Integrated ARM big.Little-Mali Pipeline for High-Throughput CNN Inference

Ehsan Aghapour, Gayathri Ananthanarayanan, and Anuj Pathania

Abstract—State-of-the-art Heterogeneous System on Chips (HMPSoCs) can perform on-chip embedded inference on its CPU and GPU. Multi-component pipelining is the method of choice to provide high-throughput Convolutions Neural Network (CNN) inference on embedded platforms. In this work, we provide details for the first CPU-GPU pipeline design for CNN inference called *Pipe-All*. *Pipe-All* uses the *ARM-CL* library to integrate an *ARM big.Little* CPU with an *ARM Mali* GPU. *Pipe-All* is the first three-stage CNN inference pipeline design with ARM's *big* CPU cluster, *Little* CPU cluster, and *Mali* GPU as its stages. *Pipe-All* provides on average 75.88% improvement in inference throughput (over peak single-component inference) on *Amlogic A311D* HMPSoC in *Khadas Vim 3* embedded platform. We also provide an open-source implementation for *Pipe-All*.

Keywords—Convolutional Neural Networks (CNNs), throughput, on-edge inference, Heterogeneous System on Chips (HMPSoCs).

1 Introduction

Heterogeneous Multi-Processor System On Chips (HMPSoCs) combines several processors (such as an embedded CPU and GPU) on a single chip. Figure 1 shows the abstract block diagram for the state-of-the-art *Amlogic A311D* HMPSoC within *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 itself contains two CPU clusters – a high-performance high-power quad-core *big* CPU cluster and a low-performance low-power dual-core *Little* CPU cluster.

Both the CPU clusters and GPU are capable of performing Machine Learning (ML) inference by deploying state-of-the-art conventional Convolutional Neural Networks (CNNs) [8]. Though it is common in non-embedded platforms for GPUs to significantly outperform the CPUs while inferencing, embedded GPUs (given their constrained design) deliver performance comparable to the embedded CPUs in embedded platforms. Therefore, CPUs are still relevant for inference in embedded platforms and must not be ignored [9].

Figure 2 shows the stand-alone inference throughput for both CPU clusters and GPU on *Khadas Vim 3* embedded platform. The results show that depending upon the network, either the *big* CPU cluster or GPU can provide the peak single-component performance. The *Little* CPU cluster provides a comparatively smaller but still significant throughput. However, in absolute terms, the single-component performance is still low. None of the benchmarks can attain even 30 Frames per Second (FPS), the least recommended FPS for basic user experience [5] running on CPU and GPU alone. Therefore, it is best to use both CPU clusters and GPU simultaneously for boosting inference throughput in embedded platforms.

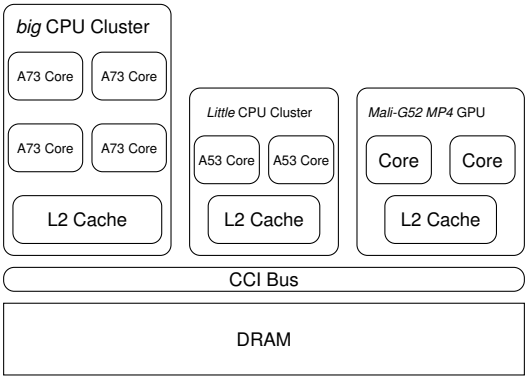


Fig. 1: An abstract block diagram of *Amlogic A311D* HMPSoC in *Khadas Vim 3* embedded platform.

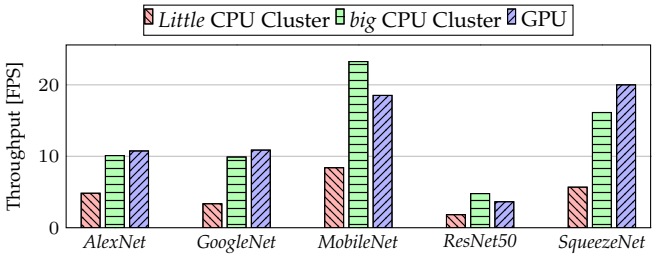


Fig. 2: The inference throughput for different inference capable components for different CNNs on *Khadas Vim 3*.

A common approach to boost throughput is to employ multi-component inference. A CNN consists of several layers that are generally processed sequentially. However, direct multi-component inference wherein a given layer from a given frame (kernels within) processes simultaneously on both CPU clusters in an asymmetric multi-core is detrimental to the inference performance [7]. Authors of [7] attribute the drop in performance to the large memory traffic generated on the interconnect to maintain cache coherence between components while processing the same layer. Furthermore, there is no known framework capable of processing a given layer from a given frame simultaneously on CPU and GPU.

One of the best-known approaches to employ multi-component inference that leads to a boost in inference throughput is to use a pipeline design that operates at the layer-level granularity. In such a pipeline design, each inference capable component act as one stage of the pipeline. Multiple frames are then processed through the pipeline simultaneously, wherein if one stage is processing layers from Frame N, then the other stage process the layers from Frame N+1. Consecutive layers

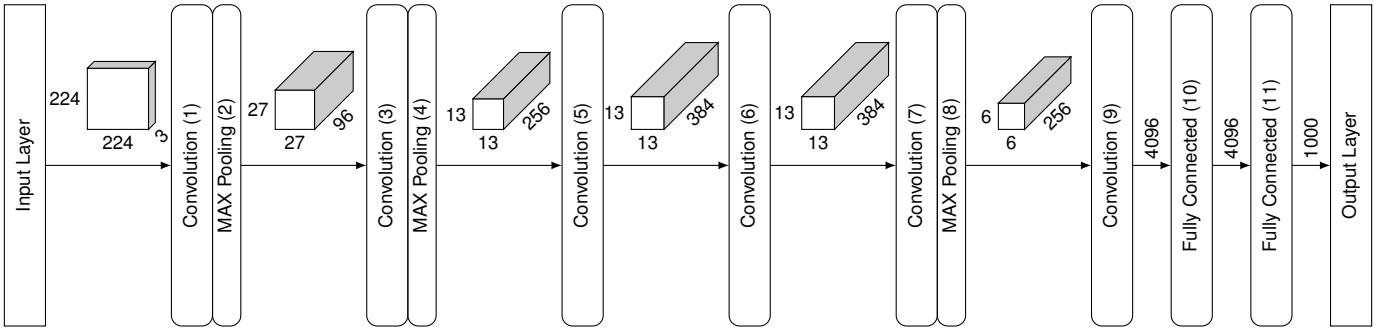


Fig. 3: The CNN architecture for *Alexnet*.

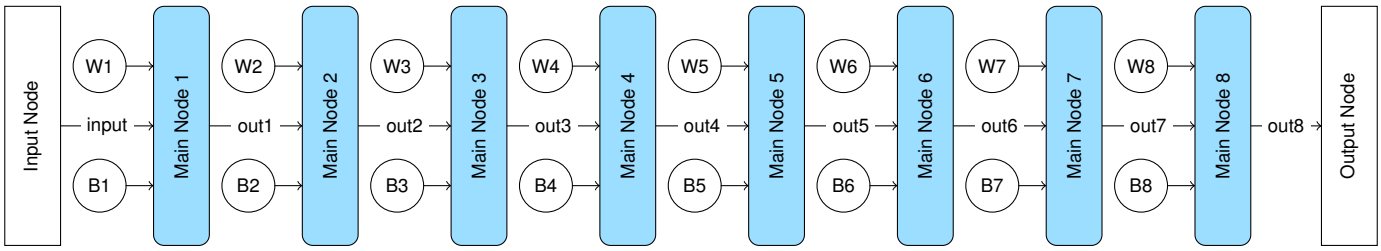


Fig. 4: The graph for *Alexnet* in *ARM-CL* corresponding to its CNN architecture.

of the CNNs are processed in the same pipeline stage, as much as possible, to minimize off-component coherence traffic on the interconnect as the output of the preceding layer is often the input for the succeeding layer.

A CPU-GPU inference pipeline design is best suited for HMPSoCs, wherein a CPU and a GPU can perform inference using the same framework. In such HMPSoCs, we can tightly integrate the CPU and GPU (as a single binary) with minimal overhead. Therefore, we use *ARM-CL* to create a pipeline between an *ARM big.Little* processor and an *ARM Mali* GPU in this work. *ARM-CL* library in its default release provides support for highly optimized stand-alone single-component CNN inference on both ARM CPU clusters or GPUs out-of-the-box. We build upon the default implementation to create a multi-component ARM CPU-GPU inference pipeline. We introduce a three-stage pipeline design, called *Pipe-All*, that has *big* CPU cluster, *Little* CPU cluster, and GPU as its stages.

Our Novel Contributions: We make the following novel contributions in this work.

- Our work is the first to create a tightly integrated three-stage CPU-GPU pipeline between ARM *big.Little* CPU and *Mali* GPU using *ARM-CL*, called *Pipe-All*, for CNN inference. We describe the implementation in detail.
- We implement and evaluate *Pipe-All* in *Amlogic A311D* HMPSoC in *Khadas Vim 3* embedded platform, wherein it provides 75.88% throughput improvement, on average, over peak single-component inference throughput.

Open Source Contributions: The code for *Pipe-All* is publicly available for download at *https://github.com/Ehsan-aghapour/ARMCL-Pipe-All* under *MIT* license.

2 Related Work:

Multi-component inference through pipelining is an active subject of research. The authors of [7] were the first to create a layer-level inferencing pipeline between *big* and *Little* CPU clusters in an *ARM big.Little* asymmetric multi-core processor to improve CPU's inference throughput. Their work also employs *ARM-CL*. However, their pipeline design works by migrating CPU threads between *big* and *Little* cores. One cannot extend this design to include a GPU as no one can migrate CPU threads to a GPU. Authors of [6], [10] propose techniques to optimize the inference pipelines on asymmetric multi-cores.

Recently, there have been works [2], [3], [4] that propose to use CPU and GPU synergistically to improve CNN inference throughput on embedded platforms with *Nvidia* GPUs using the *TensorRT* framework. These works use GPU primarily as an accelerator to offload computations from a CPU-only pipeline design. However, the CPUs used in these works are of symmetric design. These designs form a potential alternative to the pipeline design we propose in this work. Nevertheless, direct comparison between the two designs is difficult given the differences in the platform and frameworks. Finally, to the best of our knowledge, none of the works above have made their design open-source.

In contrast to the above works, we are the first to introduce an open-source integrated three-stage pipeline design, called Pipe-All, with the big CPU cluster, Little CPU cluster, and GPU as its stages.

3 Inference with ARM-CL

In this section, we elaborate on how one performs inference with the help of ARM-CL. This information is quintessential to comprehend the *Pipe-All*'s implementation details we present in the next section.

Framework: *ARM-CL* is a collection of low-level machine learning functions optimized for *ARM Cortex-A* CPU and *Mali* GPU cores. The library provides ML acceleration on *Cortex-A* CPU through *Neon* (or *SVE*) and acceleration on *Mali* GPU through *OpenCL* [1]. By design, a CNN consists of some hidden layers sandwiched between the input and output layers. Each hidden layer takes in some inputs, which it then processes to produce an output. The output from a hidden layer forms one of the inputs for the next hidden layer connected to it. Weights and biases, specific to a layer, form the layer's other inputs.

Figure 3 shows the CNN architecture for *AlexNet*. *ARM-CL* uses a graph to represent CNN. Figure 4 shows the *ARM-CL* graph corresponding to the *Alexnet* CNN architecture.

In an *ARM-CL* graph, an `Input` and `Output` node represents the input and output layer of the CNN, respectively. There exist a `Main` node for each hidden layer in the graph. The graph connects the `Input` and `Output` node through a series of sequentially connected `Main` nodes. The number of `Main` nodes is not required to be equal to the number of layers in the CNN. For example, the graph subsumes minor non-convolution layers such as max-pooling layers in *AlexNet* in the `Main` nodes corresponding to convolutions layers. The graph also connects each `Main` node with its two exclusive `Weight` and `Bias` nodes. `Weight` and `Bias` node provides the weights and biases input to the `Main` node, respectively. The primary input for the `Main` node itself comes from the preceding `Main` (or `Input`) node in the graph. After processing, the `Main` node sends its output to the succeeding `Main` (or `Output`) node in the graph. Therefore, the graph binds the nodes in a nexus of consumer-producer relationships. In the *ARM-CL* graph, edges provide the connections between the nodes. For each edge, there is a unique tensor that provides the memory for the data getting transferred.

Environment Setup: *ARM-CL* provides (Application Programming Interfaces) APIs for users to define layers of a CNN and then connect them. While setting up the execution environment, *ARM-CL* begins by generating a graph corresponding to the user-defined CNN. It then sets up the back-end context on the target processor (CPU or GPU). For the CPU, the setup includes the generation of 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 first extracts the details such as the number of cores and model number. Then, it creates an *OpenCL* context with a `CLScheduler` optimized for the detected GPU device.

After setting up context for the back-end, *ARM-CL* determines the features of tensors (such as their shape and data types) based on the producing nodes of the edges. For each type of the `Main` node, there is a corresponding highly optimized implementation of kernel functions in *ARM-CL* to support its execution. *ARM-CL* selects and configures the optimal implementation for each node based on specifications of the underlying hardware and the dimensions of its operands. Consequently, it assigns memory to the tensors corresponding to the weights and biases and load them with values. It then serializes the kernels and prepares them for execution in the correct sequence on the target processor.

Running the Graph: *ARM-CL* sends the frame (initial input) to the `Input` node to trigger the processing of the graph on the CPU. After loading the frame, kernels start processing the data within. Kernels primarily perform matrix operations on the data. If the target processor is CPU, *ARM-CL* partitions the computations within the matrix operation and distributes them between the CPU worker threads. Threads after performing the computation, fill the results in the corresponding output tensors. The process continues till all kernels (`Main` nodes) have finished execution. *ARM-CL* puts the output from the last `Main` node in the input tensor of the `Output` node. The `Output` node then makes the decision based on the values in this tensor. If the target processor is GPU, after loading the frame (initial input) in the `Input` node, *ARM-CL* pushes the kernels to the *OpenCL* queue instead of CPU worker threads. *OpenCL* then executes the kernel on the GPU cores.

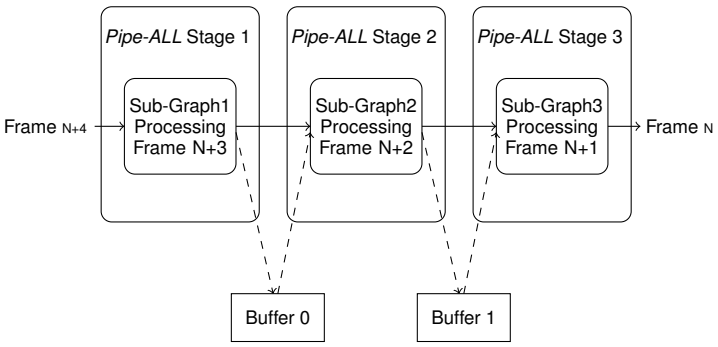


Fig. 5: An abstract block diagram showing high-throughput (low-latency) pipelined inferencing of a stream on a HMPSoC using *Pipe-All* design.

Graph	Layers	Partition Points	Design Space	Search Time
AlexNet	11	7	126	2 Hours
GoogleNet	58	12	396	5 Hours
MobileNet	28	27	2106	11 Hours
ResNet50	54	17	816	8 Hours
SqueezeNet	26	18	918	7 Hours

TABLE 1: The design space parameters for different CNNs under *Pipe-All*.

4 *Pipe-All* CPU-GPU Pipeline Design

The pipelined design within *Pipe-All* proposed in this work process three separate frames simultaneously on *Little* CPU, *big* CPU, and *GPU*, as shown in Figure 5 with an abstract block diagram. However, it doesn't process these frames in their entirety in any component. The pipeline allows us to distribute the processing of a given frame between the three components at node-level (near layer-level) granularity. The processing distribution between the components inversely correlates to their inference capabilities for the given CNN. For example, the *Little* CPU cluster always receives the least processing load for a frame as it is the weakest inference-capable component in the HMPSoC for all CNNs. Nevertheless, this distribution allows all the frames processed on HMPSoC with *Pipe-All* to have the same latency. The slowest stage in the pipeline determines the inference latency with the pipeline.

Design Space: Since the number of stages in *Pipe-All* is only three and the processing distribution between the stages must maintain the sequential layer-wise processing order, the design space for *Pipe-All* is small. Table 1 shows the number of layers and number of partition points in different CNNs. The number of partition points is less than the number of layers because non-convolution layers (except fully-connected layers) do not have a `Main` node associated with them in the *ARM-CL* graph and are therefore not viable partition points. Furthermore, within *Pipe-All*, all `Main` nodes that can be processed independently are grouped into a single partition point and processed simultaneously in parallel to maximize throughput.

Let N be the number of potential partition points in a CNN. To create a three-stage pipeline, we have to choose two partition points for splitting. Since it is always beneficial for throughput to engage a component in inferencing, we ignore the possibilities of empty pipeline stages. So, if we choose to perform the first split at the first partition point, there are $N-1$

partition points $(2, ..., N)$ to choose from for the second split. Similarly, if we choose to perform the first split at the second partition point, there are $N-2$ partition points $(3, .., N)$ to choose from for the second split. Therefore, under *Pipe-All*, there can be $(N-1)+(N-2)+....+1 = N \cdot (N-1)/2$ potential ways to split the inference workload in the three-stage pipeline. Since each stage can go on to any of three components (*little* CPU cluster, *big* CPU cluster, and GPU), there are 3! ways to assign a given workload split between the component.

Therefore, there are $6 \cdot N \cdot (N-1)/2$ different pipeline configurations possible for a CNN with N partition points under *Pipe-All*. Since the CNN inference workload is static, these configurations can be reliably profiled at design time for throughput quickly. Even for *MobileNet* with the highest number of configurations (2106 configurations) among all CNNs, we can do this in 11 hours on our platform. Therefore, we obtain the optimal configuration (with the highest throughput) for all CNN using an exhaustive search. Table 1 shows the number of configurations and time for the exhaustive search for different CNNs. We leave the option of creating faster design space exploration algorithms open as future work.

5 Implementing *Pipe-All* in ARM-CL

We now present the new modifications done to the *ARM-CL* library that enable the pipeline design within *Pipe-All*.

Sub-Graphs: We introduce the concept of sub-graphs to *ARM-CL*. We add new APIs to *ARM-CL* that allow users to partition the network (defined using existing *ARM-CL* APIs) into sub-networks. Instead of producing a single graph for the entire network, we modify *ARM-CL* to produce multiple sub-graphs, one for each user-defined sub-network.

Like a graph in *ARM-CL*, the sub-graph contains `Main`, `Weight`, and `Bias` nodes, and optionally an `Input` node or an `Output` node. The functionality of these nodes is the same in sub-graphs as it was in the original *ARM-CL* graph. In addition to these nodes, we introduce two new nodes in sub-graphs – a `Transmitter` node and a `Receiver` node. A `Receiver` node is at the start of the sub-graph unless it is the sub-graph representing the first sub-network. In that case, the `Input` node replaces the `Receiver` node. The role of the `Receiver` node is to receive input data from the preceding sub-graph. There is a `Transmitter` node at the end of the sub-graph unless it is the sub-graph representing the last sub-network. In that case, the `Output` node replaces the `Transmitter` node. The role of the `Transmitter` node is to transmit output data to the succeeding sub-graph.

Pipeline Setup: *Pipe-All* employs a three-stage pipeline. We divide a CNN into three sub-graphs, one for each stage of the pipeline. Input tensors for the second and third sub-graphs get their shapes from output tensors of the first and second sub-graphs, respectively. The selected *Pipe-All* configuration provides the parameters which determine the size of the sub-graphs. Figure 6 shows an example trio of *ARM-CL* sub-graphs for *AlexNet*, wherein we split the CNN at the fourth and sixth layers. Technically, it is possible to create up to N sub-networks (sub-graphs) for a CNN with N layers with our new *ARM-CL* APIs to support an N-stage pipeline. However, we leave that option open for exploration in the future.

Environment Setup: We set up the execution environment for each sub-graph depending upon the target processor it expects to execute on. For a sub-graph running on the *big*

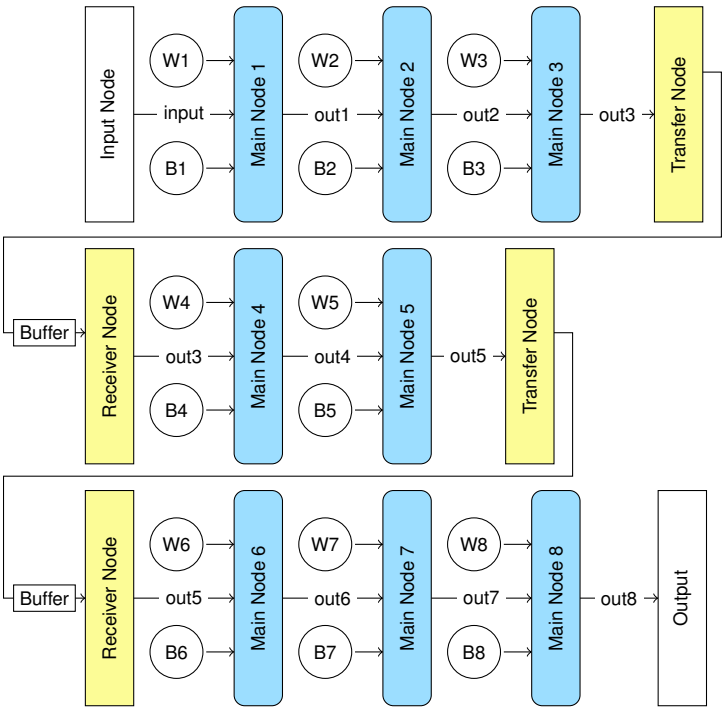
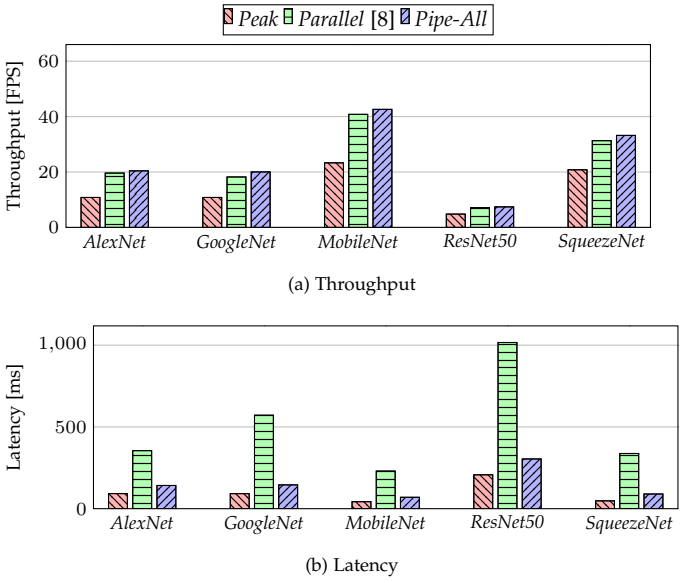


Fig. 6: The three sub-graphs for *Alexnet* obtained from partitioning it into three sub-networks.

or *Little* CPU cluster, we create worker threads equal to the number of *big* and *Little* cores in the CPU, respectively. To prevent worker threads from migrating between the clusters at will, we also pin the threads to corresponding *big* or *Little* cores (using `taskset`) in the cluster under a one-thread per-core model. For the sub-graph running on the GPU, we set up an *OpenCL* context with a `CLScheduler` optimized for the detected *ARM Mali* GPU. We then prepare the kernels corresponding to the *Main* nodes within the sub-graph. Finally, we load the weights and biases in the `Weight` and `Bias` nodes within the sub-graph. The sub-graph is now ready to execute.

Pipeline Frame Processing: The selected *Pipe-All* configuration determines the one-to-one mapping between the three sub-graphs and three processors (*big* CPU Cluster, *Little* CPU Cluster, and GPU) in the pipeline. Since it is infeasible to have a perfectly balanced pipeline, we use two buffer tensors to synchronize exchange between the pipeline stages, as shown in Figure 5. We then trigger a run-time daemon to initiate processing and subsequent run-time management. The run-time daemon pushes the first frame in the streaming queue to the initial sub-graph with the `Input` node. The initial sub-graph then starts processing the first frame. Once it finishes processing the sub-frame, it pushes the processed data into the first-stage buffer tensor through its `Transmitter` node. The middle sub-graph pulls the data from the first-stage buffer tensor into its input tensor through its `Receiver` node. It then processes the data and then pushes the processed data to the second-stage buffer tensor through its `Transmitter` node. The final sub-graph then pulls the data from the second-stage buffer tensor into its input tensor through its `Receiver` node. After processing, it pushes the final output data to the output node signaling the end of processing. Given its pipelined design, stages in *Pipe-All* are free to process kernels from another frame as soon as they finish processing kernels from their current frame.



(a) Throughput

(b) Latency

Fig. 7: Results for different CNNs with different approaches.

6 Experimental Evaluation

Experimental Setup. We use *Amlogic A311D* HMPSoC in *Khadas Vim 3* embedded platform, as shown in Figure 1, for evaluating *Pipe-All*. It contains a hexa-core asymmetric *ARM big.Little* multi-core CPU with two CPU clusters, *big* and *Little*. The quad-core *big* CPU cluster contains four *A73* cores. The dual-core *Little* CPU cluster contains two *A53* cores. HMPSoC also contains a dual-core *Mali G52 MP4* GPU. The maximum frequency for *big* CPU cluster, *Little* CPU cluster, and GPU is 1.8 GHz, 2.2 GHz, and 0.8 GHz, respectively. Though both CPU clusters and GPU support Dynamic Voltage Frequency Scaling (DVFS), in this work, we run them only at their maximum frequency, given our emphasis only on performance. A 4 GB LPDDR4 main memory supports the HMPSoC. In software, the platform is running *Android* v9.0 with kernel v4.9. We run *ARM-CL* v21.02 on top of it.

Metrics: We evaluate *Pipe-All* on two metrics – throughput measured in FPS and latency measured in milliseconds (ms). Ideally, we want the highest throughput with minimal latency.

Baselines: We use two baselines to evaluate the efficacy of *Pipe-All*. The first baseline is the peak single-component inference performance, symbolized by *Peak*. We obtain the second baseline, called *Parallel*, by performing independent simultaneous inference on all inference-capable components of HMPSoCs, as suggested by [8].

Performance Evaluation: Figures 7a and 7b show the throughput and corresponding latency that we can obtain using different techniques for different CNNs on our setup. *Peak* baseline has a low throughput since only the highest performing component for a CNN is engaged in inferencing while other components are idle. However, this approach has a very low latency which also forms an empirical lower bound of the latency we can achieve with *Pipe-All*. On the other hand, *Parallel* [8] baseline has a high throughput. However, with this approach, the latency of the slowest component (the *Little* CPU cluster) determines the worst-case latency of the inference. Consequently, the *Parallel* baseline suffers from high latency.

Figure 7 shows the *Pipe-All* approach, on average, provides 5.42% and 75.88% higher throughput than the *Parallel* and *Peak* baseline, respectively. However, compared to the *Peak* baseline, high throughput from *Pipe-All* comes at the cost of only a 55.59% increase in latency versus a 419.87% increase in latency with the *Parallel* baseline. We can attribute the results to the ability of the pipelined design within *Pipe-All* to distribute processing between components to balance out the latency of individual pipeline stages.

7 Conclusion

ML inferencing on embedded platforms using HMPSoCs is now ubiquitous. However, to achieve a high enough throughput, we must engage all components synergistically in inferencing. High throughput should also not come at the cost of high latency. Thererfore, in this work, we introduced a new open-source pipeline design called *Pipe-All* for multi-component CNN inference on HMPSoC with *ARM big.Little* asymmetric multi-core processors and *Mali* GPUs. *Pipe-All* has a three-stage pipeline with the *big* CPU Cluster, *Small* CPU Cluster, and GPU as its three stages. The on-board evaluations show *Pipe-All*, on average, can provide 75.88% higher inference throughput than single peak-component inference with only a 55.59% increase in latency. In the future, one must extend the CPU-GPU pipeline to incorporate ML accelerators in HMPSoCs such as Neural Processing Units (NPU) into the design for even higher throughput.

References

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

[2] Eun Jin Jeong, Jangryul Kim, Samnieng Tan, Jaeseong Lee, and Soonhoi Ha. Deep learning inference parallelization on heterogeneous processors with tensorrt. *IEEE Embedded Systems Letters*, 2021.

[3] Bogil Kim, Sungjae Lee, Amit Ranjan Trivedi, and William J Song. Energy-efficient acceleration of deep neural networks on realtime-constrained embedded edge devices. *IEEE Access*, 8:216259–216270, 2020.

[4] 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.

[5] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. Integrated cpu-gpu power management for 3d mobile games. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.

[6] Pirah Noor Soomro, Mustafa Abduljabbar, Jeronimo Castrillon, and Miquel Pericàs. An online guided tuning approach to run cnn pipelines on edge devices. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, pages 45–53, 2021.

[7] 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.

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

[9] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing 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.

[10] Hsin-I Wu, Da-Yi Guo, Hsu-Hsun Chin, and Ren-Song Tsay. A pipeline-based scheduler for optimizing latency of convolution neural network inference over heterogeneous multicore systems. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 46–49. IEEE, 2020.