
$$O_d = Ofm. \tag{3}$$

ARM-CL implements convolution as GEMM of input and filter matrices. Fig. 10 shows how the input tensor are divided into small patches of size of one filter ($\{F_w, F_h, F_d\}$), denoted as the red shaded region. The patches are rearranged as rows in the image matrix. Similarly, the filters are rearranged into columns in the filter matrix. Thus the convolution is transformed into a GEMM of an image matrix ($[N \times K]$) and a filter matrix ($[K \times M]$), which generates a result matrix of size $[N \times M]$ and later resize it into an output tensor. Equation (4) gives dimensions of matrices. The total number of arithmetic operations is ($N \times K \times M$)

$$N = O_w \times O_h$$
$$K = F_w \times F_h \times F_d$$
$$M = Ofm. \tag{4}$$

Compute time of GEMM is a complex function of memory accesses, arithmetic computations, and inherent exploitable parallelism in the given convolutional kernel.

B. Single Core Estimation

We create a set of micro-benchmarks with ARM-CL to capture the execution behavior of layers commonly used in networks. The micro-benchmarks contain representative layers and a convolutional layer with desired configurations (input sizes and filter sizes). We randomly generate input images and filter parameters for measurement purposes. The GEMM execution time is measured for different configuration points using the following values of the parameters:

$$I_w = I_h = \{7, 14, 28, 56, 112\}$$
$$F_w = F_h = \{1, 3, 5, 7, 11\}$$
$$I_d = F_d = \{32, 64, 92, 128, 192, 256\}$$
$$Ofm = \{32, 64, 92, 128, 192, 256\}.$$

We observe a linear correlation between the dimensions of matrices (N, K, M) and the execution time of GEMM. Lu *et al.* [22] made similar observations. Equation (5) models the execution time of convolutional layer T by using linear regression on (N, K, M) for a single-core configuration, where β_1, ($\beta_2, \ldots, \beta_8$) are constants determined with the help of linear regression. We can physically interpret interaction terms in (5) as the size of matrices involved in GEMM (NK, KM, NM) and total arithmetic operations (NMK)

$$T = \beta_1 N + \beta_2 K + \beta_3 M + \beta_4 NK + \beta_5 KM + \beta_6 NM + \beta_7 NMK + \beta_8. \tag{5}$$

C. Multicore Estimation

ARM-CL implements GEMM optimization by multithreading and tiling with tile size (ts) determined according to the cache sizes to achieve optimal memory behavior. It uses H threads for execution on an H-core multicore. As shown in Fig. 11, the total workload is divided along the rows of the image matrix into chunks of "iterations." The total count of iterations is $n_{\text{iter}} = N/ts$. These iterations are then dispatched either statically or dynamically to available threads. A thread t is assigned with iter_t number of iterations to execute sequentially. Workload assigned to all H threads add up to the total number of iterations ($\sum_{t=1}^{H} \text{iter}_t = n_{\text{iter}}$).

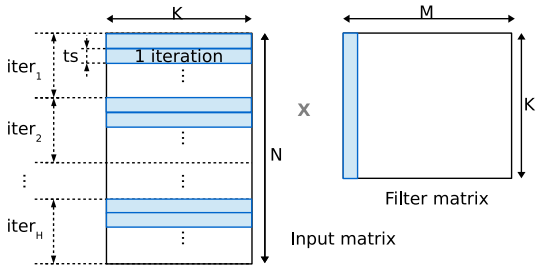


Fig. 11. Visualization of iteration allocation for convolutional layer among H threads.

TABLE III
GEMM EXECUTION TIME PREDICTION ERROR AVERAGED ACROSS ALL CONVOLUTIONAL LAYERS IN CNN FOR DIFFERENT POSSIBLE HOMOGENEOUS CORE ALLOCATIONS

CNN	1B	2B	3B	4B	1s	2s	3s	4s
AlexNet	11.3	11.9	12.3	13.1	9.6	10.5	10.5	11.1
GoogLeNet	13.8	15.0	15.1	15.0	8.8	9.5	9.6	8.9
MobileNet	21.5	19.5	17.2	17.7	18.6	17.1	17.2	18.5
ResNet50	8.2	7.5	8.0	8.4	11.5	10.9	11.1	12.1
SqueezeNet	18.1	17.9	18.0	17.7	13.9	13.0	11.8	12.7
Average	13.2%				11.4%			

For single-threaded execution, all iterations (n_{iter}) are assigned and processed sequentially on one thread, with execution time T obtained from (5). We model the time of each iteration (T_{iter}) from the single-threaded execution time with (6), assuming identical processing time for all iterations. For multithread execution, the execution time of the slowest thread determines the total time when we distribute the workload among H threads, as shown in (7) which models T_{multi}. Constant coefficients (α_1, α_2, α_3) are obtained using linear regression

$$T_{\text{iter}} = (T - \alpha_1)/n_{\text{iter}} + \alpha_2 \tag{6}$$
$$T_{\text{multi}} = \max_{t \in [1,H]} (T_{\text{iter}} * \text{iter}_t) + \alpha_3. \tag{7}$$

We can expect an equal split ($\text{iter}_t = n_{\text{iter}}/H = N/(ts * H)$) on the distribution of workload among homogeneous cores. Equation (8) combines the previous two equations and models the multithreaded execution time T_{multi} based on matrix size N, tile size ts, and the number of cores H

$$T_{\text{multi}} = T_{\text{iter}} * \text{iter}_t + \alpha_3 = T_{\text{iter}} * n_{\text{iter}}/H + \alpha_3$$
$$= (T - \alpha_1)/H + \alpha_2 * N/(ts * H) + \alpha_3. \tag{8}$$

Table III shows the prediction error for all the possible homogeneous core allocations. The proposed model predicts execution time for individual convolutional layers across all core configurations for all five benchmark CNNs accurately. We observed 13.2% and 11.4% prediction errors overall on

average for *Big* and *Small* cores, respectively. Our proposed performance-prediction model is significantly more advanced than the model presented in [22] focusing on performance prediction for the entire neural network. Model in [22] does not take into consideration the different number of cores involved in CNN execution. Their model is built and tested by profiling on all the available cores. For heavy layers, the workload is more likely to occupy all the cores and thus can be predicted with higher precision by Lu *et al.* [22]. However, for light layers, such a method cannot predict the reduction in utilization and thus results in higher errors. Lu *et al.* [22] only evaluated the model on the entire network and do not include layer-wise evaluations. They reported 13.4% prediction error for overall CNN inference time with only two CNNs. We reimplement the model in [22] on four *Big* cores and observe on average 15% estimation error for entire networks across five CNNs. However, we observe an average 54% error when using the same model to predict the execution time of a single layer. A huge error in layer-wise predictions makes the model in [22] unusable for *Pipe-it* that requires accurate per-layer performance estimation for workload allocation.

D. Fully Connected Layers

We also consider fully connected layers apart from convolutional layers as major layers in *Pipe-it*. Fig. 6 shows older networks like *AlexNet* spend a significant portion of their execution time executing fully connected layers. However, fully connected layers involve a huge number of parameters and hence result in excessive memory transfers during execution [20]. Newer CNNs usually adopt structures with no fully connected layers (*SqueezeNet*) or only one as classifier at the end (*GoogLeNet*, *MobileNet*, and *ResNet50*).

Fully connected layers are matrix multiplications. *AlexNet* has three fully connected layers with 4096, 4096, and 1000 neurons, respectively. Other networks employ fully connected layers as a classifier with 1000 neurons. We generate a set of micro-benchmarks with various input tensor sizes and number of neurons (4096 and 1000). Simple linearity is observed between input tensor sizes and execution time for a given number of neurons. Therefore, the regression-based model can also be used to predict the execution time of fully connected layers. We observed 11.8% and 14.4% prediction errors overall on average for the fully connected layers in micro-benchmarks and actual CNNs, respectively.

VI. Design Space Exploration

We can design many different pipelines with a different number of stages and each stage with different processing core combinations for a heterogeneous multicore. In addition, for fixed pipeline design, the number of design points in allocating the workload to different pipeline stages grows exponentially with the total number of convolutional layers. Therefore, we propose a robust heuristic approach that quickly navigates through the design space to obtain a high-performing layer-level split design point for any CNN. The heuristic uses an iterative two-step approach. The first step is to determine a workload split for a given pipeline configuration (Section VI-B). The second step is to merge adjacent stages to search for better pipeline configuration (Section VI-C). Two steps are iteratively engaged to approach a high-throughout pipeline configuration and corresponding workload distribution.

A. Definitions

Consider CNN with W convolutional layers to be deployed on $(H_B + H_s)$ heterogeneous multicore with H_B *Big* cores and H_s *Small* cores. The goal of *Pipe-it* is to find throughput maximizing pipeline configuration P and corresponding layer distribution L.

We use $P = \{P_1, \ldots, P_p\}$ to define core configuration of each pipeline stage for a pipeline P with p stages. We define the pipeline stage as tuple $P_i = (core\_type, core\_count)$ depicting type and count of cores that are used to construct it. The *core_type* can only be either B or s since only homogeneous cores are used to construct the pipeline stage. There are H_B and H_s core combinations for *Big* and *Small* cores, respectively. Therefore, $(H_B + H_s)$ different pipeline stage configurations are possible.

$L = \{L_1, \ldots, L_p\}$ defines corresponding layer allocation associated with the pipeline, where L_i is a set of layers allocated to pipeline stage P_i. $L_i = \{l_1, \ldots, l_W\}$ if *Pipe-it* allocates all the W layers to P_i. $L_i = \emptyset$ if it allocates none of the layers to P_i.

Section V describes performance-prediction models used to predict the execution time of a layer. We use time matrix T to represent predicted execution times. $T_{l_j}^{P_i}$ represents execution time for layer l_j on a core configuration P_i. Similarly, the following equation represents the execution time of the pipeline stage P_i with layer allocation L_i:

$$T_{L_i}^{P_i} = \sum_{l_j \in L_i} T_{l_j}^{P_i}. \tag{9}$$

B. Work-Flow Split Determination

We work with an assumption based on Fig. 7 that initial CNN layers are more compute-intensive than deeper layers and thereby requires more processing power. Thus, we order the pipeline stages to have more compute capable core combinations at the beginning, and with decreasing compute capability for stages deeper into the pipeline. Such an arrangement also ensures a monotonous increase in layer processing time as we move down pipeline stages. The compute capability of core combinations is evaluated by the execution time of layers on average. Equation (10) gives observed compute capability in executing layer l with homogeneous core combinations on our heterogeneous eight-core platform

$$\begin{aligned} T_l^{(B,4)} < T_l^{(B,3)} < T_l^{B,2} \lessapprox T_l^{(s,4)} \\ < T_l^{(s,3)} < T_l^{(s,2)} \lessapprox T_l^{(B,1)} < T_l^{(s,1)}. \end{aligned} \tag{10}$$

Equation (11) gives the throughput of pipeline P with p stages and layer allocation L. The pipeline stage that produces the longest latency determines the throughput of the pipeline. Therefore, the goal is to balance the workload among all stages

Algorithm 1 find_split: Algorithm to Split the Workload Between Adjacent Pipeline Stages

Input: $L_{wl} = \{l_a, \ldots, l_b\}, T^{P_i}, T^{P_{i+1}}$,
Output: L_i, L_{i+1}
Initialisation : $L_i = L_{wl} = \{l_a, \ldots, l_b\}$; $L_{i+1} = \emptyset$;
for $l_j \in L_{wl}$ **do**
  $T^{P_i}_{new} = T^{P_i}_{L_i} - T^{P_i}_{l_j}$;
  $T^{P_{i+1}}_{new} = T^{P_{i+1}}_{L_{i+1}} + T^{P_{i+1}}_{l_j}$;
  if $(T^{P_i}_{new} > T^{P_{i+1}}_{new})$ **then**
    $L_i = L_i \backslash \{l_j\}$; $L_{i+1} = L_{i+1} \cup \{l_j\}$ // move of l_j is helpful
  else
    break; //further flow of workload will not be helpful
  end if
end for
return L_i, L_{i+1}

to achieve minimal latency (maximum throughput)

$$\text{Throughput} = 1/\max_{i \in [1,p]} \left(T^{P_i}_{L_i}\right). \tag{11}$$

Algorithm 1 describes the division and allocation of a set of layers $L_{wl} = \{l_a, \ldots, l_b\}$ (in the original order) among two adjacent pipeline stages P_i and P_{i+1}. The ordering of pipeline stages ensures that any layer l_j is executed faster on P_i than on P_{i+1} ($T^{P_i}_{l_j} < T^{P_{i+1}}_{l_j}$). Such arrangement results in an expansion in execution time as we move deeper into the pipeline and thereby ensures one-way flow of workload.

The workload initially is entirely allocated to fastest stage P_i ($L_i = \{l_a, \ldots, l_b\}, L_{i+1} = \emptyset$) making it the bottleneck. We try to move layers to P_{i+1} to balance workload in each stage, starting with the last layer allocated to P_i (layer l_b). Moving layer l_j to P_{i+1} is helpful if $(T^{P_i}_{L_i} - T^{P_i}_{l_j} > T^{P_{i+1}}_{L_{i+1}} + T^{P_{i+1}}_{l_j})$. We keep moving the layers until l_k when P_{i+1} becomes bottleneck instead. Moving of more layers to stage P_{i+1} will make it even slower. Thus, the best split between two adjacent pipeline stage will be $L_i = \{l_a, \ldots, l_k\}$ and $L_{i+1} = \{l_{k+1}, \ldots, l_b\}$.

Pipe-it then goes to the next adjacent pipeline stages (P_{i+1} and P_{i+2}) to continue balancing stage latency. *Pipe-it* uses Algorithm 1 to go through all stages in pipeline to balance workload with its immediate next stage. We symbolize workload as water that flows from the first pipeline stage to deeper stages. There will be more space available in an initial stage once a part of workload flows from it to deeper stages. Therefore, *Pipe-it* engages Algorithm 1 iteratively to reach the final splitting configuration, wherein there is no further workload redistribution possible.

C. Pipeline Stage Merging

Running GEMM using multithreading is always beneficial. However, Fig. 12 shows saturating thread-level parallelism (TLP) can lead to concavity in multithreaded speedup gains with increasing core allocation. Furthermore, different types of layers derive different levels of benefits from multithreading. Therefore, it is important to match the size of the pipeline stage with speedup characteristics of layers allocated to it. Algorithm 3 describes the process of merging pipeline

Algorithm 2 work_flow: Algorithm for Workload Allocation for a Multistage Pipeline

Input: $P = \{P_1, ..., P_p\}, L_{wl} = \{l_1, ..., l_W\}, T = \{T^{P_1}, ..., T^{P_p}\}$,
Output: L
Initialisation : $L = \{L_1, ..., L_p\}$;
for $(L_i \in L)$ **do** $L_i = \emptyset$;
end for
$L_1 = L_{wl}$; $L_{old} = \emptyset$;
LOOP: Exit when allocation stabilized
while $L \neq L_{old}$ **do**
  $L_{old} = L$
  for $P_i, P_{i+1} \in P$ **do**
    $L_{temp} = L_i \cup L_{i+1}$
    $L_i, L_{i+1} =$ **find_split**$(L_{temp}, T^{P_i}, T^{P_{i+1}})$
  end for
end while
return L

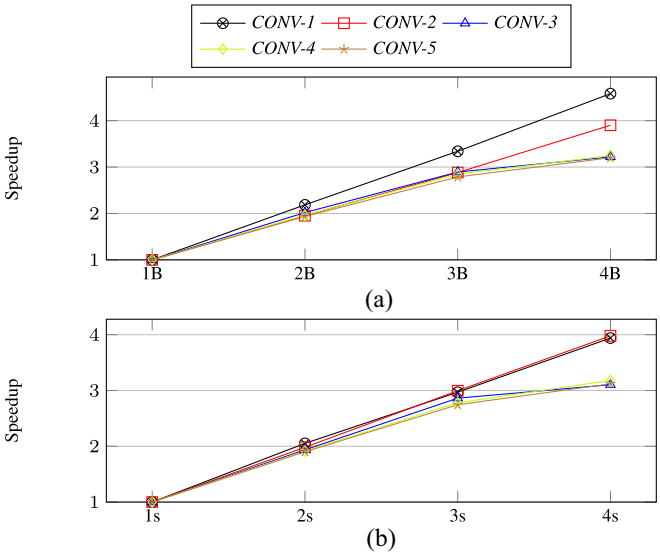


Fig. 12. Concavity in speedup for the five convolutional layers in *AlexNet* with different core configurations. (a) *Big* Core Configurations. (b) *Small* Core Configurations.

stages to create bigger pipeline stages. We consider *Big* cluster first before moving on to *Small* cluster.

We start with $(H_B + H_s)$-stage pipeline for $(H_B + H_s)$-core heterogeneous multicore, where each stage comprises of only one core. *Pipe-it* engages Algorithm 2 to search for the best split of workload for this pipeline configuration. The pipeline is likely to be bottlenecked by layers that require more compute capability given suboptimality of single-core performance. Thus, we merge pipeline stages to create a more compute capable stage to alleviate the bottleneck.

Consider the merger of stages $P_i = (core\_type, count_i)$ and $P_{i+1} = (core\_type, count_{i+1})$ to stage $P_{i'} = (core\_type, count_i + count_{i+1})$ with originally allocated set of layers L_i and L_{i+1}, respectively. Note that P_i and P_{i+1} must be of the same core type to merge.

Algorithm 3 merge_stage: Algorithm for Determining Stage Configuration and Corresponding Workload Allocation

Input: $L_{wl} = \{l_1, \ldots, l_W\}, H_B, H_s, T$
Output: P, L
Initialisation : $p = H_B + H_s$; $P = \{P_1, \ldots, P_p\}$;
$L = \{L_1, \ldots, L_p\}$;
1: L =**work_flow**(P, L_{wl}, T);
LOOP: Big cluster
2: **for** (P_i, P_{i+1}) in P **do**
3: **if** (**Equation** (12)) **then**
4: merge, update P; L =**work_flow**(...);
5: **else**
6: break; //stop further merging
7: **end if**
8: **end for**
LOOP: Small cluster
9: **for** (P_i, P_{i+1}) in P **do**
10: **if** (**Equation** (12)) **then**
11: merge, update P; L =**work_flow**(...);
12: **else**
13: break; //stop further merging
14: **end if**
15: **end for**
16: **return** P, L

The merging is only helpful when (12) holds, which implies new stage should be better in performance than at least one of two stages combined. Otherwise, we can stop as the concavity in speedup (Fig. 12) dictates no further merging of the involved stages to create an even bigger stage will be helpful either

$$T_{L_{i'}}^{P_{i'}} = T_{L_i}^{P_{i'}} + T_{L_{i+1}}^{P_{i'}} < \max\left(T_{L_i}^{P_i}, T_{L_{i+1}}^{P_{i+1}}\right). \tag{12}$$

Successful merge updates the pipeline configuration and reengages Algorithm 2 to find a new higher-performing layer split. Merging decision depends largely on layers allocated to the stage as different layers respond differently to different stage configurations. Therefore, the reallocation of workload is necessary for presenting the right layer information to the merging algorithm. Algorithm 3 runs iteratively until no further merging of stages is helpful.

D. Example

We illustrate with an example of how antecedent algorithms work to locate optimal pipeline configuration and workload allocation. The example considers deployment of *ResNet50* with 54 major layers (Table I) on an eight-core heterogeneous multicore with four *Big* and four *Small* cores. We can create eight different pipeline stages with different core combinations for this architecture. Therefore, eight different sets of layer execution time are predicted to generate time matrix T of size (54, 8). We plug the following corresponding inputs to Algorithm 3:

$$L_{wl} = \{l_1, l_2, \ldots, l_{54}\}; H_B = 4; H_s = 4.$$

Algorithm 3 initializes an eight-stage pipeline, wherein each stage consists of only a single core. It then engages Algorithm 2 to find split for the eight-stage pipeline.

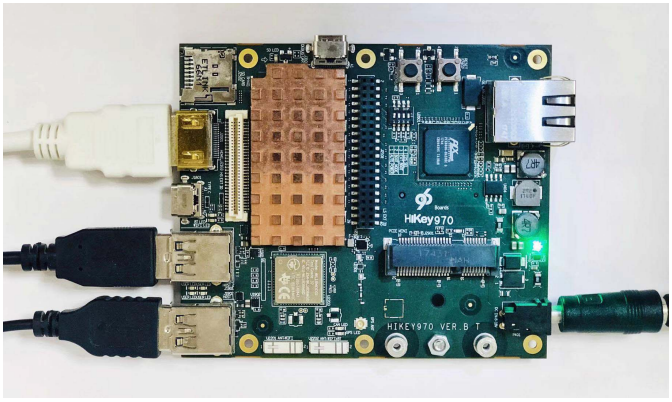


Fig. 13. Picture of *Hikey 970* mobile development board.

Algorithm 2 allocates all layers to the first pipeline stage P_1 at the beginning. It then engages Algorithm 1 to balance the workload between the first two stages (P_1 and P_2). Layers starting with the last layer allocated to P_1 (Layer l_{54}) are moved to stage P_2 for processing until two stages are balanced. Algorithm 1 returns $L_1 = \{l_1, \ldots, l_{25}\}, L_2 = \{l_{26}, \ldots, l_{54}\}$. We use l_{1-25} as a short-hand notation for $\{l_1, \ldots, l_{25}\}$. Thus, *Pipe-it* updates the workload allocation to $L = \{l_{1-25}, l_{26-54}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\}$.

Algorithm 2 then continues to balance workload between P_2 and P_3. Algorithm 2 repeats the process with the remaining pipeline stages. The first iteration returns $L = \{l_{1-25}, l_{26-38}, l_{39-46}, l_{47-50}, l_{51}, l_{52-54}, \emptyset, \emptyset\}$. The algorithm returns to rebalance workload of P_1 and P_2 again once it has rebalanced P_2 with P_3 and other stages. The iterative rebalancing, in the end, returns $L = \{l_{1-18}, l_{19-32}, l_{33-41}, l_{42-48}, l_{49-51}, l_{52-54}, \emptyset, \emptyset\}$. The last two pipeline stages are not allocated any workload because of poor computation capabilities. Therefore, the merging of stages is necessary to achieve higher performance.

Algorithm 3 evaluates a merger of the first two stages P_1 and P_2 to create a stage comprising of two *Big* cores ($(B, 2)$). Workload allocation is recalculated with Algorithm 2 if (12) holds. Otherwise, the merger of stages is not helpful. The algorithm will not try with further mergers. Merger in our case is helpful and algorithm updates the pipeline configuration to $P = \{(B, 2), (B, 1), (B, 1), (s, 1), (s, 1), (s, 1), (s, 1)\}$, with $L = \{l_{1-29}, l_{30-38}, l_{39-48}, l_{49-51}, l_{52}, l_{53-54}, l_{\emptyset}\}$. The algorithm then goes on merging P_1 and P_2 to create $(B, 3)$ and beyond. It recalculates allocation every time the pipeline stage is updated. The merge goes on for the *Small* cluster afterward following similar rules. *Pipe-it* finally decides upon a three-stage pipeline with configuration $P = \{(B, 4), (s, 2), (s, 2)\}$ and workload allocation $L = \{l_{1-35}, l_{36-44}, l_{45-54}\}$.

VII. Experimental Evaluation

We conduct experimental evaluations on *Hikey 970* mobile development platform [4] for five CNN models as specified in Table I. Fig. 13 shows a photograph of the board in use. The board features *ARM big.LITTLE* octa-core CPU with four-core *A73* and four-core *A53* cluster running at the maximum frequency of 2.4 and 1.8 GHz, respectively. It is connected to a normal desktop monitor through HDMI cable for display. It

TABLE IV
CNN THROUGHPUT COMPARISON OF HOMOGENEOUS VERSUS *Pipe-It* HETEROGENEOUS EXECUTION WITH PIPELINED PREDICTED FROM ACTUAL MEASURED AND PREDICTED LAYER EXECUTION TIME

CNN	Homogeneous Throughput (Imgs/s)		*Pipe-it* – Heteogeneous Throughput (Imgs/s)		Percentage Benefit (%)
	Big Cluster	*Small* Cluster	with measured layer time	with **predicted** layer time	
AlexNet	8.1	1.5	8.9	**8.9**	9.8
GoogLeNet	7.8	3.3	11.8	**11.3**	45.5
MobileNet	17.4	6.6	24.0	**23.5**	35.5
ResNet50	3.1	1.5	5.5	**5.2**	67.5
SqueezeNet	15.6	6.9	21.4	**21.4**	37.5
				Average	39.2%

comes equipped with an inbuilt WiFi module through which it can connect with a host machine over secure shell (SSH). Standard dc 5-V USB fan is used in experiments to eliminate unstable thermal effects.

We classify a continuous stream of 50 images and report average throughput (images processed per second) for each data point. The board is left idle for cooling down after each run resulting in approximately a 10-s run-time for each point. An exhaustive search on average size CNN with five million points would take hundred of days to run. Therefore, the run-time eliminates the possibility of obtaining the optimal configuration using an exhaustive search.

Recall that *kernel-level* split on all eight heterogeneous cores performs worse than four homogeneous *Big* cores. Therefore, our baseline configuration is *kernel-level* split on four homogeneous *Big* cores. This baseline provides the best possible throughput with default ARM-CL (Table IV).

A. Resultant Configurations

Table V shows the outcome of our DSE in the form of pipeline stages *P* and layer allocation *L*. We simplify notation for easier representation. For example, the pipeline configuration B4-s2-s2 for *ResNet50* implies three pipeline stages consisting of four *Big* cores, two *Small* cores, and two *Small* cores. *Pipe-it* allocates layers 1–35, 36–44, and 45–54 to the first, second, and third stage, respectively. Table IV shows the throughput of the respective pipelines.

In general, throughput benefit of *Pipe-it* comes from a deep and yet balanced pipeline configuration. *Pipe-it* can create a better-balanced pipeline with a large number of major layers in a network. Nevertheless, we still observe 20.6% benefit even for small networks like *LeNet* by using a three-stage pipeline designed by *Pipe-it*, compared to the default execution with four cores in the big cluster. *Pipe-it* on average improves throughput by 39% over baseline. Throughput obtained through pipelined configuration approaches or surpasses combined throughput of individual clusters for all CNNs.

B. Layer Performance-Prediction Model

We use micro-benchmarks to create our layer performance-prediction model. Predicted layer execution time guide the search for optimal configuration. Table III shows the model has good accuracy with on average overall prediction error of 13.2% and 11.4% for *Big* and *Small* clusters, respectively.

TABLE V
BEST THROUGHPUT PIPELINE CONFIGURATION WITH *Pipe-It* AND RESPECTIVE LAYER ALLOCATIONS FROM LAYER PERFORMANCE-PREDICTION MODEL

CNN	Pipeline Config.	Layer allocation
AlexNet	B4 - s4	[1,9] - [10,11]
GoogLeNet	B4 - s2 - s1 - s1	[1,29] - [30,41] - [42,45] - [46,58]
MobileNet	B2 - B2 - s3 - s1	[1,11] - [12,21] - [22,26] - [27,28]
ResNet50	B4 - s2 - s2	[1,35] - [36,44] - [45,54]
SqueezeNet	B4 - s4	[1,16] - [17,26]

TABLE VI
BEST THROUGHPUT PIPELINE CONFIGURATION WITH *Pipe-It* AND RESPECTIVE LAYER ALLOCATIONS FROM ACTUAL MEASURED LAYER TIMINGS

CNN	Pipeline Config.	Layer Allocation
AlexNet	B4 - s4	[1,9] - [10,11]
GoogLeNet	B4 - s2 - s1 - s1	[1,25] - [26,39] - [40,44] - [45,58]
MobileNet	B2 - B2 - s3 - s1	[1,11] - [12,19] - [20,26] - [27,28]
ResNet50	B2 - B2 - s3 - s1	[1,16] - [17,34] - [35,47] - [48,54]
SqueezeNet	B4 - s4	[1,19] - [20,26]

TABLE VII
BENEFIT OF *Pipe-It* ON A NONSTANDARD CONFIGURATION WITH THREE *Big* CORES AND TWO *Small* CORES

CNN	Throughput (Imgs/s)			Config.	Pct. Benefit (%)
	3 *Big*	2 *Small*	*Pipe-it*		
AlexNet	5.7	0.7	5.8	B3-s2	1.5
GoogLeNet	6.2	0.7	7.4	B3-s2	19.3
MobileNet	14.2	3.7	15.3	B3-s2	7.1
ResNet50	2.8	1.0	3.7	B3-s1-s1	31.5
SqueezeNet	11.4	3.6	13.5	B3-s2	18.1
				Average	15.5%

Table VI shows *Pipe-it* pipeline configurations with actual measure layer timings instead of predicted timings. The configurations in Tables V and VI are the same in most cases. There is a mere 4% difference in performance in the worst-case. Results establish the efficacy of our model.

C. General Applicability

Pipe-it is applicable across different heterogeneous multi-cores that have at least two clusters. We run *Pipe-it*, on the same *Hikey 970* platform, but with one *Big* core and two *Small* cores turned off to simulate an arbitrary *big.Little* CPU configuration. The layer timing estimations obtained as before are plugged into the DSE algorithm to locate the best pipeline configuration with the remaining three *Big* cores and two *Small* cores. Table VII shows the results obtained. *Pipe-it* predicts pipeline configurations with both clusters engaged. The performance benefit is not as significant compared to results shown in Table IV with four *Big* and four *Small* cores. This is because, in this CPU configuration, only two small cores are additionally engaged in the pipeline. Less additional resources result in lower performance improvement.

D. Power Efficiency

We are not able to obtain the individual power values of each CPU component due to lack of power sensors in our development board. We instead utilize a power measurement module [5] that supplies and measures whole board power

TABLE VIII
AVERAGE POWER (W) AND POWER-EFFICIENCY (IMGS/J) FOR EXECUTION ON HOMOGENEOUS CORES AND WITH *Pipe-It*

CNN	Average Active Power (W)			Power Efficiency (Imgs/J)		
	Big	*Small*	*Pipe-it*	*Big*	*Small*	*Pipe-it*
AlexNet	3.8	0.7	5.1	2.1	2.1	1.8
GoogLeNet	4.6	1.1	6.6	1.7	3.1	1.7
MobileNet	4.2	1.0	5.9	4.2	6.6	4.0
ResNet50	4.0	1.0	6.5	0.8	1.5	0.8
SqueezeNet	4.9	1.3	6.9	3.2	5.5	3.1

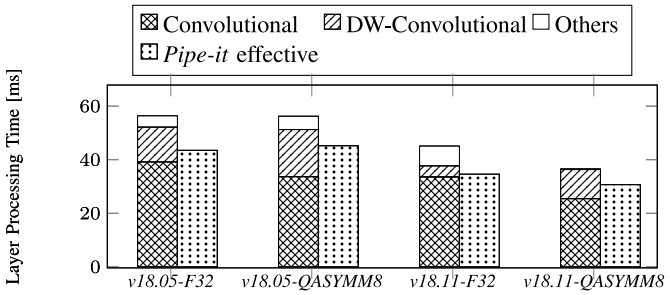


Fig. 14. Performance comparison of *MobileNet* with quantization across two ARM-CL versions (v18.05 and v18.11).

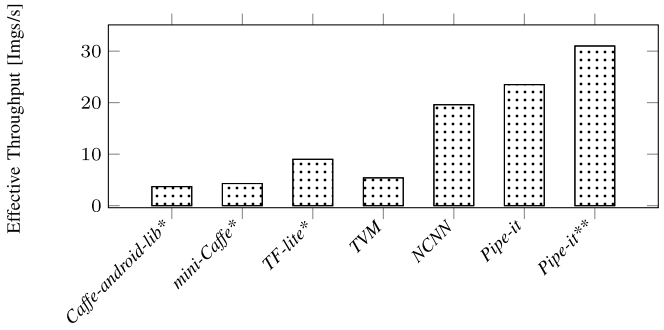
consumption. Cluster not engaged in execution during homogeneous runs is turned off to eliminate its contribution to total power consumption. Measured whole board socket power P includes everything on board beside CPU. We mitigate the effect of non-CPU components on total power by subtracting off idle power P_I. Idle power can vary with several factors. Therefore, we measure it again before each run. The active power readings reported are $P_A = P - P_I$. Table VIII shows power measurements and corresponding power-efficiency.

We cannot separate active memory power from the CPU. Therefore, power-efficient *Small* cluster shows lower than expected power-efficiency for memory intensive CNNs like *AlexNet*. We attribute the lower power-efficiency with *Pipe-it* to extra memory power consumed due to coherency between different core clusters.

E. Quantization Considerations

Pipe-it aims to improve throughput by engaging all on-chip CPU resources for execution. It is orthogonal to optimization techniques such as quantization [32]. ARM-CL provides support for execution with quantized 8 bits using asymmetric integers (QASYMM8). However, the benefit of quantization is largely dependent on the implementation. Overheads induced by de-quantization and requantization operations subdue the benefits of quantization [28]. Fig. 14 shows a similar effect by comparing the execution of nonquantized F32 and QASYMM8 for *MobileNet* with ARM-CL. Execution of convolutional layers is improved by 14%. However, overall execution time remains unchanged for *ARM-CL v18.05*.

We also evaluate the effect of quantization on the latest *ARM-CL version 18.11*. F32 implementation of *MobileNet* executes 20% faster on *ARM-CL v18.11* compared to *ARM-CL v18.05*. Its convolutional layers are 24% faster with quantization with an overall 19% faster execution.



* scaled performance with *AI-benchmark* [13].
** *Pipe-it* with *ARM-CL v18.11* and quantization as shown in Figure 14.

Fig. 15. Performance comparison of *MobileNet* with several frameworks.

The performance we report above is with homogeneous cores only. We create pipelines for both original and quantized *MobileNet* using *Pipe-it* across ARM-CL versions. Fig. 14 shows the effective per-frame latency (inverse of throughput). *Pipe-it* introduces further performance improvement in all implementations. *MobileNet* reaches a throughput of 31 Imgs/s with *Pipe-it* for its quantized *ARM-CL v18.11* implementation.

F. Comparison With Other Frameworks

We compare the performance of *Pipe-it* against other CNN frameworks using *MobileNet* as the common denominator. Fig. 15 shows the performance of several frameworks. We measure the Performance of *TVM*, *NCNN*, *Pipe-it*, and *Pipe-it*** with actual experiments on our platform. Performance numbers for the remaining frameworks are taken from other sources [3], [13]. The borrowed numbers are scaled approximately to compensate for differences in platforms. *Pipe-it* provides the highest performance amongst all cores.

We also compare the energy-efficiency of *Pipe-it* against *DeepX* [17]. *DeepX* is designed to consume the least power within a latency requirement. Lane *et al.* [17] evaluated *DeepX* on Qualcomm Snapdragon 800 SoC with *Krait* four-core 2.3-GHz CPU. *DeepX* provides a configuration which consumes 444 mJ of energy for *AlexNet* with the latency requirement of 500 ms (2 Imgs/s) resulting in energy-efficiency of 2.2 Imgs/J. *Pipe-it* achieves comparable energy-efficiency of 1.8 Imgs/J but with a much higher throughput of 8.9 Imgs/s.

VIII. RELATED WORK

The development of CNNs is moving toward more complex network structures with moderate resource requirements. Starting with 250 MB for AlexNet [16] in 2012, the size of models has reduced to less than 0.5 MB for SqueezeNet [13] in 2016 without losing accuracy. Such advancements allow for CNN deployment on mobile platforms even with their limited computational and memory resources. To effectively deploy CNN on embedded platforms, researchers are approaching from different angles. The network structure is modified to fit on the resource-constrained mobile platform, such as quantization [32] that accelerates the computation and reduces the memory usage, and network pruning [33] that compromise the

accuracy with fewer resource requirements. In addition, sparsity is exploited in NN applications [25] to reduce the computation and improve execution performance on edge devices.

Accelerators enable highly energy-efficient execution of CNNs on edge devices. Several works rely on the computational capability of embedded GPUs to enable CNN with collaborative execution on CPUs and other processors. *DeepX* [17] framework enables NN on edge through co-execution on multiple processors, including GPU and low power processors (LPUs). It first engages runtime layer compression to control the resource requirement of an NN workload. It then decomposites the workload into unit-blocks for assignment to multiple processors. *DeepX* derives substantial benefit in performance and energy for *AlexNet* mainly from its fully connected layers. Use of fully connected layers is now minimal in state-of-the-art CNNs. *DeepSense* [11] and *DeepMon* [10] present an *OpenCL*-based framework for mobile GPUs. *DeepSense* adopts GPU memory management techniques which accelerate compute-heavy executions including convolutional and fully connected layers execution on GPU. *DeepMon* extends *DeepSense* to include further caching optimizations and improves convolutional layer implementation. ASICs are now being designed specifically for neural network processing, such as *Google's* tensor processing unit (TPU) and *Huawei's* neural processing units (NPUs). Researchers also co-design algorithm and architecture with application-specific characteristics [34], [35].

Researchers have characterized resource requirements of CNNs [18], [22] that provide insights on designing CNN with resource-constraints. Efficient libraries [1], [3], [6], [19] are created to facilitate the implementation of deep learning on edge devices. Frameworks, like *CGOOD* [14] are created to facilitate deployment of CNN on edge devices by automatically generating *C* and GPU (*CUDA* or *OpenCL*) code that runs on respective platforms with hardware specifications and optimization requirements.

On the other hand, older technology node or cost-sensitive platforms that lack capable GPUs and accelerators still need to execute CNNs via their CPUs. *Graphi* [30] presents a framework that accelerates deep learning models through layer-level parallelism within NN on many-cores. It leverages on the inherent layer-level parallelism in network structure and schedules independent layers for concurrent execution. *Graphi* is beneficial for networks, such as *LSTM* and *GoogLeNet* that have high layer-level parallelism. In comparison, *Pipe-it* looks at computational *kernel-level* parallelism. It applies to general network structures and targets CNN acceleration on heterogeneous multicores.

IX. Conclusion

On-chip inference using CNNs is now becoming commonplace on edge devices. We show in this article that *kernel-level* splitting across heterogeneous core types is detrimental to throughput. Instead, *layer-level* splitting that minimizes cross-cluster coherency can be employed to improve inferencing throughout. We introduce a layer-level splitting technique called *Pipe-it* that efficiently uses entire heterogeneous multicores to improve CNN inference throughput. We study the design space involved and introduce a search algorithm to locate a high performing design point within it. *Pipe-it* improves the throughput on average by 39% using all heterogeneous cores in comparison to using only homogeneous cores. *Pipe-it* is not limited to CNN applications and also applies to other streaming applications that show similar behaviors. In future, we plan to include more co-processors, such as GPUs and NPUs into the design space to further exploit the potential of the embedded SoCs in enabling deep learning.

References

[1] *Compute Library: A Software Library for Computer Vision and Machine Learning*. Accessed: Oct. 9, 2019. [Online]. Available: https://developer.arm.com/ip-products/processors/machine-learning/compute-library

[2] *Gluon Model Zoo–MXNET Documentation*. Accessed: Oct. 9, 2019. [Online]. Available: https://mxnet.incubator.apache.org/api/python/gluon/model_zoo.html

[3] *NCNN: A High-Performance Neural Network Inference Framework Optimized for the Mobile Platform*. Accessed: Oct. 9, 2019. [Online]. Available: https://github.com/Tencent/ncnn

[4] "Hi3670 V100 application processor data sheet," HiSilicon Technol., Shenzhen, China, Rep., 2018. [Online]. Available: http://mirror.lemaker.org/Hi3670%20V100%20Application%20Processor%20Data%20Sheet.pdf

[5] "Keysight Technologies B2900 series precision source/measure unit user's guide," Keysight Technol., Santa Rosa, CA, USA, Rep., 2019. [Online]. Available: http://literature.cdn.keysight.com/litweb/pdf/B2910-90010.pdf

[6] T. Chen *et al.*, "TVM: End-to-end optimization stack for deep learning," *arXiv preprint arXiv:1802.04799*, pp. 1–15, 2018. [Online]. Available: https://arxiv.org/abs/1802.04799

[7] M. Damschen, F. Mueller, and J. Henkel, "Co-scheduling on fused CPU-GPU architectures with shared last level caches," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2337–2347, Nov. 2018.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 770–778.

[9] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017. [Online]. Available: https://arxiv.org/abs/1704.04861

[10] H. N. Loc, Y. Lee, and R. K. Balan, "DeepMon: Mobile GPU-based deep learning framework for continuous vision applications," in *Proc. Int. Conf. Mobile Syst. Appl. Services (MobiSys)*, 2017, pp. 82–95.

[11] H. N. Loc, R. K. Balan, and Y. Lee, "DeepSense: A GPU-based deep convolutional neural network framework on commodity mobile devices," in *Proc. ACM Workshop Wearable Syst. Appl. (WearSys)*, 2016, pp. 25–30.

[12] F. N. Iandola *et al.*, "SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and < 0.5 MB model size," *arXiv preprint arXiv:1602.07360*, 2016. [Online]. Available: https://arxiv.org/abs/1602.07360

[13] A. Ignatov *et al.*, "AI benchmark: Running deep neural networks on Android smartphones," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 288–314.

[14] D. Kang, E. Kim, I. Bae, B. Egger, and S. Ha, "C-GOOD: C-code generation framework for optimized on-device deep learning," in *Proc. ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2018, p. 105.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1097–1105.

[16] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *Proc. Int. Symp. Microarchitect. (ISCA)*, 2003, p. 81.

[17] N. D. Lane *et al.*, "DeepX: A software accelerator for low-power deep learning inference on mobile devices," in *Proc. Int. Conf. Inf. Process. Sensor Netw. (IPSN)*, 2016, p. 23.

[18] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, and F. Kawsar, "An early resource characterization of deep learning on wearables, smartphones and Internet-of-Things devices," in *Proc. ACM Int. Workshop Internet Things Towards Appl. (IoT-App)*, 2015, pp. 7–12.

[19] S. S. L. Oskouei, H. Golestani, M. Hashemi, and S. Ghiasi, "CNNdroid: GPU-accelerated execution of trained deep convolutional neural networks on Android," in *Proc. ACM Int. Conf. Multimedia (MM)*, 2016, pp. 1201–1205.

[20] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013. [Online]. Available: https://arxiv.org/abs/1312.4400

[21] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing CNN model inference on CPUs," in *USENIX Annu. Tech. Conf. (USENIX ATC '19)*, 2019, pp. 1025–1040.

[22] Z. Lu, S. Rallapalli, K. Chan, and T. L. Porta, "Modeling the resource requirements of convolutional neural networks on mobile devices," in *Proc. ACM Int. Conf. Multimedia (MM)*, 2017, pp. 1663–1671.

[23] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *Proc. ACM Design Autom. Conf. (DAC)*, 2013, p. 174.

[24] A. Pacheco, P. Cano, E. Flores, E. Trujillo, and P. Marquez, "A smart classroom based on deep learning and osmotic IoT computing," in *Proc. IEEE Congreso Internacional de Innovación y Tendencias en Ingeniería (CONIITI)*, 2018, pp. 1–5.

[25] S. Sen, S. Jain, S. Venkataramani, and A. Raghunathan, "SparCE: Sparsity aware general-purpose core extensions to accelerate deep neural networks," *IEEE Trans. Comput.*, vol. 68, no. 6, pp. 912–925, Jun. 2019.

[26] N. Smolyanskiy, A. Kamenev, J. Smith, and S. Birchfield, "Toward low-flying autonomous MAV trail navigation using deep neural networks for environmental awareness," in *Proc. Int. Conf. Intell. Robots Syst. (IROS)*, 2017, pp. 4241–4247.

[27] T. S. Muthukaruppan, A. Pathania, and T. Mitra, "Price theory based power management for heterogeneous multi-cores," *SIGPLAN Notices*, vol. 49, no. 4, pp. 161–176, 2014.

[28] D. Sun, S. Liu, and J.-L. Gaudiot, "Enabling embedded inference engine with ARM compute library: A case study," *arXiv preprint arXiv:1704.03751*, 2017. [Online]. Available: https://arxiv.org/abs/1704.03751

[29] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2015, pp. 1–9.

[30] L. Tang, Y. Wang, T. L. Willke, and K. Li, "Scheduling computation graphs of deep learning models on manycore CPUs," *arXiv preprint arXiv:1807.09667*, 2018. [Online]. Available: https://arxiv.org/abs/1807.09667

[31] C.-J. Wu *et al.*, "Machine learning at Facebook: Understanding inference at the edge," in *Proc. Int. Symp. High Perform. Comput. Architect. (HPCA)*, 2019, pp. 331–344.

[32] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 4820–4828.

[33] T.-J. Yang *et al.*, "NetAdapt: Platform-aware neural network adaptation for mobile applications," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 285–300.

[34] Y. Zhu, M. Mattina, and P. Whatmough, "Mobile machine learning hardware at ARM: A systems-on-chip (SoC) perspective," *arXiv preprint arXiv:1801.06274*, 2018. [Online]. Available: https://arxiv.org/abs/1801.06274

[35] Y. Zhu, A. Samajdar, M. Mattina, and P. Whatmough, "Euphrates: Algorithm-SoC co-design for low-power mobile continuous vision," in *Proc. IEEE Int. Symp. Comput. Architect. (ISCA)*, 2018, pp. 547–560.

[36] A. Zlateski, K. Lee, and H. S. Seung, "ZNN—A fast and scalable algorithm for training 3D convolutional networks on multi-core and many-core shared memory machines," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2016, pp. 801–811.

Siqi Wang received the B.Eng. degree (Hons.) in electrical engineering from the National University of Singapore, Singapore, in 2014, where she is currently pursuing the Ph.D. degree with the Department of Computer Science, School of Computing, National University of Singapore.

She is currently a Research Assistant with the Department of Computer Science, School of Computing, National University of Singapore. Her current research interests include performance and energy optimization, task scheduling, general-purpose GPUs, and deep learning on heterogeneous multiprocessor embedded systems.

Gayathri Ananthanarayanan received the B.E. degree (Hons.) in electronics and instrumentation engineering and the master's degree in embedded systems from the Birla Institute of Technology and Science, Pilani, Pilani, India, in 2008 and 2010, respectively, and the Ph.D. degree from the Indian Institute of Technology Delhi, New Delhi, India, in 2017.

She was a Post-Doctoral Researcher with the School of Computing, National University of Singapore. She is currently an Assistant Professor with the Department of Computer Science and Engineering, Indian Institute of Technology Dharwad, Dharwad, India. Her current research interests include computer architecture and embedded systems, with specific focus on power, thermal and performance modeling, characterization, and management.

Yifan Zeng received the B.Eng. degree in electrical and electronic engineering from the Southern University of Science and Technology, Shenzhen, China, in 2018. He is currently pursuing the Ph.D. degree with the Department of Computer Science, School of Computing, National University of Singapore, Singapore.

His current research interests include energy efficient computation in deep learning and neural network compression.

Neeraj Goel received the B.Tech. degree in electronics and communication engineering from NIT Kurukshetra, Kurukshetra, India, in 2002, the M.Tech. degree in VLSI design tools and technology and the Ph.D. degree from the Indian Institute of Technology Delhi, New Delhi, India, in 2004 and 2012, respectively.

He is an Assistant Professor with the Department of Computer Science and Engineering, Indian Institute of Technology Ropar, Rupnagar, India. His current research interests include processor architecture, system on chip design and modeling, low power design, behavior synthesis, retargetable code generation, and compiler optimizations.

Anuj Pathania received the Ph.D. degree from the Karlsruhe Institute of Technology, Karlsruhe, Germany, in 2018.

He is currently a Research Fellow with the National University of Singapore, Singapore. His current research interest includes resource management algorithms with emphasis on performance-, power-, and thermal-efficiency in embedded systems. He has published papers in top peer-reviewed conferences and journals in the above areas.

Tulika Mitra received the B.E. degree in computer science from Jadavpur University, Kolkata, India, in 1995, the M.E. degree in computer science from the Indian Institute of Science, Bengaluru, India, in 1997, and the Ph.D. degree from the State University of New York, Stony Brook, NY, USA, in 2000.

She is a Professor of Computer Science with the School of Computing, National University of Singapore, Singapore. Her current research interest includes design automation of embedded real-time systems with particular emphasis on software timing analysis/optimizations, application-specific processors, energy-efficient computing, and heterogeneous computing.