Pipelined CNN Inference on Heterogeneous Multi-Processor System-on-Chip

Ehsan Aghapour, Yujie Zhang, Anuj Pathania, and Tulika Mitra

Abstract Convolutional Neural Networks (CNNs) based inference is a quintessential component in mobile machine learning applications. Privacy and real-time response requirements require applications to perform inference on the mobile (edge) devices themselves. Heterogeneous Multi-Processor System-on-Chips (HMPSoCs) within the edge devices enable high-throughput, low-latency edge inference. An HMPSoC contains several processing cores, each capable of independently performing CNN inference. However, to meet stringent performance requirements, an application must simultaneously involve all core types in inferencing. A software-based CNN inference pipeline design allows for synergistic engagement of all the cores in an HMPSoC for a high-throughput and low-latency CNN inference. In this chapter, we present two different CNN inference pipeline designs. The first design creates a pipeline between two different types of CPU cores. The second design extends the pipeline from CPU to GPU. We also provide a future perspective and research directions on the subject.

Key words: Asymmetric Multi-Cores, Embedded GPUs, Neural Processing Units (NPUs), Edge Computing, Machine Learning, Artificial Intelligence

Ehsan Aghapour
University of Amsterdam, Science Park 904, 1098 XH, Amsterdam, Netherlands e-mail: e.aghapour@uva.nl

Yujie Zhang
National University of Singapore, Computing 1, 13 Computing Drive, Singapore, 117417 e-mail: zyujie@comp.nus.edu.sg

Anuj Pathania
University of Amsterdam, Science Park 904, 1098 XH, Amsterdam, e-mail: a.pathania@uva.nl

Tulika Mitra
National University of Singapore, Computing 1, 13 Computing Drive, Singapore, 117417 e-mail: tulika@comp.nus.edu.sg

1 Introduction

Pattern recognition plays an important role in several edge applications such as image and speech processing. Application developers commonly use Deep Neural Networks such as Convolution Neural Networks (CNNs) in their applications [1]. However, CNNs are resource-intensive solutions that require significant computational power. Heterogeneous Multi-Processor System on Chips (HMPSoCs) [21] powering mobile devices can potentially meet this challenge. However, the applications must engage all the compute elements within such systems to provide an adequate level of performance.

HMPSoCs contain several heterogeneous processing cores with different power-performance characteristics, such as general-purpose CPU cores of varying complexity, GPUs, DSPs, non-programmable accelerators, and reconfigurable computing elements. The heterogeneity enables compute-intensive application kernels to execute on the appropriate processing core best suited for that kernel [21]. This execution results in substantially improved performance and energy efficiency [2]. Executing the CNN inference on one of these processing core types is straightforward. However, such execution may not meet the throughput or latency requirements except for Neural Processing Units (NPU). However, many HMPSoCs do not include NPUs. Therefore, we need to harness the power of the HMPSoC by engaging all the compute elements during inference to achieve the power-performance goals. It is non-trivial to execute CNNs on multiple processing core types in parallel, all performing a part of the inference.

Direct execution of CNN workload (each layer) on different types of processing cores simultaneously results in a reduction of performance (throughput) instead of improvement because of the additional communication overhead involved [4] in such execution. We demonstrate this with CNN inference execution on *Amlogic A311D HMPSoC* within *Khadas Vim 3* embedded platform [3], which contains two different types of CPU cores, *Big* and *Small*. Figure 1 shows how throughput changes when we move from engaging one CPU core type to engaging simultaneously two different CPU core types. We begin by executing the inference on one *Big* core and report the throughput. We then perform multi-threaded inference by creating two threads and then pinning them on two different *Big* cores. The performance (throughput) for the inference continues to increase as long as we include more *Big* CPU cores for inferencing. However, the performance degrades once we create more threads and try to include both *Big* and *Small* CPU cores in inferencing simultaneously using Heterogeneous Multi-Processing (HMP). Improving neural network inference performance by directly using multi-threaded HMP within each layer remains an open research problem.

The pipelined designs presented in this chapter highlight the software challenges and opportunities in harnessing the computation power offered by diverse components within a heterogeneous multiprocessor systems-on-chip device in mobile and edge computing for deep neural network inference. The clear advantage of co-execution is significantly improved inference throughput that can be as high as 2x compared to single component execution. But there are many challenges in reaching

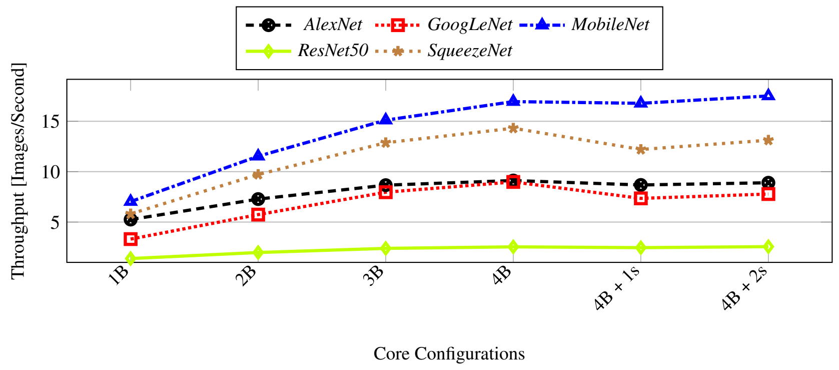

Fig. 1 Throughput of different CNNs with a different number of heterogeneous cores (B: *Big* core, s: *Small* core) with direct parallelization of CNN workloads (layers).

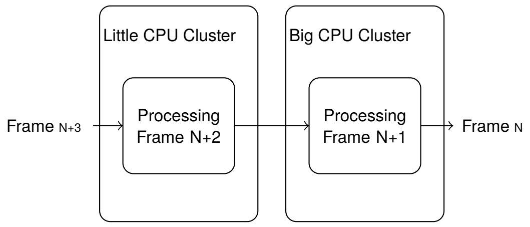

Fig. 2 An abstract block diagram showing high-throughput (low-latency) pipelined inferencing of an input data stream on HMPSoC using a two-component two-stage pipeline. The number of stages can be more than the number of components if there are multiple cores within a component.

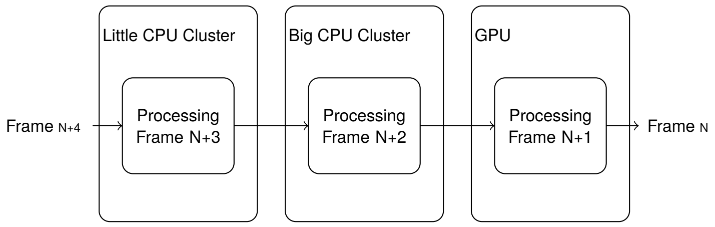

Fig. 3 An abstract block diagram showing high-throughput (low-latency) pipelined inferencing of an input data stream on HMPSoC using a three-component three-stage pipeline. The number of stages can be more than the number of components if there are multiple cores within a component.

this high throughput. A simple approach is to deploy all the on-chip compute resources by allocating each incoming input (e.g., an image) to the available compute component for inference. This approach can provide close to superior throughput. Still, it has several drawbacks: much longer latency (defined by the slowest on-chip component) and high jitter with uneven inference time per input depending on the compute resource allocated to it.

Parallel execution of different CNNs layers across different processor core types through layer-level software pipelining presents a way out of this problem. In this approach, we do not parallelize within a layer and hence do not incur the high cost of communication overhead. Layer-level software pipelining entails far less communication overhead than direct parallelization of layers of a CNN across the different core types. We also do not have the bottleneck of the slowest compute component in terms of high latency or jitter. Nonetheless, it still comes at some latency cost. We can balance the load across different stages of the software pipeline by assigning a different number of layers to each pipeline stage. The balancing maximizes throughput while ensuring the minimum penalty to the latency.

In this chapter, we describe two different pipelined designs for CNN inference. The first design creates a software pipeline between two different processor core types within an asymmetric multi-core CPU (big and little) now commonplace in HMPSoCs to create a two-component pipeline, as shown in Figure 2. This design, however, cannot be extended beyond the asymmetric multi-core. Therefore, we then describe a completely different design that creates a software pipeline between the two asymmetric multi-core CPU core types and a GPU to create a three-component pipeline, as shown in Figure 3.

The challenge with these software pipelines is the software and systems design to support inference across different core types, such as ARM big.LITTLE CPU and GPU. The current commercial CNN inference frameworks and software libraries are designed and highly optimized for single-component inference through multi-threading. Extending them in a straightforward fashion to execute different pipeline stages on different components leads to high overhead due to the replication of neural network weights in each thread. For ARM big.LITTLE CPU, we take advantage of the fact that the big and little cores share the same instruction-set architecture and hence can execute the same software binary. Therefore, instead of assigning a different thread for each pipeline stage, we migrate the thread from one pipeline stage to another, carrying all the data. This leads to less overhead and more efficient implementation. We also share all the weights and biases across different threads working on different input data frames. For co-execution across CPU and GPU, the software binaries differ due to different instruction-set architectures and require a different strategy. In this case, we partition the compute graph corresponding to the inference into multiple components, one corresponding to each pipeline stage. The data transfer between two pipeline stages that belong to two different compute elements is carefully orchestrated with an explicit buffer and transformation of the data to satisfy the requirement of the corresponding compute elements.

We design, implement and experimentally evaluate the different CNN inference co-execution approaches on a contemporary heterogeneous multiprocessor system-on-chip with asymmetric multi-core, GPU, and NPU. This allows us to compare the different design choices on a single hardware platform and analyze their relative benefits and shortcomings. To the best of our knowledge, this is the first time that these different techniques have been studied on a common heterogeneous multiprocessor systems-on-chip architecture.

Finally, we describe the challenges in extending the design to create a multi-component pipeline by further including the NPU.

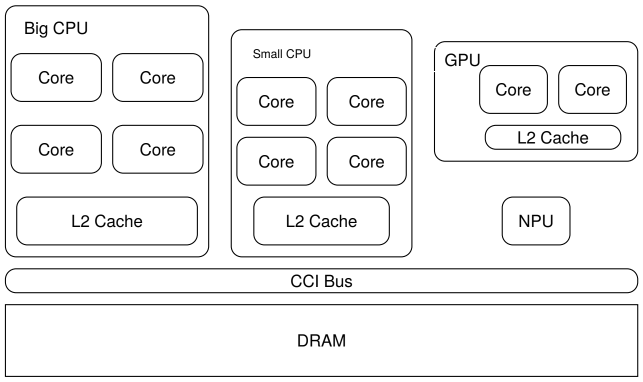

Fig. 4 An abstract block diagram of heterogeneous multi-processor system-on-chip (HMSoC) with an asymmetric multi-core CPU, a multi-core GPU, and an NPU along with other components. Such HMPSoCs are now ubiquitous in edge (mobile) devices.

2 Background

2.1 Heterogeneous Multi-Processor System on Chips

Figure 4 shows an abstract block diagram of a heterogeneous multi-processor system-on-chip (HMPSoC) now commonplace in edge (mobile) devices. HMPSoC contains several different processor core types that communicate among themselves and with the Dynamic Random Access Memory (DRAM) based main memory using a Cache Coherent Interconnect (CCI) bus. It contains an asymmetric multi-core CPU [10, 11] which contains a *Big* and a *Small* CPU cluster. The *Big* CPU cluster is a cluster of high-performance, high-power cores. The *Small* CPU cluster is a cluster of low-performance, low-power cores. The power-performance envelope of the cores in *Big* CPU and *Small* CPU can sometimes overlap. All the CPU cores have their private L1 data and instruction cache. The cores within a CPU cluster share an L2 cache. The L2 cache in the *Big* CPU is bigger than the L2 cache in the *Small* CPU. The cores within a cluster communicate using an intra-cluster bus called Snoop Control Unit (SCU). One can also expect various many-cores to replace multi-cores in HMPSoCs [43]. Finally, it is also possible to design reconfigurable multi-core architectures that combine multiple small cores into a big core at runtime, depending on the workload [35–37].

An HMPSoC also contains a multi-core GPU. GPUs are essential for mobile games [8, 9]. However, they can also run other generic parallel processing workloads such as those commonly found in CNNs using frameworks like *OpenCL* and *CUDA* [12]. GPU cores also share a common L2 cache. GPU cores are generally more power-efficient than CPU cores. However, they cannot work in isolation and must rely on the necessary support from the CPU cores.

The HMPSoCs also increasingly include hardware accelerators called Neural Processing Units (NPUs) for executing deep neural network workloads. NPUs are generally Application-Specific Integrated Circuits (ASICs) with a vendor-specific design. These ASIC designs make them significantly more power-efficient than even the GPUs. However, the efficiency comes at the cost of flexibility, wherein CNNs must be compatible with the NPU design to work. Re-programmable NPUs based on reconfigurable accelerators such as Coarse-Grained Reconfigurable Arrays (CGRAs) architectures [13–19] and Scratch-Pad Memories (SPM) [20] may address this problem in the future. Accelerators also require support from the CPU to work with CNNs and therefore cannot operate independently.

2.2 Convolution Neural Networks

Convolution Neural Networks are neural networks that take inspiration from nature, particularly the process of stimuli response in the visual cortex within the eyes of living beings. Similar to the eyes and ears in the real world, the primary purpose of CNNs is to perform some form of pattern recognition. Therefore, CNNs find use in several computational problems that require pattern recognition, such as image classification, speech recognition, and natural language processing.

CNNs are regularized versions of common machine learning constructs called Multi-Layer Perceptrons (MLP). They primarily employ a mathematical construct called convolution that gives them their namesake. Convolution is computationally similar to general-purpose matrix multiplication and is easy to parallelize.

A CNN typically contains three different layer sets – an input layer, multiple hidden layers, and an output layer. Figure 5 shows an architecture of a popular image classification CNN called *AlexNet* as an example. Input layers take in the input that requires recognition. Hidden layers perform the convolutions on the input given to them. Input and output are not human-comprehensible as an activation function (commonly ReLU), and final convolution masks them. Convolution layers within the hidden layers primarily compute the dot product (generally Frobenius inner product) of convolution kernel(s) with the layer's input. The convolution kernel matrix slides along the input matrix to generate a feature map. The feature map is the input for the next layer. Hidden layers also involve several other layer types such as pooling, fully-connected, and normalization layers working with the output from convolution layers. Pooling layers reduce data dimensions by combining the output of clusters of neurons. Fully connected layers connect all its neurons with all other neurons in the next layer, similar to layers in traditional MLPs. In CNNs, neurons receive input data from only a part of previous layers. This part defines the receptive field of the neuron. A neuron then computes its output by applying its function to the input it receives from its receptive field in the previous layer. Besides the input data, the neuron function also takes in a set of static input vectors called weights and biases for computation.

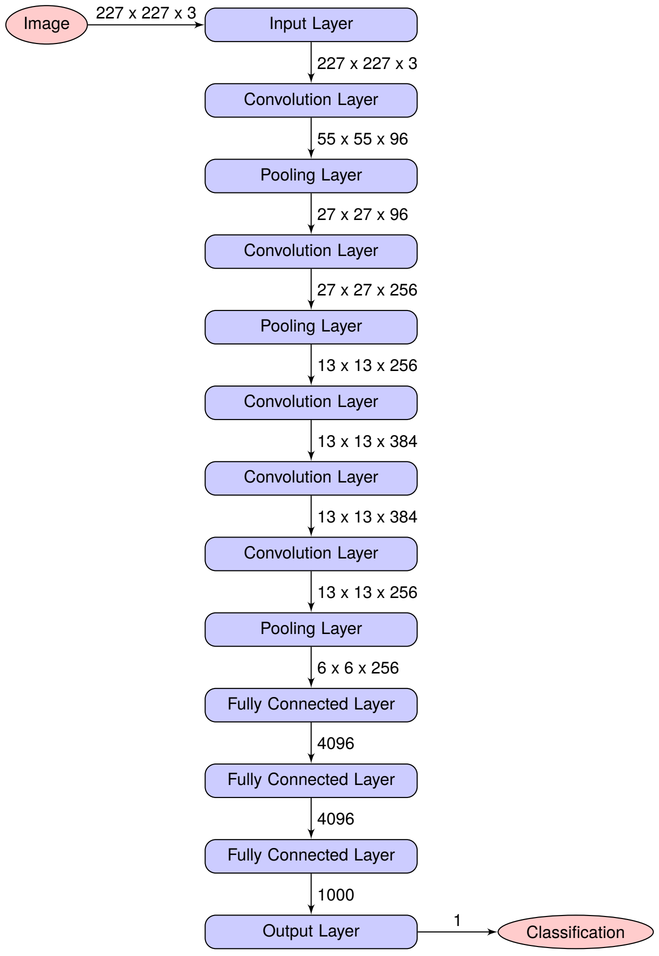

Fig. 5 An abstract block diagram of CNN architecture for a popular image classification CNN called *AlexNet*.

Deployment of CNNs is a two-step process. In the first steps, we train a CNN primarily on a set of labeled input data wherein we decide upon the best weights and biases for the neurons in the neural network. In the second step, we place the trained neural network on the device requiring CNN inferencing. Training CNNs is a computationally expensive process that requires a significant amount of processing and storage resources. This level of resources is generally beyond the means of edge devices. Therefore, the training process happens primarily in cloud data centers or other high-performance computing machines. Edge devices are computationally well suited to perform inference. In fact, due to privacy and performance (latency) reasons, users prefer the inference to happen on the edge device itself. Edge inference also brings additional benefits, such as operating offline.

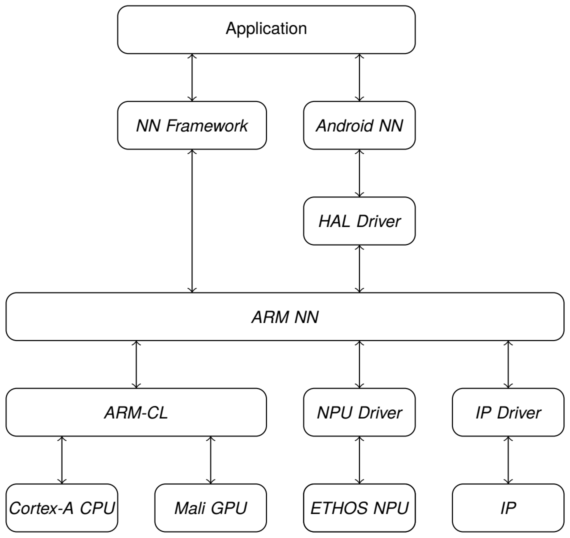

Fig. 6 An abstract block diagram showing the placement of *ARM-CL* in the entire CNN inference ecosystem for *ARM*-based HMPSoCs.

2.3 ARM Compute Library (ARM-CL)

ARM Compute Library (*ARM-CL*) is a library of highly optimized low-level machine learning functions for ARM *Cortex-A* CPU cores (*ARM v7* and *v8* ISA) and *Mali* GPU cores (*Midgard* and *Bifrost* architecture). It uses *Neon* (or *OpenCL*) for accelerating CNN inference on the CPU (or GPU) cores. *ARM-CL* is written primarily in *C++ 14*. *ARM-CL* comes as a part of the open-source *ARM NN* Software Development Kit (SDK), as shown in Figure 6.

ARM NN is an inference engine that bridges the gap between existing Neural Network (NN) frameworks and the underlying IPs. It supports several popular NN frameworks such as *TfLite*, *ONNX*, *PyTorch*, and *Caffe*. ARM NN also supports *ARM Ethos* NPUs out-of-the-box and keeps the option of supporting third-party NPUs, and other IPs open. *ARM NN* interfaces with *Google Android NN* using the Hardware Abstraction Layer (HAL) driver to target the underlying *ARM-based* HMPSoCs. *ARM NN* (or *ARM-CL*) has the highest known inference performance for *ARM-based* HMPSoCs against other generic vendor-agnostic equivalents [5].

3 Related Work

Heterogeneous computing emerged as a solution to balance computation energy efficiency and performance in the dark silicon era [21]. The asymmetric multi-core architecture includes a mixture of cores with different power-performance characterizations, such as general-purpose CPU, GPU, and special-purpose accelerators. An evaluation of the CNN inference capabilities of these different cores appears in [5]. Application developers can utilize these capabilities to determine CNN workload partition on processing cores. However, the synergistic orchestration of various on-chip computing resources needs significant software support [22, 23] to improve performance and energy efficiency. There are different ways of partitioning CNN workload on multiple processing cores to exploit task-level (pipeline) or data-level parallelism during compile time or run time.

Existing state-of-the-art machine learning inference libraries, like *ARM-CL* [24], *Tengine* [25], and *NCNN* [26], partition the CNN models at the kernel level within a layer across the cores of the same type on HMPSoC. However, this strategy is unsuitable for heterogeneous component co-execution because of the high inter-component communication overhead. The authors of [34] propose a pipeline-based scheduler to reduce the delay due to feature map data movement between *ARM big.LITTLE* CPU processing cores. The authors iteratively bi-partition the model-based Directed Acyclic Graph (DAG) to multiple parts with nearly the same computation time to match the latencies of the pipeline stages. This simple implementation achieves 73% greater CNN inference throughput than the traditional approach utilizing only data-level parallelism. Similarly, the authors of [38] present a hardware management technique, *NeuroPipe*, to pipeline consecutive neural network layers on the host and accelerators to maximize task-level parallelism. *NeuroPipe* obtains greater hardware utilization and achieves energy-efficient acceleration than the conventional host-device execution mode.

On CPUs-GPUs MPSoCs, the authors of [28] formulate an execution pipeline by distributing CNN layers on CPUs to leverage task-level parallelism among the layers. Then to aid the CPUs, the GPUs are used to accelerate part of the computation within layers to improve the inference speed. In this case, the authors utilize both the task-level and data-level parallelism of CNN models on HMPSoCs. However, the performance degradation due to the communication overhead between CPUs and GPUs for the same layer computation is unavoidable. The authors of [29] extend this work and utilize Genetic Algorithms to find the Pareto-optimal mapping of CNN workloads with appropriate voltage and frequency scaling configuration. The mapping reduces the power consumption further. To maximize the throughput, authors of [30] pipeline the inference network at layer level using GPU and NPUs and utilize multi-stream to improve parallelism within these accelerators.

The above works leverage only profiling data during compile time to schedule the CNN implementations on heterogeneous processing cores. Some research also adopts runtime strategies to help determine workload mapping configuration on HMPSoCs, such as *work-stealing* [39] and online tuning [32]. The work in [39] exploits a multi-threaded pipeline at layer granularity to simultaneously utilize all on-chip resources of embedded FPGA-based HMPSoCs and relies heavily on high-level synthesis [40, 41]. The authors employ a work-stealing scheduler to balance the workload of different processing units at run time. The authors of [32] propose an online tuning algorithm *Pipe-search* that generates near-optimal pipeline configuration on HMPSoCs, based on compile-time hints and real-time performance

Table 1 Specifications summary for the experimental setup used in this work.

Component	Details
Board	*Khadas VIM3 Pro*
SoC	*Amlogic A311D* @ 12 nm
CPU	*ARM big.Little*
big CPU	Quad-Core *ARM Cortex-A73* @ 2.2 GHz
Little CPU	Dual-Core *ARM Cortex-A53* @ 1.8 GHz
GPU	*ARM Mali G52* @ 0.8 GHz
NPU	*VeriSilicon* IP @ 5 TOPS
Memory	4 GB LPDDR4
Storage	32 GB EMMC 5.1
OS	*Android Pie* v9.0
Kernel	*Linux* v4.9
CNN Framework	*ARM-CL* v21.02

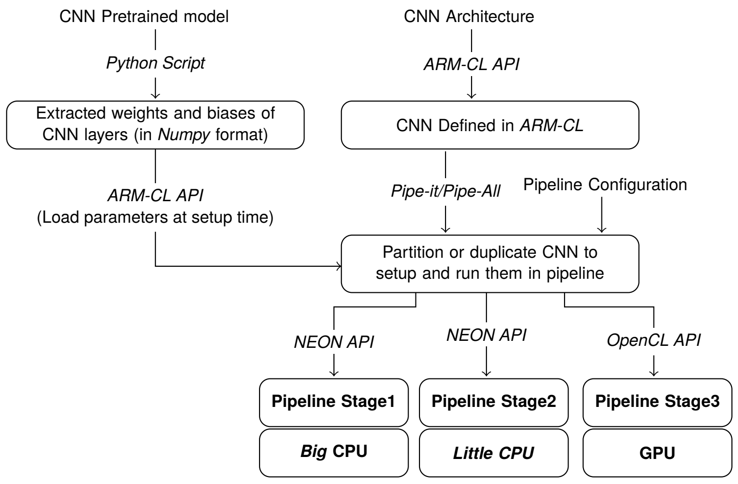

Fig. 7 The overview of tool flow for pipeline CNN inference using *Pipe-It* and *Pipe-All*.

measurements. With the help of task moldability supported by XiTAO runtime [27], *Pipe-search* keeps tuning adaptive pipelines to minimize the execution time of the slowest stage.

4 Experimental Setup

We use the *Khadas VIM3 PRO* board for our experiments. Table 1 summarizes the relevant details of the board. *Khadas VIM3 PRO* board contains an *Amlogic A311D* HMPSoC fabricated on a 12 nm technology node. It has an *ARM big.Little*; the *big* CPU cluster contains quad-core *ARM Cortex-A73* high-performance, high-power general-purpose processing cores with a peak frequency of 2.2 GHz, and the *Lit-*

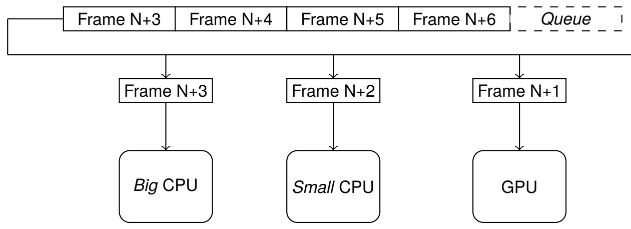

Fig. 8 An abstract block diagram showing high-throughput high-latency parallel inferencing of input stream on HMPSoC by simultaneous engagement of multiple components.

tle CPU cluster contains dual-core *ARM Cortex-A53* low-performance, low-power general-purpose processing cores with a peak frequency of 1.8 GHz. It also contains a dual-core *ARM Mali G52 MP4 GPU* of *BiFrost* design with four execution units running at their peak frequency of 800 MHz. There is also a *VeriSilicon* NPU capable of INT8 quantized CNN inference at the rate of 5 Tera Operations per Second (TOPS) (up to 1536 Multiply-Accumulate (MAC) operations). A 4 GB Low-Power Double Data Rate 4 (LPDDR4) memory acts as the main memory. A 32 GB embedded MultiMediaCard (eMMC) 5.1 drive provides the storage. The board runs *Android Pie* v9.0 OS with *Linux* kernel v4.9. It runs *ARM-CL* v21.02 to support on-chip CNN inference. Figure 7 shows the tool flow for running pipeline inference using *ARM-CL*. CNN architecture is defined (layer by layer) in ARM-CL using the ARM-CL APIs. *Pipe-All* and *Pipe-it* are integrated into ARM-CL and partition or create multiple instances of the defined CNN, respectively. Then they create multiple Graphs/SubGraphs instead of one Graph equivalent to the whole CNN corresponding to the different pipeline stages. Next, they set up these Graphs/SubGraphs and run them simultaneously in a pipelined fashion. After the creation of each graph/subgraph, ARM-CL loads weights and biases of each layer (extracted from the pre-trained CNN model) at setup time. For the CPU cores, NEON API is used to take advantage of the Arm Neon SIMD architecture extension, while for the GPU cores, OpenCL API is used.

5 Non-Pipelined Parallel Inference

We first present a non-pipelined, high-throughput inferencing approach using a very basic design wherein all the components in the HMPSoC perform parallel CNN inference independently. However, this approach suffers from high latency. Figure 8 shows such a design using an abstract block diagram. The authors of [5] were the first to propose and evaluate such a design.

The design consists of a shared arrival queue shared across HMPSoC components. All frames for inference get placed in this queue on arrival. All components of the HMPSoC – *Big* CPU cluster, *Small* CPU cluster, GPU, and NPU – pull a frame for inference from the shared queue. This approach results in minimal inter-component

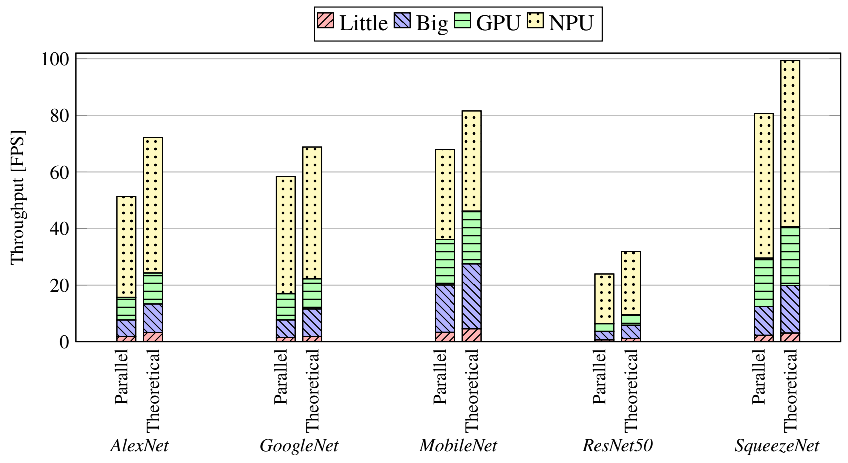

Fig. 9 Throughput of different CNNs with non-pipelined parallel CNN inference on all components simultaneously (Parallel) compared to theoretical addition of components' inference throughput (Theoretical).

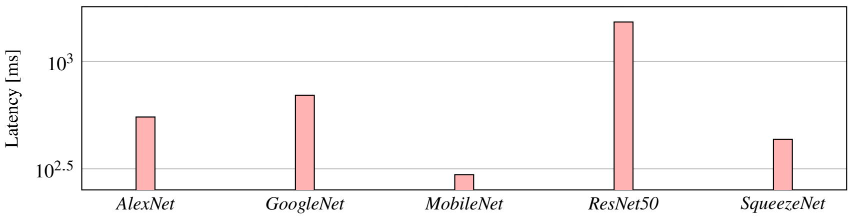

Fig. 10 Latencies of different CNNs with non-pipelined parallel CNN inference on all components simultaneously. The *Little* CPU cluster being the slowest component determines the latency.

communication. The performance is almost equivalent to the sum of all components inferencing in parallel in isolation. Figure 9 shows the throughput with a non-pipeline parallel inference design and the contribution of the individual components. One can expect slight performance degradation from the theoretical sum because of shared resources such as CCI bus and DRAM controllers. The figure shows that parallel inference throughput, on average, is 21% less than the theoretical sum of all components inference throughput.

The downside of this design is that the latency of the slowest inferencing component on the HMPSoC becomes the latency of the non-pipelined parallel inference design. The *Small* CPU cluster is the slowest inference component in the HMPSoC. Therefore, the latency of the *Small* CPU cluster defines the latency of the non-pipelined parallel inference design. Figure 10 shows the latency of the non-pipelined parallel inference design on our setup. The non-pipelined parallel inference design cannot engage an HMPSoC component in its design if the latency requirements for inference are less than the latency of that component. It is also not suitable for appli-

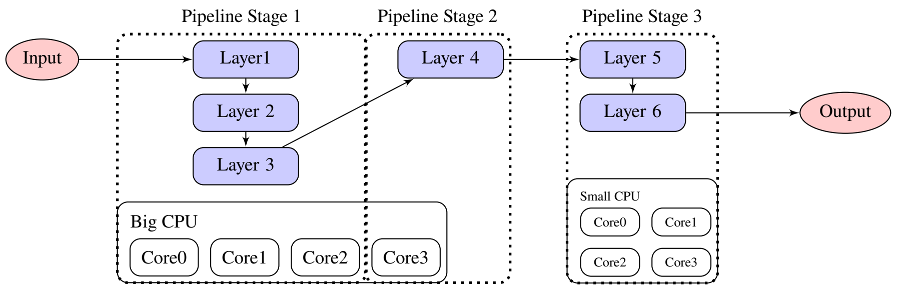

Fig. 11 An abstract diagram showing how the two-component pipeline of *Pipe-it* works in a three-stage configuration on an asymmetric multi-core.

cations that are jitter sensitive. Jitter-sensitive applications cannot tolerate different frames having inference with significantly different latencies.

Pipelined inference designs address shortcomings of the non-pipelined parallel inference design but are significantly more involved in implementation. They bring the same benefits as the non-pipelined parallel inference design to the throughput but at a much lower latency.

6 Pipelined CNN Inference on Asymmetric Multi-Core

We first present the pipelined design of CNN inference on asymmetric multi-cores called *Pipe-it* that we originally proposed in [4]. *Pipe-it* uses both *Small* and *Big* CPU clusters in an asymmetric multi-core for inferencing using a software pipeline. The design of *Pipe-it* is implemented within the *ARM-CL* framework. Figure 1 shows processing a given layer directly on *Big* and *Small* CPU clusters simultaneously is detrimental to the CNN inference performance (both latency and throughput). *Pipe-it*, therefore, chooses to split CNN inferencing processing at the granularity of layers.

Figure 11 shows the design behind *Pipe-it* using an abstract block diagram. *Pipe-it* splits the available cores into different stages. Each stage is homogeneous in terms of the core types. *Pipe-it* assigns a series of consecutive layers to each stage. Each frame gets processed after passing through all the stages in a given order. *Pipe-it* process multiple frames in parallel to boost throughput. The processing of frames is synchronously time-shifted to be not in the same stage simultaneously.

It primarily builds upon CPU thread migrations for implementing the pipeline. Even though *Big* and *Small* CPU clusters in an asymmetric multi-core CPU have significantly different power-performance envelopes, they share the same Instruction Set Architecture (ISA). This commonality allows the threads from the same binary to execute on cores of both *Big* and *Small* CPU clusters. One can also freely migrate the threads between the *Big* and *Small* CPU cluster mid-execution with minimal penalty at run-time [10]. *ARM-CL*, by default, allows the workload for CNN inference to be parallelized using multi-threading. We can define the number of threads to split the CNN inference workload for a given frame. *Pipe-it* builds upon the in-built multi-threading constructs of *ARM-CL* for its implementation.

ARM-CL, by default, allows inference of only one frame at a given time. One can run multiple binaries in *ARM-CL* in parallel for inferencing various frames. However, the system duplicates the common denominators (weight and biases) across binaries in such execution. This duplication leads to the wastage of precious memory resources. Therefore, *Pipe-it* extends ARM-CL to perform inference of multiple frames in parallel in code. *Pipe-it* then performs inference of multiple frames using the same binary wherein the parallel inference processes share the read-only weights and biases. Input and output data structures are still separate for each inference.

ARM-CL does not have a construct to identify when a particular layer of CNN inference starts or finishes execution. Therefore, *Pipe-it* adds hooks to the *ARM-CL*, which get triggered when each layer in a CNN starts execution. It identifies when *ARM-CL* threads related to CNN inference of a given frame start processing a frame. It then migrates (pins) the thread to *Big* and *Small* CPU clusters, depending on which core type it wants to execute the layer in the pipeline. Since the workload for a CNN inference is independent of input, *Pipe-All* calculates these decisions statically at design time.

Two-component inference with *Pipe-it* does not necessarily mean a two-stage pipeline. *Pipe-it* allows cores in a CPU cluster to be sub-divided into multiple stages up to the granularity of an individual core. Therefore, a quad-core CPU cluster can have one pipeline stage of four cores or four pipeline stages of one core each or any other combination. *Pipe-it* uses a sophisticated Design Space Exploration (DSE) algorithm to figure out the number of stages in the CNN inference pipeline, their size (the number of cores), and the CNN layers they execute. *Pipe-it* stipulates that the layers assigned to a stage are consecutive. *Pipe-it* also proposes a regression-based performance model for layers of CNN on different CPU core types to guide the DSE. Figure 12 shows the overall design of pipe-it to find the most efficient pipeline configuration. First, it starts with one core for each stage of the pipeline, and the workload split module uses a heuristic to balance the workload among stages of the pipeline. Then it explores if merging two stages could improve the performance; if yes, it merges them and initiates the workload splitting module; if not, the current configuration is returned as the most efficient one for the pipeline. Readers can refer to [4] for more details on the DSE that maximizes CNN inference throughput with minimal penalty to latency.

Figure 13 shows the throughput of *Pipe-it* with different CNNs. It compares the throughput of *Pipe-it* to the CNN inference of the best single-component limited to the asymmetric multi-core CPU and parallel execution on multi-core CPUs. The figure shows that *Pipe-It* can achieve 16% higher throughput than the best single component on average and is almost the same as parallel execution of multi-core CPUs for entirely different frames. The small CPU cluster in the SoC used for our experiments has two cores compared to the Big CPU cluster, which has four cores. Hence, adding the small CPU cluster increases the throughput by just 16% compared to inference on the Big CPU cluster. We note that in the original work [4],

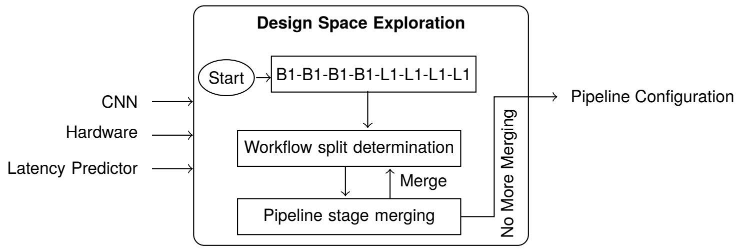

Fig. 12 *Pipe-it* design flow for finding the most efficient pipeline configuration and workload splitting.

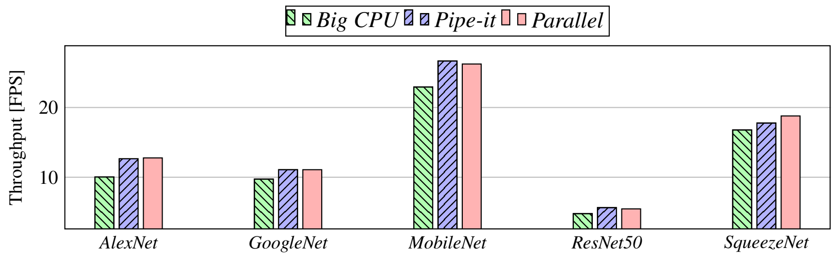

Fig. 13 Throughput of different CNNs with pipelined inference using *Pipe-it* compared to best single component CNN inference and parallel CNN inference on asymmetric multi-core.

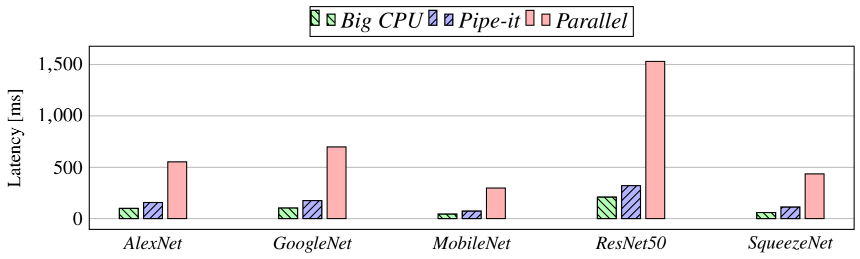

Fig. 14 Latency of different CNNs with pipelined inference using *Pipe-it* compared to best single component CNN inference and parallel CNN inference on asymmetric multi-core.

we conducted the experiments on an SoC with four big cores and four small cores, and hence Pipe-It increased inference throughput by 39% compared to the best single-component throughput on average. Figure 14 shows the latency of *Pipe-It* with different CNNs and compares it with the latency of the single best component inference and the parallel inference on components. The figure shows *Pipe-It* has 67% higher latency than the best single component on average, while it has 62% lower latency than parallel inference on components on average. This is because the slowest component, the small CPU cluster, determines the latency of parallel inference.

In other words, *Pipe-it* strikes a balance between throughput and latency. Its throughput is closer to the non-pipelined parallel implementation, which is almost the maximum throughput achievable by utilizing both the CPU clusters. On the other hand, the latency is closer to the best CPU cluster's latency than parallel inference on components.

Open-Source: The source code for *Pipe-it* is publicly available at the following link - *https://github.com/Elliezza/CLCode/blob/master/multi_split*

7 Pipelined CNN Inference on Asymmetric Multi-Core and GPU

The next natural step is to extend the CNN inference software pipeline from asymmetric multi-core to the embedded GPU in HMPSoCs. However, the design of *Pipe-it* relies on migrating CPU threads between *Small* and *Big* CPU clusters of an asymmetric multi-core to achieve the software pipeline. It is infeasible to migrate CPU threads from a CPU to GPU. Therefore, it is impossible to extend the *Pipe-it* design to create a CPU-GPU pipeline.

We now present a new design called *Pipe-All* to combine *Small* and *Big* CPU clusters with GPU in a unified three-component pipeline, originally proposed in [6]. The design of *Pipe-All* differs significantly from *Pipe-it* even though they share the same end goal. *Pipe-All* also works on top of *ARM-CL* and uses the extended version of *ARM-CL* that supports CNN inference of multiple frames in parallel.

ARM-CL converts a given neural network architecture into an *ARM-CL* graph. The graph consists of three main node types – Input Node, Main Node, and Output Node. Input Node corresponds to the input layer of the CNN and processes the input image. Main Node corresponds to processing layers of the CNN and clubs together several layers such as convolution, pooling, and fully-connected layers into one. Main Node takes in weights and biases as input. Output Node corresponds to the output layer of the CNN and provides a classification for the input image. Figure 15 shows the *ARM-CL* graph corresponding to the *AlexNet* CNN architecture in Figure 5.

Pipe-All introduces the concept of sub-graph into *ARM-CL*. Sub-graph contains two new types of nodes called Transfer and Receiver Node besides the Input, Main, and Output Nodes. A sub-graph is a part of the *ARM-CL* graph that connects with other sub-graphs using a synchronized data buffer. The Transfer Node transfers the data to the succeeding sub-graph, whereas the Receiver Node receives the data from the preceding sub-graph. Sub-graphs dividing the unified *ARM-CL* graph represent the stages of the software pipeline that execute parts of the graph in unison. Sub-graphs can execute either on the CPU or GPU, and the most efficient component for each sub-graph can be different [42]. *Pipe-All* invokes CPU and GPU implementation of the sub-graphs depending upon where they need to execute. *Pipe-All* provides the APIs for the users to define the number of sub-graphs and their size. APIs also allow for their placement on different components of the HMPSoC.

Figure 16 shows the *ARM-CL* graph divided into three sub-graphs that *Pipe-All* uses to create a three-stage CNN inference pipeline. *Pipe-All* can support an N-Stage pipeline by creating N sub-graphs. It uses internal *ARM-CL* multi-threading constructs to parallelize processing within a component when necessary. A Transfer

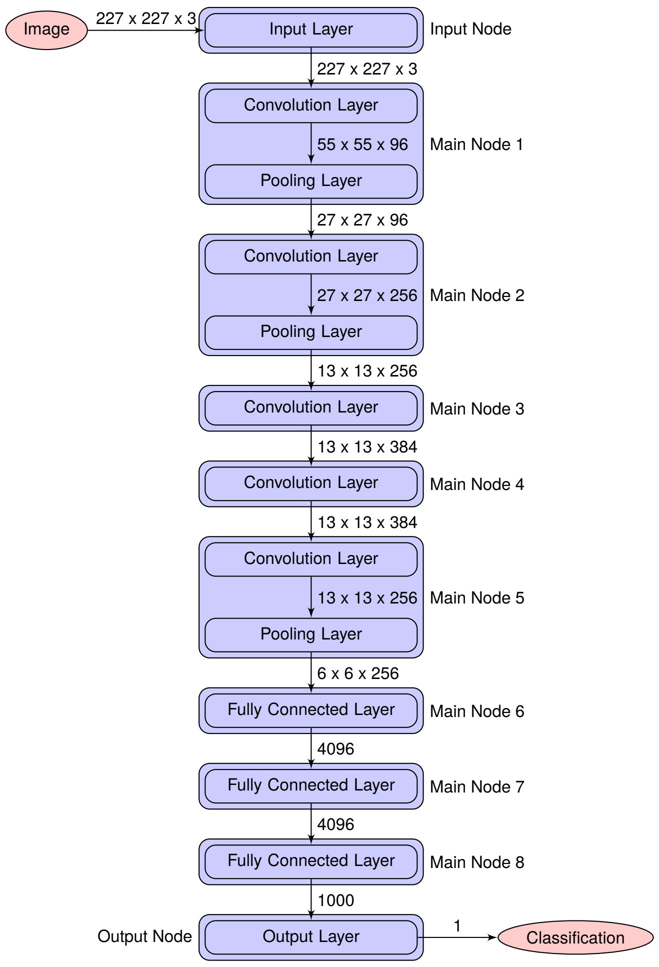

Fig. 15 An abstract block diagram of an *ARM-CL* graph for *AlexNet* CNN.

Node performs no data transformation when *Pipe-All* places two connecting sub-graphs on cores that use the same ISA. However, the Transfer Node makes the data compatible with the Receiver Node when *Pipe-All* places two connecting sub-graphs on cores that use different ISAs.

Figure 17 shows the flow designed to find the most efficient CNN partitioning between pipeline stages and mapping these partitions to processing elements, including GPU and CPU clusters. In *Pipe-All*, the pipeline configuration is limited to considering each component as one pipeline stage. The reason is that the increasing number of pipeline stages increases the inference's latency because each stage's processing capacity decreases, and the number(overhead) of transactions increases. *Pipe-All* uses grid search to find the most efficient configuration. For this purpose, it explores all possible partitioning of the workload for each mapping of the pipeline

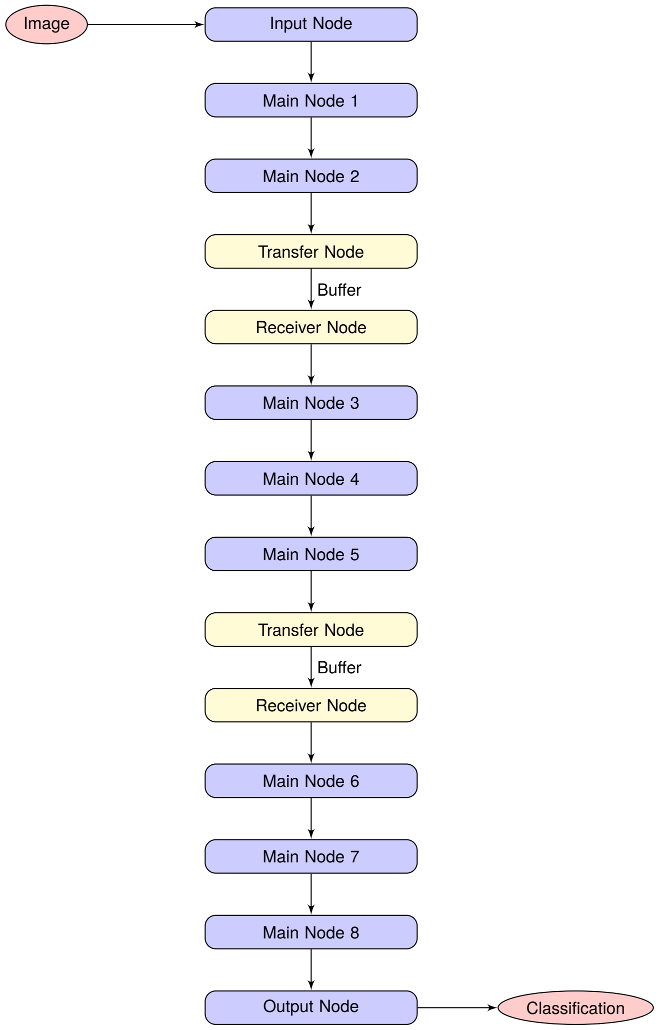

Fig. 16 An abstract block diagram of an *ARM-CL* graph for *AlexNet* CNN divided into three sub-graphs by *Pipe-All* to create a three-stage three-component pipeline.

stages to the processing elements. It evaluates each design by running inference with that configuration and measuring the performance on the real platform. Readers can refer to [6] for more details about the optimizations to maximize throughput in *Pipe-All*.

Figure 18 shows the throughput of *Pipe-All* with different CNNs. It puts the throughput of *Pipe-All* in context with the throughput of the best single-component inference limited to the asymmetric multi-core CPU and GPU. The figure shows that *Pipe-All* can achieve 73% higher throughput on average. Similarly, Figure 19 shows the latency of *Pipe-All* with different CNNs and puts it in the context with the best

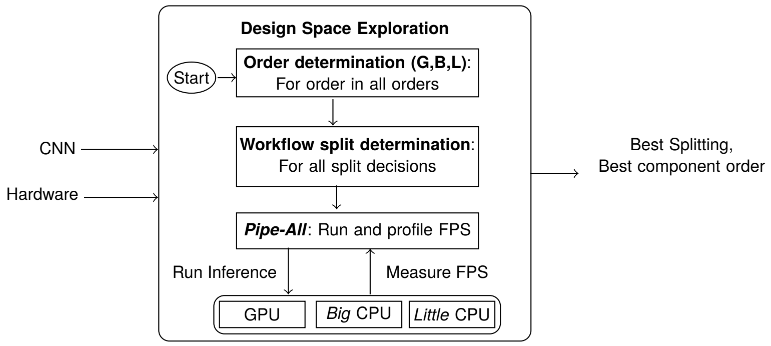
Design Space Exploration
Start
Order determination (G,B,L): For order in all orders
Workflow split determination: For all split decisions
Pipe-All: Run and profile FPS
Run Inference
Measure FPS
GPU
Big CPU
Little CPU
CNN
Hardware
Best Splitting, Best component order

Fig. 17 *Pipe-All* design flow for finding the pipeline components' most efficient workload splitting and order.

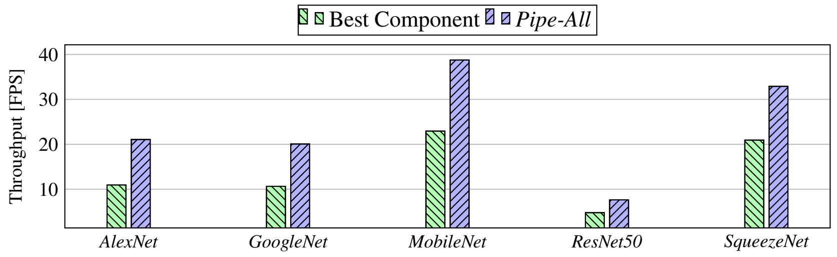

Fig. 18 Throughput of different CNNs with pipelined inference using *Pipe-All* compared to the best single-component CNN inference on asymmetric multi-core and GPU.

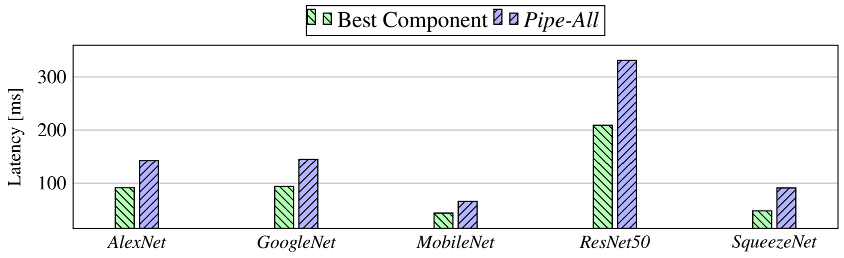

Fig. 19 Latency of different CNNs with pipelined inference using *Pipe-All* compared to best single-component CNN inference on asymmetric multi-core and GPU.

single-component inference. The figure shows that *Pipe-All* latency is 62% higher on average.

Open-Source: The source code for *Pipe-All* is publicly available at the following link - *https://github.com/Ehsan-aghapour/ARMCL-pipe-all*

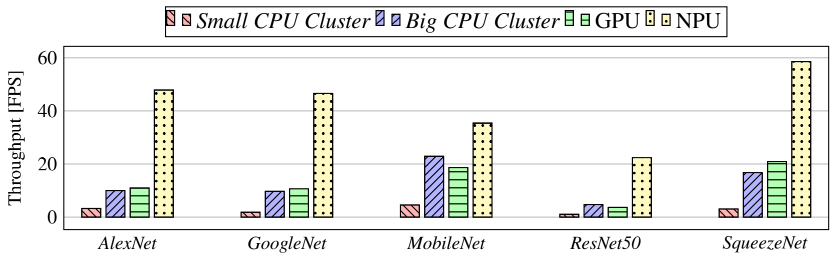

Fig. 20 Throughput of different HMPSoC components with different CNNs.

8 Pipelined CNN Inference and NPU

The final step to complete the software pipeline would be to include the NPU in the pipeline. To the best of our knowledge, no such pipeline design exists. However, *ARM-CL* does not support execution on any NPU directly. Unlike CPU and GPU, the use of NPU is also not standardized. NPU is available from multiple vendors. Each vendor requires using their proprietary framework for converting a CNN into a framework compatible with execution on their NPU. This multi-framework environment makes any CPU-GPU-NPU pipeline challenging to port to different HMPSoCs. Nevertheless, it remains technically feasible to create a CPU-GPU-NPU pipeline.

Figure 20 shows the throughput of NPU in the context of other HMPSoC components – the *Big* CPU cluster, the *Small* CPU cluster, and the GPU. It indicates that NPU has higher throughput than all other HMPSoC components. Still, there is scope to significantly improve the throughput for some CNNs, such as *MobileNet*, by using components other than NPU. Furthermore, NPU performs an INT8 quantized inference, whereas other components perform an unquantized inference. Therefore, the pipeline designs in Sections 6 and 7 are still helpful when performing unquantized inference for higher accuracy. Many low-end budget HMPSoCs also lack an NPU wherein the pipelined designs we describe in this chapter are valuable.

9 Conclusion and Future Outlook

We present two different designs for pipelined CNN inference on HMPSoCs. The first design, *Pipe-it*, creates a two-component software pipeline for CNN inference between the *Big* and *Small* CPU clusters of an asymmetric multi-core CPU of an HMPSoC. However, extending the *Pipe-it*'s design to include the GPU is not feasible. Therefore, we present another design called *Pipe-All*. *Pipe-All* creates a three-component software pipeline between CPU clusters of the asymmetric multi-core CPU and the GPU of an HMPSoC. Finally, we present the challenges and opportunities in extending the software pipeline to include the NPU.

The heterogeneous multiprocessor system-on-chip provides a rich playground for software innovations to embedded neural network inference. While the NPU provides superior latency, throughput, and power for inference, it has limitations regarding the supported models and computation kernels. In the rapidly changing landscape of neural network models, it is imperative to utilize the general-purpose compute resources such as asymmetric multi-core CPU and GPU prevalent across all embedded systems and adequate to execute any workload. Thus, software co-executing neural network inference across different on-chip compute components will become increasingly important. This chapter presented static approaches where the software pipeline and partitioning of the neural network model across various computing components are determined a priori. As the network models become more complex and dynamic, a hybrid approach where the statically designed pipeline can be adapted at runtime based on input will be an interesting research direction.

10 Acknowledgements

This research is partially supported by the National Research Foundation Singapore under its Competitive Research Program Award NRF-CRP23-2019-0003 and Singapore Ministry of Education Academic Research Fund T1 251RES1905.

References

1. LeCun, Y., Bengio, Y. and Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.
2. Prakash, A., Wang, S. and Mitra, T., 2020. Mobile application processors: Techniques for software power-performance optimization. IEEE Consumer Electronics Magazine, 9(4), pp.67-76.
3. Khadas VIM 3, https://www.khadas.com/vim3, 23 12 2011.
4. Wang, S., Ananthanarayanan, G., Zeng, Y., Goel, N., Pathania, A. and Mitra, T., 2019. High-throughput CNN inference on embedded ARM Big. LITTLE multicore processors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(10), pp.2254-2267.
5. Wang, S., Pathania, A. and Mitra, T., 2020. Neural network inference on mobile socs. IEEE Design & Test, 37(5), pp.50-57.
6. Aghapour, E., Pathania, A. and Ananthanarayanan, G., 2021. Integrated ARM big. Little-Mali Pipeline for High-Throughput CNN Inference. TechRxiv preprint.
7. Salamin, S., Rapp, M., Pathania, A., Maity, A., Henkel, J., Mitra, T. and Amrouch, H., 2020. Power-efficient heterogeneous many-core design with ncfet technology. IEEE Transactions on Computers, 70(9), pp.1484-1497.
8. Pathania, A., Jiao, Q., Prakash, A. and Mitra, T., 2014, June. Integrated CPU-GPU power management for 3D mobile games. In 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC) (pp. 1-6). IEEE.
9. Pathania, A., Irimiea, A.E., Prakash, A. and Mitra, T., 2015, June. Power-performance modelling of mobile gaming workloads on heterogeneous MPSoCs. In Proceedings of the 52nd Annual Design Automation Conference (pp. 1-6).
10. Somu Muthukaruppan, T., Pathania, A. and Mitra, T., 2014. Price theory based power management for heterogeneous multi-cores. ACM SIGPLAN Notices, 49(4), pp.161-176.

11. Mitra, T., Muthukaruppan, T.S., Pathania, A., Pricopi, M., Venkataramani, V. and Vishin, S., 2017. Power Management of Asymmetric Multi-Cores in the Dark Silicon Era. In The Dark Side of Silicon (pp. 159-189). Springer, Cham.
12. Prakash, A., Wang, S., Irimiea, A. E. and Mitra, T. Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms. In 2015 33rd IEEE International Conference on Computer Design (ICCD) (pp. 208-215).
13. Karunaratne, M., Mohite, A. K., Mitra, T., Peh, L. S. (2017, June). Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In Proceedings of the 54th Annual Design Automation Conference 2017 (pp. 1-6).
14. Li, Z., Wijerathne, D., Chen, X., Pathania, A. and Mitra, T., 2021. Chordmap: Automated mapping of streaming applications onto CGRA. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.
15. Wijerathne, D., Li, Z., Pathania, A., Mitra, T. and Thiele, L., 2021. Himap: Fast and scalable high-quality mapping on CGRA via hierarchical abstraction. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.
16. Wijerathne, D., Li, Z., Karunarathne, M., Pathania, A. and Mitra, T., 2019. Cascade: High throughput data streaming via decoupled access-execute cgra. ACM Transactions on Embedded Computing Systems (TECS), 18(5s), pp.1-26.
17. Li, Z., Wu, D., Wijerathne, D., Mitra, T. (2022, April). LISA: Graph Neural Network based Portable Mapping on Spatial Accelerators. In 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA) (pp. 444-459). IEEE.
18. Bandara, T. K., Wijerathne, D., Mitra, T., Peh, L. S. (2022, February). REVAMP: a systematic framework for heterogeneous CGRA realization. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 918-932).
19. Wijerathne, D., Li, Z., Bandara, T. K., Mitra, T. (2022, July). PANORAMA: Divide-and-Conquer Approach for Mapping Complex Loop Kernels on CGRA. In Proceedings of the 59th Annual Design Automation Conference 2022
20. Venkataramani, V., Pathania, A. and Mitra, T., 2020, March. Unified thread-and data-mapping for multi-threaded multi-phase applications on spm many-cores. In 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1496-1501). IEEE.
21. Mitra, T., 2015. Heterogeneous multi-core architectures. Information and Media Technologies, 10(3), pp.383-394.
22. Wang, S., Prakash, A. and Mitra, T., 2018, July. Software support for heterogeneous computing. In 2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI) (pp. 756-762). IEEE.
23. Prakash, A., Wang, S. and Mitra, T., 2020. Mobile application processors: Techniques for software power-performance optimization. IEEE Consumer Electronics Magazine, 9(4), pp.67-76.
24. ARM. Arm Compute Library. Available online: https://developer.arm.com/ip-products/processors/machine-learning/compute-library (accessed on 17 March 2022).
25. OAID. Tengine. Available online: https://github.com/OAID/Tengine (accessed on 17 March 2022).
26. Tencent. NCNN. Available online: https://github.com/Tencent/ncnn (accessed on 17 March 2022).
27. Xitao. https://github.com/CHART-Team/xitao. (accessed on 17 March 2022).
28. Minakova, S., Tang, E. and Stefanov, T., 2020, July. Combining task-and data-level parallelism for high-throughput CNN inference on embedded CPUs-GPUs MPSoCs. In International Conference on Embedded Computer Systems (pp. 18-35). Springer, Cham.
29. Tang, E., Minakova, S. and Stefanov, T., Energy-efficient and High-throughput CNN Inference on Embedded CPUs-GPUs MPSoCs. International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS).
30. Jeong, E., Kim, J., Tan, S., Lee, J. and Ha, S., 2021. Deep learning inference parallelization on heterogeneous processors with tensorrt. IEEE Embedded Systems Letters.
31. Tang, E. and Stefanov, T., 2021, December. Low-memory and high-performance CNN inference on distributed systems at the edge. In Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion (pp. 1-8).

32. Soomro, P.N., Abduljabbar, M., Castrillon, J. and Pericàs, M., 2021, May. An online guided tuning approach to run cnn pipelines on edge devices. In Proceedings of the 18th ACM International Conference on Computing Frontiers (pp. 45-53).
33. Scrugli, M.A., Meloni, P., Sau, C. and Raffo, L., 2021. Runtime Adaptive IoMT Node on Multi-Core Processor Platform. Electronics, 10(21), p.2572.
34. Wu, H.I., Guo, D.Y., Chin, H.H. and Tsay, R.S., 2020, August. 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) (pp. 46-49). IEEE.
35. Pricopi, M., Mitra, T. (2013). Task scheduling on adaptive multi-core. IEEE transactions on Computers, 63(10), 2590-2603.
36. Pricopi, M., Mitra, T. (2012). Bahurupi: A polymorphic heterogeneous multi-core architecture. ACM Transactions on Architecture and Code Optimization (TACO), 8(4), 1-21.
37. Mitra, T., Pricopi, M. (2017). U.S. Patent No. 9,690,620. Washington, DC: U.S. Patent and Trademark Office.
38. Kim, B., Lee, S., Trivedi, A.R. and Song, W.J., 2020. Energy-efficient acceleration of deep neural networks on realtime-constrained embedded edge devices. IEEE Access, 8, pp.216259-216270.
39. Zhong, G., Dubey, A., Tan, C. and Mitra, T., 2019. Synergy: An hw/sw framework for high throughput cnns on embedded heterogeneous soc. ACM Transactions on Embedded Computing Systems (TECS), 18(2), pp.1-23.
40. Zhong, G., Prakash, A., Liang, Y., Mitra, T., Niar, S. (2016, June). Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators. In 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC) (pp. 1-6). IEEE.
41. Zhong, G., Venkataramani, V., Liang, Y., Mitra, T., Niar, S. (2014, October). Design space exploration of multiple loops on FPGAs using high level synthesis. In 2014 IEEE 32nd international conference on computer design (ICCD) (pp. 456-463). IEEE.
42. E. Aghapour, D. Sapra, A. Pimentel and A. Pathania, "CPU-GPU Layer-Switched Low Latency CNN Inference," 2022 25th Euromicro Conference on Digital System Design (DSD), 2022.
43. Rapp, M., Pathania, A., Mitra, T. and Henkel, J., 2020. Neural network-based performance prediction for task migration on s-nuca many-cores. IEEE Transactions on Computers, 70(10), pp.1691-1704.