ARM-CO-UP: ARM COoperative Utilization of Processors

EHSAN AGHAPOUR, University of Amsterdam, Amsterdam, Netherlands
DOLLY SAPRA, University of Amsterdam, Amsterdam, Netherlands
ANDY PIMENTEL, University of Amsterdam, Amsterdam, Netherlands
ANUJ PATHANIA, University of Amsterdam, Amsterdam, Netherlands

HMPSoCs combine different processors on a single chip. They enable powerful embedded devices, which increasingly perform ML inference tasks at the edge. State-of-the-art HMPSoCs can perform on-chip embedded inference using different processors, such as CPUs, GPUs, and NPUs. HMPSoCs can potentially overcome the limitation of low single-processor CNN inference performance and efficiency by cooperative use of multiple processors. However, standard inference frameworks for edge devices typically utilize only a single processor.

We present the *ARM-CO-UP* framework built on the *ARM-CL* library. The *ARM-CO-UP* framework supports two modes of operation – Pipeline and Switch. It optimizes inference throughput using pipelined execution of network partitions for consecutive input frames in the Pipeline mode. It improves inference latency through layer-switched inference for a single input frame in the Switch mode. Furthermore, it supports layer-wise CPU/GPU *DVFS* in both modes for improving power efficiency and energy consumption. *ARM-CO-UP* is a comprehensive framework for multi-processor CNN inference that automates CNN partitioning and mapping, pipeline synchronization, processor type switching, layer-wise *DVFS*, and closed-source NPU integration.

Additional Key Words and Phrases: Edge Artificial Intelligence (AI), On-Chip Machine Learning (ML), Low-Power Design (LPD), Electronic Design Automation (EDA).

1 INTRODUCTION

Heterogeneous Multi-Processor Systems on Chips (HMPSoCs) consolidate multiple processors, including Central Processing Units (CPUs), Graphic Processing Units (GPUs), and Neural Processing Units (NPUs), onto a single chip [14]. Figure 1 shows the *RK3399Pro HMPSoC* within the *Rock Pi N10* embedded platform that exemplifies this consolidation. The *RK3399Pro HMPSoC* incorporates a hexa-core *ARM big.Little* asymmetric multi-core CPU, a quad-core *ARM Mali* GPU, and a dedicated NPU. The *ARM big.Little* CPU consists of two core clusters: a high-performance, high-power dual-core *big* CPU and a low-performance, low-power quad-core *Little* CPU. The CPUs, GPU, and NPU all support on-chip Machine Learning (ML) inference using Convolutional Neural Networks (CNNs) [25].

Figure 2 shows performance under single-processor inference tests conducted on the *Rock Pi N10* embedded platform. The figure shows that the *big* CPU or GPU can outperform the others in single-processor performance, depending on the CNN. The *Little* CPU exhibits a comparatively lower but still noteworthy throughput. Therefore, an embedded CPU is comparable to an embedded GPU in terms of performance and remains relevant for inference in embedded platforms [27]. However, the single-processor performance falls short of meeting the minimal user experience when running on the CPU or GPU alone [4]. Therefore, embedded applications require collaborative

Authors' addresses: Ehsan Aghapour, e.aghapour@uva.nl, Informatics Institute, University of Amsterdam, Amsterdam, Noord-Holland, Netherlands; Dolly Sapra, Informatics Institute, University of Amsterdam, Amsterdam, Noord-Holland, Netherlands, d.sapra@uva.nl; Andy Pimentel, Informatics Institute, University of Amsterdam, Amsterdam, Noord-Holland, Netherlands, a.d.pimentel@uva.nl; Anuj Pathania, University of Amsterdam, Amsterdam, Noord-Holland, Netherlands, a.pathania@uva.nl.

ACM 1084-4309/2024/4-ART
https://doi.org/10.1145/3656472

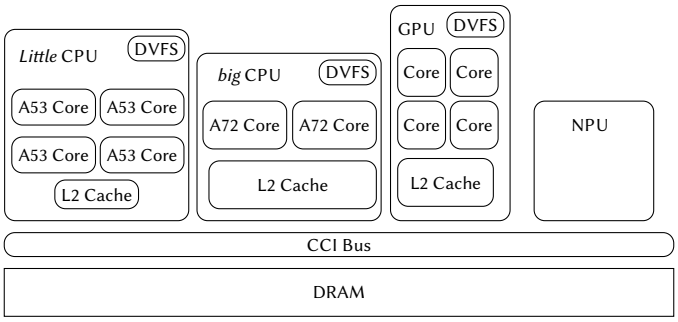


Fig. 1. An abstract block diagram of the *RK3399Pro* HMPSoC within *Rock Pi N10* embedded platform.

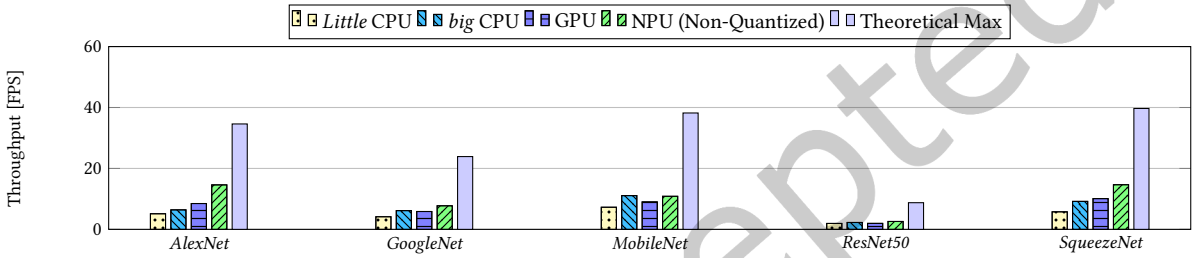


Fig. 2. Single-processor CNN inference throughput of different HMPSoC processors.

utilization of CPU and GPU to meet the requirements [1]. NPU can also provide a comparable non-quantized inference performance. Combining CPU, GPU, and NPU for inference opens up possibilities for high-performance inference, as shown by theoretical max in Figure 2.

We present the *ARM-CO-UP* framework that seamlessly integrates the NPU alongside the *ARM* CPU and GPU in a single CNN inference binary. *ARM-CO-UP* creates a streamlined backend engine that collaboratively executes inference tasks to minimize overhead. *ARM-CO-UP* framework builds on top of the *ARM Compute Library* (*ARM-CL*). *ARM-CL* supports highly optimized single-processor CNN inference on *ARM* CPUs or GPUs. *ARM-CO-UP* extends this default implementation to establish a multi-processor CPU-GPU-NPU inference environment to ensure a comprehensive utilization of computational resources. *ARM-CO-UP* is the first open-source framework to cooperatively utilize ARM-based CPUs, GPU, and vendor-specific NPUs without the need for access to the source codes of their respective libraries. This innovative approach allows for a more cohesive and streamlined computing environment.

Throughput and latency are the two preferred metrics for measuring the performance of CNNs [7]. *ARM-CO-UP* supports cooperative CPU-GPU-NPU inference in Pipeline and Switch modes for improving CNN inference throughput and latency, respectively. Pipeline mode inferences multiple frames simultaneously using a multi-stage CPU-GPU-NPU pipeline to improve throughput. Switch mode inferences one frame at a time on either CPU, GPU, or NPU but switches between them mid-inference depending upon the executing CNN layer to improve latency.

Another significant factor for CNN execution on embedded platforms is the power consumption incurred during inference [17]. The *ARM-CO-UP* plays a pivotal role in enhancing the power efficiency of CNN inference by supporting layer-level Dynamic Voltage and Frequency Scaling (DVFS). DVFS enables fine-grained control over power consumption, optimizing resource utilization without sacrificing performance [8]. This capability is crucial for efficient deployment of AI solutions in resource-constrained environments.

Novel Contributions: We make the following novel contributions with the *ARM-CO-UP* framework in this work.

- *ARM-CO-UP* allows for cooperative CPU-GPU-NPU CNN inference on edge devices in Pipeline and Switch modes to improve latency and throughput, respectively.
- *ARM-CO-UP* automates CNN graph partitioning into sub-graphs and subsequent sub-graphs to processors mapping at the granularity of CNN layers.
- *ARM-CO-UP* provide a model-independent implementation that facilitates adding new desired models. It even works for models with complex graphs containing branching blocks and shortcut branches between layers.
- *ARM-CO-UP* presents the APIs and command line options that enable setting the desired configuration, such as Pipeline or Switch mode, partitioning points, mapping to the processors, number of cores in CPU, frequency settings of *big* and *Little* CPUs and GPU, host CPU for GPU and NPU devices, etc.
- *ARM-CO-UP* provides a fine-granularity profiler for layer-level profiling of CNNs for metrics like performance, power, etc.
- *ARM-CO-UP* eases the integration of any new NPU without requiring its library source.
- *ARM-CO-UP* provides *Python* libraries that automate extracting parameters and splitting pre-trained models based on desired partitioning points compatible with popular frameworks, such as *TensorFlow*, *Caffe*, *Caffe2*, and *Keras*.

Open Source Contributions: The code for the *ARM-CO-UP* framework is publicly available for download at *https://github.com/Ehsan-aghapour/ARM-CO-UP* under *MIT* license.

2 RELATED WORK

Multi-processor CNN inference is an active research area. Most existing works on the subject create a multi-stage software pipeline [10, 18, 24, 29] between processors to improve CNN throughput. A software pipeline provides a mechanism to trade off throughput with latency. However, a software pipeline inherently by design cannot improve the inference latency. Some works attempt to improve latency by altering the underlying neural network [21, 30] or searching for an appropriate neural network [16, 19]. However, our work is independent of the software optimization endeavours. The primary focus of the *ARM-CO-UP* is to facilitate the cooperative and efficient utilization of processors to perform inference. Furthermore, our work complements initiatives focused on latency improvement through mapping optimization algorithms, such as searching for appropriate processor switch points [1, 2].

Table 1 qualitatively compares *ARM-CO-UP* with similar relevant frameworks for CNN inference on ARM-based HMPSoCs. *TVM* [5] and *ARM-CL* [23] are the popular frameworks for high-performance CNN inference on ARM CPUs. However, *TVM* and *ARM-CL* can only support one CPU at a time. Therefore, they under-utilize asymmetric multi-core CPUs. *ARM-CL* also support GPU-only CNN inference. *ARM-CL* can also perform NPU-only CNN inference using the *ARM Vela* compiler. However, *ARM Vela* only supports ARM NPUs such as *ARM Ethos*.

Authors of [24] introduce the *Pipe-it* framework based on the *ARM-CL*. *Pipe-it* creates a CNN inference pipeline between *Little* and *big* CPUs of *ARM big.Little* asymmetric multi-core CPUs. It also uses CNN micro-benchmarks to create model-based profiles for performance prediction. Similarly, authors of [3] introduce a framework called *PipeBert* based on *TVM*. *PipeBert* also creates a CNN inference pipeline between *Little* and *big* CPUs of *ARM big.Little* asymmetric multi-core CPUs. *PipeBert* primarily focuses on pipelined inference for BERT transformers but also supports CNN inference. Authors of [11] introduce a framework called *OmniBoost* [11] based on *ARM-CL*. *OmniBoost* supports a CNN inference pipeline between *Little* CPU, *big* CPU, and *GPU*.

The frameworks such as *ARM-CO-UP* are inherently architecture-specific. *ARM-CO-UP* focuses on generic ARM-based platforms. Similar comprehensive CNN inference frameworks designed for other platforms, such as those

Table 1. Qualitative comparison between different frameworks that support CNN inference on generic ARM-based HMPSoCs.

Framework	CPU		GPU	NPU	Pipeline	Switching	Layer DVFS	Profiling
	Symmetric	Asymmetric						
TVM [5]	✓	✗	✗	✗	✗	✗	✗	✗
ARM-CL + ARM Vela [23]	✓	✗	✓	*ARM* NPU	✗	✗	✗	✗
Pipe-it [24]	✓	✓	✗	✗	✓	✗	✗	Model-level
PipeBert [3]	✓	✓	✗	✗	✓	✗	✗	✗
OmniBoost [11]	✓	✓	✓	✗	✓	✗	✗	✗
ARM-CO-UP	✓	✓	✓	✓	✓	✓	✓	Layer-level

for *Nvidia* [9] and *Qualcomm* [20], are complementary and hard to compare against *ARM-CO-UP*. Furthermore, supporting multi-processor CNN inference [12, 20] for multiple networks is beyond the scope of *ARM-CO-UP*. *ARM-CO-UP* also does not support configurable CNN accelerators such as those based on Configurable Coarse-Grained Reconfigurable Arrays (CGRAs) [15, 26] and Field-Programmable Gate Array (FPGAs) [22, 28]. Moreover, *ARM-CO-UP* cannot distribute CNN inference workload across multiple HMPSoCs [6]. In future, we plan to add support for transformer inference to *ARM-CO-UP* [3].

ARM-CO-UP operates independently of methodologies that generate schedules for executing CNNs on heterogeneous platforms, such as SLO-aware inference scheduler [20]. ARM-CO-UP's primary function is to facilitate implementation and performance evaluation of schedules. *ARM-CO-UP* is a valuable tool for any scheduler, enabling performance assessment at intermediate stages that aid in optimal schedule development.

None of the frameworks above support both CPU-GPU-NPU pipelining or switching as ARM-CO-UP. *Furthermore,* ARM-CO-UP *is the only framework with fine-grained per-layer DVFS and profiling support and allows vendor-neutral NPU integration in ARM-based HMPSoCs.*

3 BACKGROUND

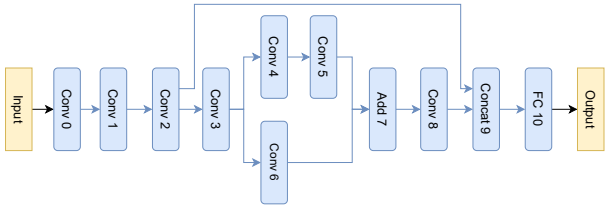
In this section, we delve into the foundational aspects of the *ARM-CL* framework, with a specific focus on its mechanisms for enabling efficient inference processing on edge devices. This exploration serves as a groundwork to understand the subsequent advancements introduced with *ARM-CO-UP*.

3.1 ARM-CL Library

The *ARM-CL* library utilizes its API to define a model architecture. Subsequently, based on the model, it employs the backend context to create and run an equivalent graph on the target processor. The graph manager oversees the graph configuration and execution. Furthermore, the graph manager is responsible for loading the input data and scheduling the workload functions on the target processor through its backend context. We delve next into the fundamental components of the *ARM-CL* framework to provide a detailed exposition of its structure.

Network Architecture. *ARM-CL* APIs provide the mechanism to define CNN model architecture. Using this API, a user can define specific layers and their interconnections. The definition starts with establishing a stream for the sequential addition of layers. This stream includes a graph sub-structure whose tail the stream tracks continuously. The stream generates a new node when an API adds a layer and attaches the node to the tail node of the graph. Figure 3 shows a simple structure (Figure 3a) and its pseudo-code definition (Figure 3b) in *ARM-CL*.

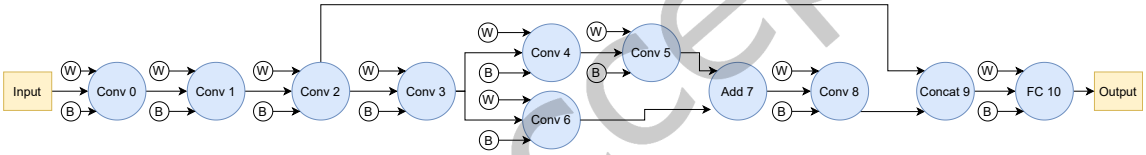
Graph. *ARM-CL* creates a graph corresponding to the desired CNN using its established network architecture, where the primary nodes represent the CNN layers and the tensors represent the connection between the layers. This representation encapsulates the architecture of the defined model and serves as the foundation for subsequent processing and computations within *ARM-CL*. *ARM-CL* creates `Const` nodes and connects them to the primary nodes for the trained parameters of a layer. *ARM-CL* considers the trained parameters of a layer (weights and biases) to be the layer operands along with the inputs from the other layers. Figure 3c shows the equivalent graph



(a) CNN structure

```
Stream main_stream;
main_stream << Input_Layer << Conv0 << Conv1 << Conv2;
sub_stream s1(main_stream); //Create substream (branch) from Conv 2
main_stream << Conv3;
sub_stream s2(main_stream); //Create substream (brach) from Conv 3
s2 << Conv4 << Conv5;
main_stream << Conv6;
main_stream << Add(main_stream, s2);
main_stream << Conv8;
main_stream << Concat(main_stream, s1);
main_stream << FC10 << Output_Layer;
```

(b) CNN Pseudo-code



(c) CNN Graph

Fig. 3. A sample CNN structure, along with its definition and graph representation in *ARM-CL*.

of the structure in Figure 3a. The graph contains the nodes and tensors that execute with *ARM-CL*. We display only the primary nodes of the graph throughout the remainder of this paper for brevity.

Node. An *ARM-CL* graph comprises various nodes, where each node correlates to a distinct layer, with each node type characterized by a specific number of inputs and outputs. For instance, the Node "Conv 3" depicted in Figure 3c represents the third convolution layer "Conv 3" shown in Figure 3a. Each node links with an associated tensor for every input or output edge. The node executes its operation on the input tensors (operands) and subsequently populates its output tensor.

Functions. *ARM-CL* implements multiple variants of functions for CPU and GPU processors for each node type. The best function variant for a node depends on the sizing parameters of the node (inputs, weights, and biases). It also depends upon the hardware specification of the target processor, such as the capacities of cache levels. *ARM-CL* implements essential deep-learning operations specific to its corresponding node through core kernels within each function. These kernels are designed for efficient execution on the CPU and GPU processors, leveraging technologies such as *NEON* for the CPU and *OpenCL* for the GPU.

ARM-CL employs *OpenCL* for GPU functions by providing a comprehensive framework for implementing and executing kernel functions on *ARM* GPUs. *OpenCL* excels in parallel computing for the complex calculations typically handled by GPUs. *ARM-CL*, in complement, employs *NEON* for CPU functions in *ARM* architectures.

NEON is a Single Instruction, Multiple Data (SIMD) Instruction Set Architecture (ISA) extension for high-performance parallel processing on CPUs.

Edge. *ARM-CL* implements edges housing a tensor to connect the graph nodes. The edges facilitate the flow of data between nodes. Each edge has a source and destination node, with the source node populating the tensor and the destination node accessing and utilizing the tensor as part of the overall computational flow. *ARM-CL* attributes the tensors of the `Input`, `Output`, and `Const` nodes to an accessor. The accessor is responsible for loading and pre-processing the input data and the weights. Additionally, the *Memory Manager* of the backend device is responsible for storing and managing the memory required for the tensor. Direct access to a tensor by different processors is not always feasible due to variations in memory configurations among processors.

Workload. *ARM-CL* creates a workload for the graph that loads the input data (image), executes the primary node functions, and post-processes the output data to generate the prediction results. The function factory of the backend device dynamically generates the most efficient function corresponding to each node based on the sizing parameters of the layer (node) and the hardware specifications of the backend device, such as cache sizes. The workload incorporates the accessors of `Input` and `Output` tensors, along with the graph functions.

Graph Manager. *ARM-CL* has a *Graph Manager* responsible for setting up the graph and executing the workload. After graph generation, the *Graph Manager* selects the backend device, configures the nodes, allocates the tensors, and calls the accessor for `Const` tensors to load the weights and biases into the relevant processor memory. Subsequently, the *Graph Manager* also manages the workload execution.

Scheduler. Within *ARM-CL*, there are separate CPU and GPU schedulers, each tasked with managing the scheduling of work processes for their respective processors. These schedulers play a crucial role in orchestrating the distribution of computational tasks, ensuring efficient parallel processing on both the CPU and GPU to maximize the overall performance. The CPU Scheduler is responsible for splitting the function process and scheduling it onto the processor threads. It employs two scheduling strategies – *Static* and *Dynamic*. The *Static* strategy divides data among threads for simultaneous processing. The *Dynamic* strategy partitions data into chunks, with each thread processing a chunk and requesting the next chunk upon completion. The *Dynamic* approach, helped by a *Feeder* class, optimally utilizes the threads, especially on cores with varying performance capabilities. The scheduler can configure the number of threads with or without affiliation to processing cores. The user must provide the underlying threads-to-cores mapping function to utilize the affiliation approach.

ARM-CL schedules its *OpenCL* kernels using the *CLScheduler* in the GPU processing workflow when a task invokes the associated function. The *CLScheduler* offloads an *OpenCL* kernel to the GPU by placing it into the command queue of the GPU processor. Internally, the GPU scheduler handles the kernel execution from the command queue into the processing resources. This transfer process between the CPU and GPU is asynchronous and non-blocking.

Backend Context. Within *ARM-CL*, backend contexts are crucial in computational graph execution on CPU and GPU processors. Each distinct context tailors to its specific processor type, and selecting a processor for the graph execution employs the corresponding backend context. Choosing a CPU or GPU initiates the backend context for CPU or GPU, respectively. The backend contexts are responsible for initializing and setting up their respective processors, creating tensors, and generating functions particular to each node. Additionally, they oversee the memory allocation for weights and activation data during run-time, ensuring systematic and efficient computation.

CPU Backend Context. *ARM-CL* provides a CPU backend context to navigate the operations tailored for the CPU execution. The context initializes the desired number of threads in the scheduler and handles memory allocation for tensors, focusing on weights and activation data. The function factory selects and generates the most efficient variant of implemented functions for each node within this context. These functions incorporate *NEON* kernels optimized to facilitate basic deep-learning operations on the CPU.

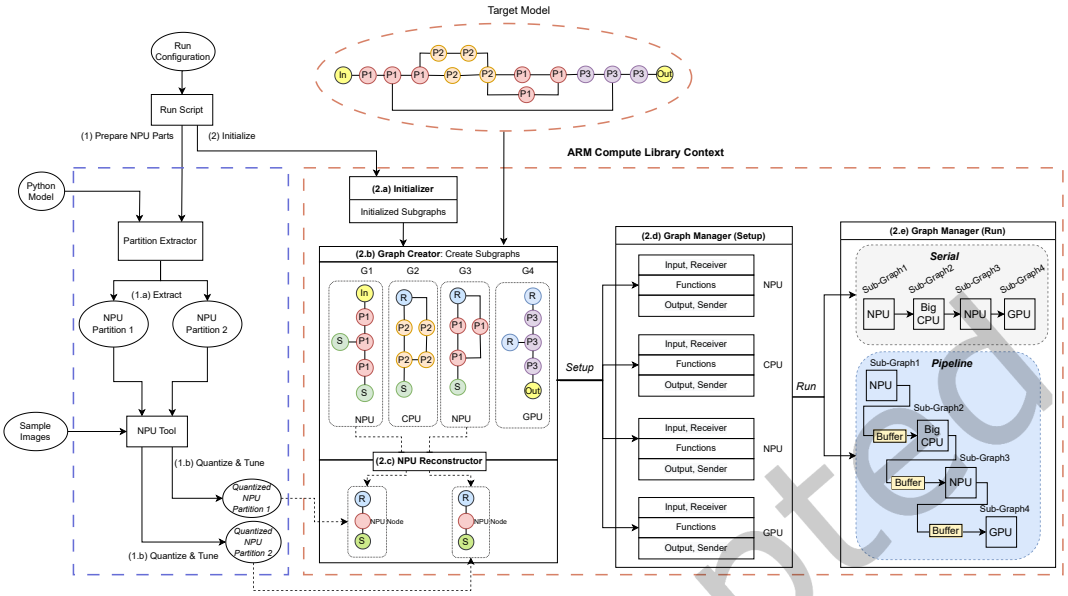


Fig. 4. *ARM-CO-UP* design Flow for an example that utilizes CPU, GPU, and NPU.

GPU Backend Context. Similarly, a GPU backend context within *ARM-CL* manages operations specific to GPU devices, initiating the *OpenCL* scheduler to coordinate the deployment of *OpenCL* kernels. This context involves identifying the GPU processor, assigning it to the *OpenCL* device, generating an *OpenCL* context, and initializing the *OpenCL* queue for the associated device and context. The function factory within this context opts for the most efficient variant of implemented *OpenCL* functions per node.

4 ARM-CO-UP FRAMEWORK

The *ARM-CO-UP* framework builds on top of the *ARM-CL* Library. The original *ARM-CL* supports CNN inference with either CPU or GPU processors. In contrast, *ARM-CO-UP* focuses on the cooperative use of the available processors simultaneously – CPU, GPU, and NPU – for inference. We explain next the various components in the *ARM-CO-UP* framework.

4.1 Co-operative utilization

A CNN graph can execute with a CPU or GPU processor by default. The distinct backend context associated with each processor does not allow for a collaborative execution of CNN. The *ARM-CO-UP* introduces the concept of sub-graphs for simultaneous model inference with different processors. It defines sub-graphs that execute on separate processors and memory spaces with separate backend contexts. Consequently, a sub-graph is free to map to its processor for execution. Pipelining and switch mechanisms can employ sub-graphs to improve the overall performance of the CNN using multiple processors. Therefore, it becomes necessary to manage the transfer of intermediate data between sub-graphs and coordinate the execution of these individual sub-graphs. The *ARM-CO-UP* provide `Receiver` and `Sender` nodes (and associated tensors) to extend the capabilities of the *Graph Manager*. We elaborate on these components next and explain the workings of the sub-graph.

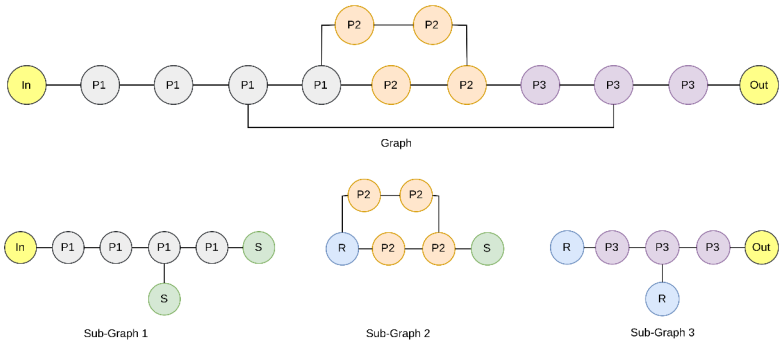


Fig. 5. *ARM-CO-UP* partitioning of a graph into three sub-graphs to run on three different processors.

Sub-Graph. The *ARM-CO-UP* structures the model as sub-graphs rather than a comprehensive graph determined by the target processor assigned to its nodes. Figure 5 demonstrates an equivalent graph of a model and the mapping of its nodes to the target processor. Based on the layer mapping, the consecutive layers with the same target processor constitute a sub-graph. Each sub-graph has the same target processor, and a unified backend context is established for the sub-graph on the target processor, overseeing their execution on that specific processor. *ARM-CO-UP* offers users two distinct modes for inference using sub-graphs: Pipeline and Switch mode.

Pipeline Mode. In the Pipeline mode, every sub-graph operates as a distinct stage in the overarching pipeline. As a sub-graph concludes its workload execution, it transmits its data and either commences processing the subsequent data in its queue or momentarily halts if the input data isn't yet available. This mechanism ensures that the sub-graphs, representing different pipeline stages, operate concurrently for consecutive input frames. Such a methodology empowers users to harness processors collaboratively, enhancing the throughput and energy efficiency of the inference.

Switch Mode. In the Switch mode, sub-graphs operate serially for each frame, eliminating any parallel operation. Here, the inference process for an image switches between processors. This mode allows the flexibility to allocate layers to the most suitable processor, optimizing energy efficiency and end-to-end latency for individual frames.

Sender and Receiver Nodes. The *ARM-CO-UP* creates sub-graphs within different backend contexts. Therefore, data transfer is necessary between the processors in intermediate terminal nodes of the sub-graphs. The `Sender` and `Receiver` nodes add to the source and destination of the connection between two sub-graphs. Figure 5 shows the `Sender` and `Receiver` nodes in the intermediate terminals of the sub-graphs. The *ARM-CO-UP* establishes an edge, via an associated `Sender` tensor, between the last node in the source sub-graph and the attached `Sender` node. Additionally, within the subsequent sub-graph, it creates an edge between the `Receiver` node and the first node and creates a `Receiver` tensor for it.

Sender and Receiver Tensors. The `Sender` and `Receiver` tensors, integral components within the *ARM-CO-UP* framework, facilitate data transfer across processors' backend contexts. These tensors are embedded with attributes and mechanisms to synchronize and communicate data effectively. Specifically, the `Sender` tensor holds `Receiver` tensors as its data transfer targets. Figure 6 shows the structure and partitioning configuration within the *ARM-CO-UP*, wherein there are two receivers for the sender of the first sub-graph. The `Sender` tensor in the first sub-graph dispatches data to `Receiver` tensors located in both the second and third sub-graphs.

The *Graph Manager* delineates receiver nodes for each sender tensor, forming sub-graphs during the setup. Subsequently, upon completing sub-graph node tasks, the *Graph Manager* triggers its senders as outlined in

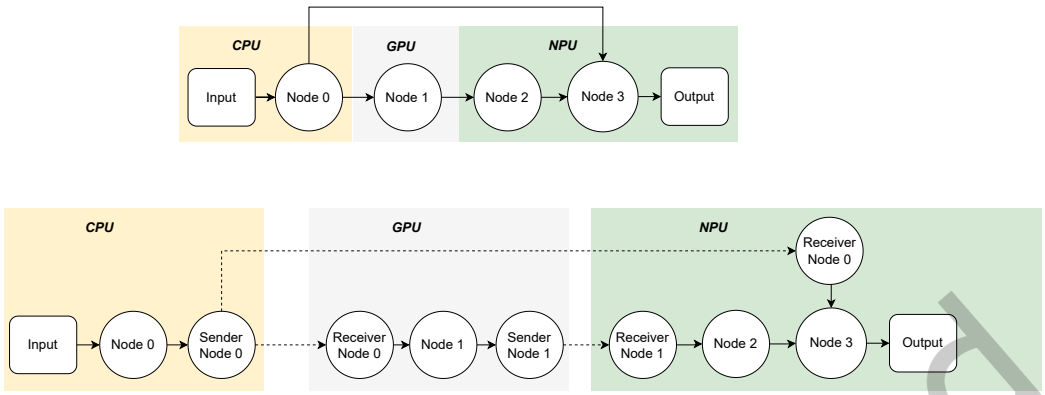


Fig. 6. Structure and partitioning configuration where there are two receivers for sender of the first sub-graph.

Algorithm 1: Sending Data

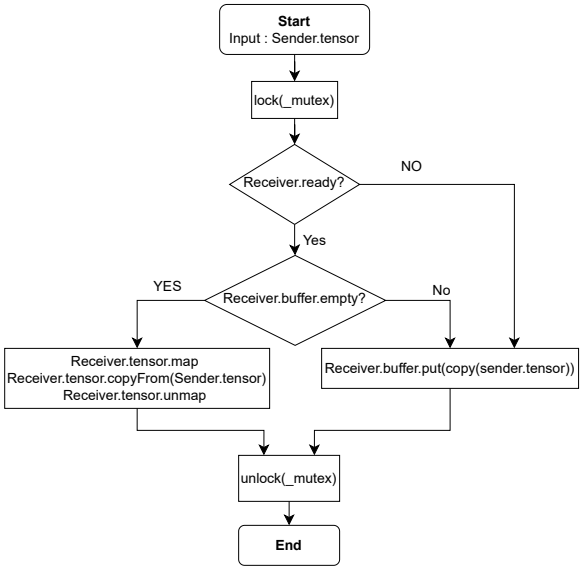
```
foreach sender ∈ graph.senders do
    map(sender.tensor to main memory);
    foreach receiver ∈ sender.receivers do
        receiver.transfer(sender.tensor);
    unmap(sender.tensor);
```

Algorithm 2: Receiving Data

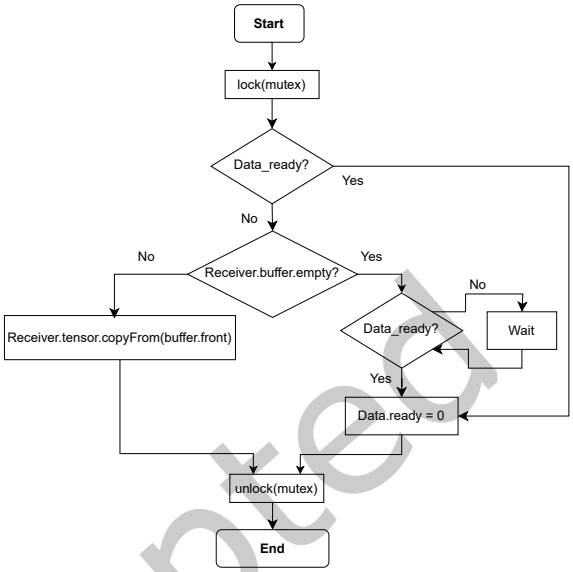
```
foreach receiver ∈ graph_receivers do
    receiver.set_ready();
foreach receiver ∈ graph_receivers do
    receiver.receive_data;
```

Algorithm 1 at the run-time. Each sender invokes the transfer function for its linked receiver nodes. Figure 7a depicts the transfer function's methodology. The methodology commences with mutex utilization to prevent race conditions with the receiver thread of the destination sub-graph. It then ascertains receiver readiness and buffer status. If the receiver awaits data and its buffer is vacant, the sender directly transmits its tensor data to the receiver's memory in a different processor, simultaneously notifying the receiver. Conversely, if the receiver is preoccupied or the buffer is non-empty, the sender's tensor data is queued in the receiver's buffer.

The receivers in each sub-graph precede node task execution, as depicted in Algorithm 2. Each sub-graph sets its receivers to a ready state and initiates their receive functions. This process, illustrated in Figure 7b, involves the receiver examining the `Data_ready` status. If true, it indicates the sender has already populated the receiver's tensor memory, requiring no further action. If false and the buffer contains data, the receiver transfers the earliest buffered data to its tensor memory. If the buffer is empty, the receiver employs a condition variable mechanism (*Condvar* in *C++*) for efficient wait management, pending data transfer from the source



(a) *transfer* function: Transferring sender's tensor data of the source sub-graph into the receiver of the destination sub-graph or its buffer



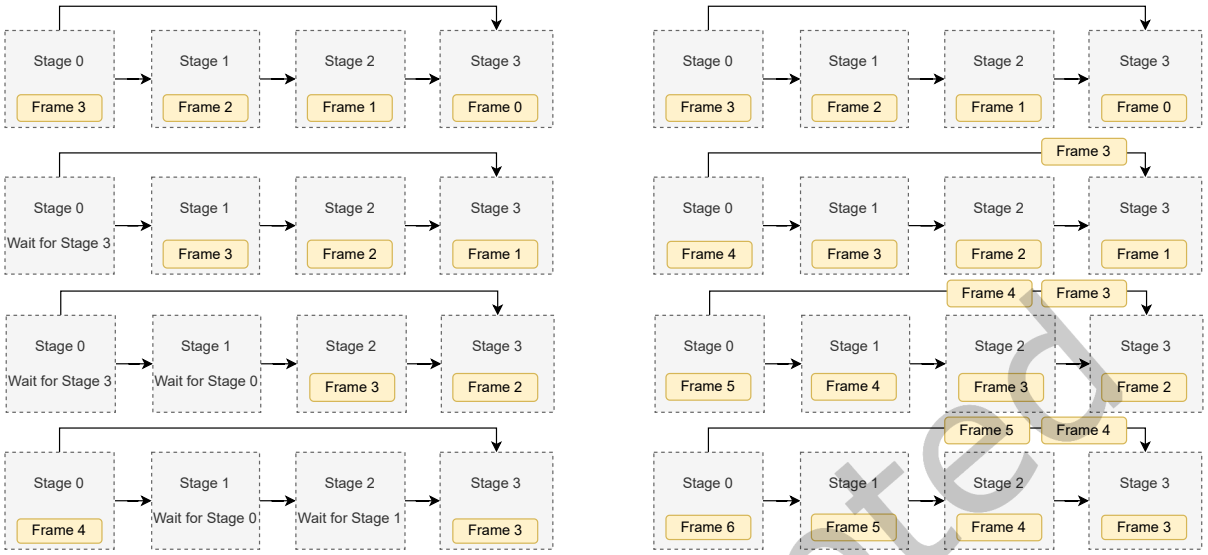
(b) *receive* function: Read data from its buffer, or wait for transferring data from the sender of the source sub-graph

Fig. 7. The *ARM-CO-UP transfer* and *receive* functions in source and destination sub-graphs respectively.

sub-graph. Upon data transfer completion by the sender, which also sets `Data_ready` to true, the receiver resumes processing.

The queue buffer of the receiver plays a pivotal role, accommodating instances where the sender has readied the data but the receiver is not prepared to accept it, often due to the ongoing processing of preceding data. This buffer ensures continuous, seamless data flow between sender and receiver tensors. Consider a scenario where sub-graphs execute concurrently in the parallel mode across consecutive frames using a software pipeline. A branch extends from the first to the fourth stage (sub-graph). Figure 8 depicts sub-graphs formed based on node-to-processor mappings. Pipeline stages process consecutive frames. While the first stage ($stage_0$) processes frame i, $stage_n$ processes frame $(i - n)$. Upon completing the execution of frame number i by the first stage, it sends data to the second and fourth stages. However, a direct data transfer to the fourth stage is not feasible. The fourth stage has just concluded processing frame $i - 3$ and must next process frame $i - 2$ that the third stage has just finished. Without a buffer for the first stage to place data from frame i, it cannot process frame $i + 1$ for the following two pipeline clocks. This lack of buffer causes two stalls in the first stage of the pipeline during the subsequent clocks. These stalls propagate to the end of the pipeline stages. As soon as the fourth stage receives the frame i data from the third stage, it can be processed, and the first stage can deliver the frame $(i + 1)$ and start processing the next frame $(i + 2)$. Therefore, during each pipeline clock, the two stages experience stalls, resulting in only two active stages, as opposed to all four. Consequently, *ARM-CO-UP* incorporates buffers into `Receiver` tensors to minimize the pipeline stalls.

Graph Management. The *ARM-CO-UP* extends the original *ARM-CL* graph management to manage the coordination and execution of various sub-graphs in Pipeline or Switch mode. The *ARM-CO-UP* equips the *Graph Manager* with the list of sub-graphs, their backend contexts and workloads. The setup of a full graph is a



(a) Without buffer (b) With buffer

Fig. 8. A pipeline configuration example necessitating the need for buffers.

time-consuming process involving the preparation of the workload and loading the layer parameters into the memory of the target processor using the corresponding backend context. *ARM-CO-UP* partitions the graph into multiple sub-graphs according to the mapping of the layers to processors. Each sub-graph has its context for setup, so the *ARM-CO-UP* establishes the sub-graph configurations concurrently, effectively reducing the overall setup time. *ARM-CO-UP* exploits a multi-threaded approach for executing the sub-graph workloads on their target processors using a host CPU for each sub-graph. A thread to manage the task execution spawns for each sub-graph and pins to the host CPU cores. The *ARM-CO-UP* can select the sub-graph host(s) among the CPU core(s). The receivers and senders are responsible for receiving and sending the data from and to the source and destination sub-graphs, respectively.

Scheduler. A scheduler divides the workload across all available threads within the original *ARM-CL* library. These threads execute across all cores in the asymmetric CPUs [13]. However, the communication cost between different CPUs can be prohibitively high, even though they may share the same backend context [24]. Consequently, the *ARM-CO-UP* establishes separate schedulers for the CPUs. Each CPU is an independent processor tasked with processing a specific sub-graph. The host, assigned to a particular sub-graph, invokes the relevant scheduler, which then allocates the sub-graph to the cores within its corresponding CPU. This strategy reduces the need for extensive communication between CPUs. Inter-CPU communication is reserved only for boundary layers, which relay their data to the other CPU to process subsequent sub-graphs.

Figure 9 shows the performance benefits of using a separate scheduler for each CPU versus a unified scheduler for multiple CPUs in an asymmetric multi-core. The figure shows two distinct configurations: one where inference executes jointly on a combination of *Little* and *big* CPUs and another utilizing a two-stage pipeline involving *Little* and *big* CPUs. The comparative analysis underscores the efficiency gains achieved by the two-scheduler approach, where workload distribution and reduced inter-CPU communication contribute to enhanced system performance.

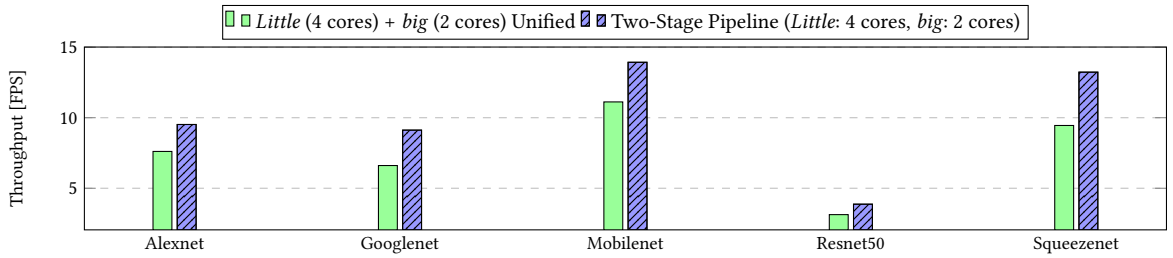


Fig. 9. Performance comparison between unified and separate scheduler designs.

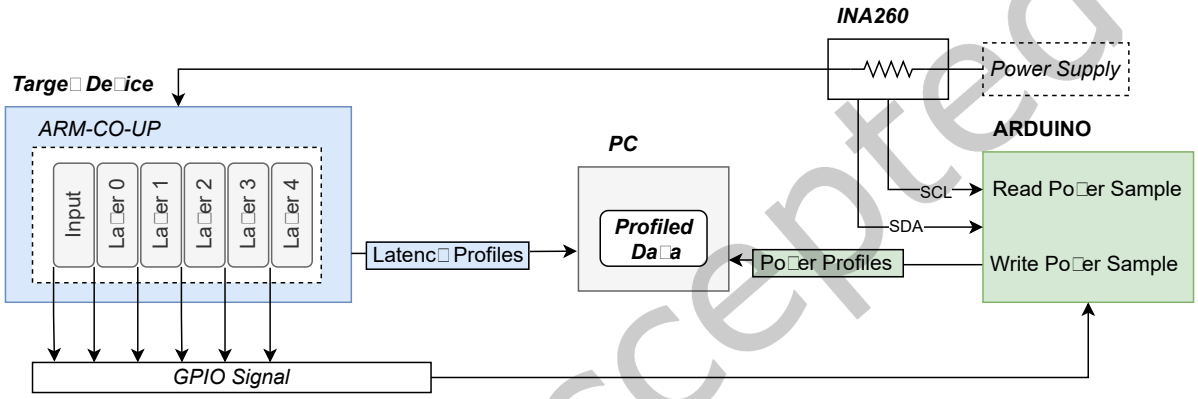


Fig. 10. An abstract diagram for integration of an external power-performance profiling setup with a target device using *ARM-CO-UP*.

4.2 Profiling

The *ARM-CO-UP* provides detailed profiling for both execution time and power consumption of individual layers while taking inter-layer communication into account. Each task within the workload tracks its execution duration. Upon request by the *Graph Manager*, the average execution time across all frames is computed and relayed for reporting purposes. The *Graph Manager* monitors the timing for communication, input, and output operations.

Furthermore, the *ARM-CO-UP* supports *GPIO* signals, enabling external power measurement setups. These signals mark the commencement and conclusion of each layer's processing, thereby facilitating layer-specific power measurements. Figure 10 illustrates the power measurement configuration utilized by *ARM-CO-UP* for layer-wise power consumption analysis. When power measurement activates within *ARM-CO-UP*, it transmits signals to the *ARDUINO* board. Consequently, the *ARDUINO* captures power samples and tags them with these signals. This procedure ensures *ARM-CO-UP* extracts the power samples for each layer's execution cycle [2].

4.3 NPU Integration

The NPU is a dedicated Application-Specific Integrated Circuit (ASIC) accelerator processor integrated into the latest edge HMPSOCs to optimize power and performance for neural network inference. The NPU operates with lower precision operation units for significantly higher performance and energy efficiency. Therefore, it is essential to integrate this specialized processor with the CPU and GPU processors in embedded devices.

The *ARM-CL* has no backend context for the NPU. Creating an NPU context presents significant challenges, primarily because the libraries for the NPU are not open source. Additionally, the NPU supports the execution of networks in various formats, adding complexity in integrating a dedicated context within *ARM-CL*. This lack of a standardized, accessible backend for the NPU complicates its incorporation and utilization. However, the *ARM-CO-UP* successfully incorporates NPU alongside CPU and GPU cores. The integration of NPU in the *ARM-CO-UP* involves harmonizing the distinct contexts and ensuring compatibility with several accelerators, each with specific libraries and APIs.

ARM-CO-UP adds an interface layer to the top of the *ARM-CL* to achieve NPU integration. This *Python*-based layer manages the sub-graphs of the pre-trained model derived from established *Python*-based frameworks. The layer also allows for an efficient extraction and conversion of the relevant parts of the model, which are marked to execute on the NPU. Concurrently, *ARM-CO-UP* integrates NPU generic functions, backend, and node classes into the core of *ARM-CL*. This design allows defining the NPU configuration based on the specific NPU integrated with their embedded device. These additions strengthen the integration of NPUs into existing CPU and GPU environments. The goal is to ensure smooth integration and extend the capabilities of *ARM-CL*.

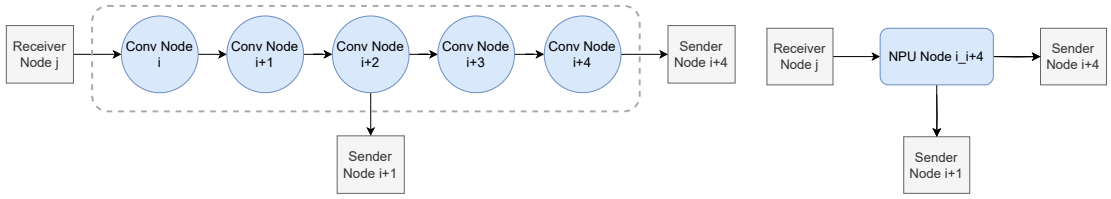
We elaborate next on this newly added interface layer and its position within the *ARM-CO-UP*. Additionally, we comprehensively analyze the NPU's generic functions, backend, and node classes, highlighting their pivotal role in supporting accelerators without being restricted to particular contexts and libraries.

Interface Layer. The *ARM-CL*, developed in *C++*, is tailored for optimal efficiency on edge devices. It provides specialized APIs that outline the neural network's architecture and layers. On the contrary, most neural network models originate from *Python*-based libraries such as *TensorFlow*, *Keras*, *Caffe*, *PyTorch*, etc. As a result, accelerators and NPUs predominantly interact with models from these libraries. Once these *Python*-centric models translate into the accelerator-specific format, the accelerators offer dedicated APIs, typically in *Python* and *C++*, that handle tasks such as model loading, input loading, inference execution, and output extraction. Therefore, an interface layer is required to fulfil several requirements. This layer segments, extracts, and prepares the parts of the model that execute within the NPU context. Based on the layer-to-processor mapping, the interface layer extracts the NPU partitions. It adds the input and output layers and saves the partition for the upcoming processing. Then, it converts the extracted partition to the NPU format using the NPU-specific tools. In this step, it quantized the NPU partitions of the model, if required.

The interface layer, for each sub-graph, provides unique terminology based on the input and output layer indexes. This naming convention allows the NPU node in the *ARM-CO-UP* to locate and load the corresponding NPU-specific model partition for later execution. The *ARM-CO-UP* can identify and retrieve the appropriate model sub-graphs based on the specified partition points (input and output layer indexes). The interface layer streamlines and automates the workflow, allowing models developed in popular *Python* libraries to execute effortlessly by the *ARM-CO-UP*.

NPU Node. *ARM-CO-UP* introduces an NPU node, expanding the available variety of node types within the *ARM-CL*. The method used to create sub-graphs for the NPU is similar to those for CPU and GPU sub-graphs, ensuring a consistent approach across the *ARM-CO-UP* framework. However, since the design and functionality of NPU differ from the already supported CPU and GPU contexts, adjustments to the NPU sub-graph are necessary. Therefore, as depicted in Figure 11, *ARM-CO-UP* reconstructs the NPU-target sub-graphs. For this purpose, it replaces all the internal nodes in the sub-graph with the NPU node and then connects all terminal nodes and their associated tensors to this NPU node after creating an NPU sub-graph. This approach treats the entire NPU sub-graph as a single NPU node connected to other sub-graphs using regular edges.

The *ARM-CO-UP* begins by creating an NPU node. It then updates the connections to link the terminal nodes to the sub-graph internal nodes and then connects them to the NPU node instead. Figure 11a displays a sub-graph for the NPU, built using the *ARM-CL* API and context and representing the model layers. Terminal nodes, shown as squares, include `Input`, `Receiver`, `Sender`, and `Output` nodes, while the internal nodes, which represent model



(a) NPU sub-graph initially created same as for CPU and GPU

(b) Reformed sub-graph for NPU

Fig. 11. An abstract diagram depicting the process of NPU reconstruction within *ARM-CO-UP*.

layers, are shown as circles. The *ARM-CO-UP* disconnects the connections between the terminal and internal nodes and removes internal nodes. Then, it creates an NPU node with the name embedding the starting and ending indices of the original nodes for naming convention. Finally, it redirects the connection from the terminal nodes to the NPU node. Figure 11b shows the updated sub-graph after these changes. This approach allows the creation of sub-graphs regardless of the specific type of NPU (or accelerator) that will execute them.

The terminal nodes transfer data between the NPU and other sub-graphs. They load input data sent by other sub-graphs into the NPU's memory. Subsequently, they get the output from the NPU sub-graphs and pass it on to the next sub-graphs through the tensors of the connecting edges. Since data formats and types might differ between NPUs and CPU or GPU processors, these terminal nodes adjust the data type and format between sub-graphs based on the *ARM-CO-UP* configuration.

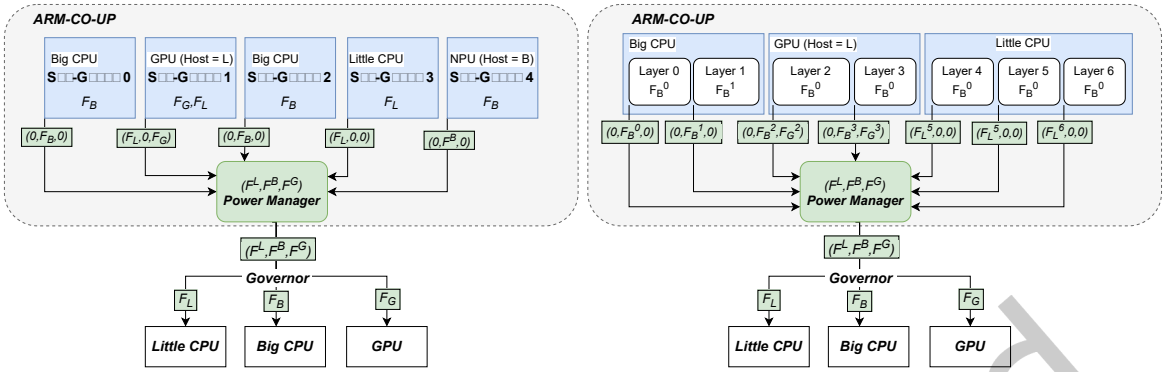
NPU Backend Context. When *ARM-CO-UP* creates a graph (or sub-graphs) for a neural network model, the *Graph Manager* produces the workloads of these sub-graphs using the function factory of the associated backend context. The *ARM-CO-UP* introduces the NPU backend context to manage the creation and execution of the NPU-based model execution function. This context comes equipped with a function factory designed to craft an NPU function specifically for the NPU node. Notably, the NPU backend context is a universal backend suitable for all NPU varieties. It establishes a generic NPU function template, linking it to the NPU type defined.

NPU Function. The NPU function introduced in the *ARM-CO-UP* acts as a versatile template that builds upon the foundational function type present in the *ARM-CL*. It retains the primary characteristics of the *ARM-CL* functions that invoke during workload execution. This function consists of two parts: the configuration of the NPU by loading the model and the execution, which handles the loading of input tensors, executing the model sub-graph, and retrieving the outputs.

The NPU function, as a template class, accommodates a range of NPU-specific APIs. Given that different NPUs possess distinct APIs for model loading and execution, this method ensures that incorporating a new NPU type is streamlined. Users can extend support to any new NPU by merely integrating its unique API into the pre-established template associated with a specific NPU classification. Consequently, when generating the NPU function for an NPU Node, the *ARM-CO-UP* employs the definitions tied to the NPU for model loading and execution. Beyond APIs, custom binary implementations of the shared libraries accompany each NPU. The *ARM-CO-UP* seamlessly manages the task of integrating these libraries into the finalized executable binary. This integration allows easy incorporation of a new NPU variant. A user only deposits the shared libraries pertinent to that NPU in the specified NPU libraries directory (Libs/NPU/) in the *ARM-CO-UP*.

4.4 Power Manager

The *ARM-CO-UP* is additionally equipped with DVFS to regulate the power consumption of the CPU and GPU during inference. *ARM-CO-UP* allows adjusting of processor voltage and frequency for each sub-graph or individual layer. In sub-graph-level DVFS, users define the DVFS levels for each sub-graph, and the *ARM-CO-UP*



(a) Sub-graph-level power manager

(b) Layer-level power manager

Fig. 12. *ARM-CO-UP* power manager demonstration for approaches at different level.

power manager accordingly adjusts the frequency settings for the processor assigned to each sub-graph. *ARM-CO-UP* makes this adjustment using platform-specific system commands. These commands are configurable within the power manager to accommodate platform-specific idiosyncracies. The operational modes of the power manager vary. In Switch mode, the power manager adjusts the DVFS levels of the processors for the upcoming sub-graph upon completion of the current one. In Pipeline mode, it operates differently as all sub-graphs execute simultaneously on their designated processors. *ARM-CO-UP* tasks each processor with a single sub-graph in the pipeline mode. Therefore, the DVFS levels for multiple sub-graphs on the same processor should be identical. Hence, in Pipeline mode, it is only necessary to set the processor frequency levels once during the initial setup.

Contrastingly, layer-level DVFS allows for more granular control, where users can specify DVFS levels for each layer. The *ARM-CO-UP* system dynamically adjusts the DVFS level of the respective processor according to the setting chosen for each layer during the inference process. Given the short execution time of layers, this mode necessitates a swift DVFS mechanism to ensure minimal delay in applying the desired voltage and frequency settings to the hardware. *ARM-CO-UP* incorporates a DVFS class that can interface with a kernel-level DVFS governor. After integrating the kernel-level DVFS governor into the kernel space, the API for the user-defined governor is responsible for setting processor frequency within *ARM-CO-UP*'s DVFS class. This integration empowers *ARM-CO-UP* to modify processor frequency on a per-layer basis. As a practical example, a kernel-level DVFS governor has been incorporated into *ARM-CO-UP* for the *Rock Pi N10* board to facilitate layer-level DVFS. Figures 12a and 12b demonstrate the DVFS mechanisms at sub-graph and layer levels, utilizing system and kernel governors.

5 ARM-CO-UP METHODOLOGY (WORKFLOW)

This section delves into the methodology and workflow of the *ARM-CO-UP* framework. The *ARM-CO-UP* framework first prepares the NPU partitions of the model, adapting them based on user-defined mapping. Second, it configures the DVFS and power management components for the CPU/GPU. Subsequently, the creation of sub-graphs ensues, structured according to node mapping. Finally, *ARM-CO-UP* performs careful configuration of processors and their associated sub-graphs. It then executes these sub-graphs in their predetermined modes.

Central to *ARM-CO-UP* is the *Run Command*, which encapsulates all user-defined configurations for the inference procedure. The users do not need to manually adjust the model, *ARM-CO-UP*, or processor settings

with this feature. Instead, they can directly utilize the *Run Command* options to perform inference according to their preferences.

```
$ ./graph_alexnet_co_up --threads=4  --threads_little=2 --n=60 --cores=2 --cores_little=4 --order=BBLGGNNN --
    mode=pipeline --frequency=7-4-6-[3,2]-[4,4]-1-1-1
```

Table 2 provides a comprehensive list of configuration options for executing the inference. The options encompass mapping model layers to specific processors, choosing between pipeline or switch modes, designating the number of threads for CPUs, electing appropriate hosts for GPU and NPU devices, modulating processor frequencies, and determining the profiling level. Finally, *ARM-CO-UP* executes these sub-graphs in their predetermined modes. We delve deeper into each stage: *Pre-Setup*, *Sub-Graph Creation*, *Setup*, and *Run Inference*.

5.1 Pre-Setup

The *ARM-CO-UP* framework begins its operational sequence by preparing the model according to the desired mapping. Initially, it identifies and transforms the segments of the model designated for the NPU via the interface layer. It achieves this by isolating the specified segments from the original Python-based model and then appending them with necessary input(s) and output(s). It then translates these isolated segments into a format compatible with the target NPU. During this transformation phase, it applies an optional quantization step depending on the user preferences. It saves each process segment under a distinctive naming convention to ensure seamless identification in subsequent phases. The naming comes from the starting and ending layers. This systematic naming approach facilitates *ARM-CO-UP*'s ability to locate swiftly the pertinent segments.

Beyond model preparation, *ARM-CO-UP* also undertakes system-level initialization. *ARM-CO-UP* then configures the DVFS governor and activates power measurement components. If the user opts for additional configuration capabilities, *ARM-CO-UP* spawns a DVFS object instance, empowering users to tweak the DVFS settings of the processors. Concurrently, it also establishes the GPIO pins earmarked for signal transmission.

5.2 Sub-Graph Creation

Following the model preparation and power management configuration, *ARM-CO-UP* initializes sub-graphs by the chosen mapping per model layer. When adding the layers, a corresponding node is instantiated and incorporated into the designated sub-graph. The *ARM-CO-UP* framework takes charge of this node addition process, ensuring that each node is aptly placed within its relevant sub-graph and automatically creates the required interconnections.

The *ARM-CL* algorithm uses a single graph for the entire model containing all nodes and edges. In contrast, the *ARM-CO-UP* allows adding nodes and edges to specific sub-graphs and automatically creates interconnections between different sub-graphs. *ARM-CO-UP* examines the input nodes associated with the node before adding it to a sub-graph. If an input node is missing in the current sub-graph, an interconnection is established between the source node and the new node, even if they belong to different sub-graphs. *ARM-CO-UP* appends a `Sender` node to the source node (in the source sub-graph) and inserts a `Receiver` node before the new node (in the target sub-graph) to achieve this. The address of the `Sender` tensor (with the `Sender` node) is in the `Receiver` tensor (with the `Receiver` node). It saves the address of the `Sender` tensor in the `Receiver` tensor associated with the `Receiver` node. This step ensures data communication and synchronization between these nodes during run-time. *ARM-CO-UP* tracks the mapping of nodes to their `Sender` and `Receiver` nodes. Subsequent nodes can directly use this existing `Receiver` node for interconnection, rather than creating a new one, if the `Receiver` node for an input node already exists in the current sub-graph. This automated, model-independent graph creation process enables the algorithm with new models without significant additional effort. Furthermore, this mechanism is effective even for complex models with branches and shortcuts, such as in Figure 5.

Table 2. Configuration Attributes for Inference Run Commands

Attribute	Purpose	Details
order	Maps layers to processors	Determine the processor type for each layer. 'L' for *Little CPU*, 'B' for *big CPU*, 'G' for *GPU*, and 'N' for *NPU*.
mode	Defines execution mode	Either 'Pipeline' or 'Switch'
threads	Number of threads for *big* CPU	Number of threads within *big CPU*, that distribute tasks of a layer to them (from 1 to the number of *big* cores)
cores	Number of cores in *big* CPU of the platform	For platforms with different number of cores in *big CPU*
threads_little	Number of threads for *Little CPU*	Number of threads within *Little CPU*, that distribute tasks of a layer to them (from 1 to number of *Little* cores).
cores_little	Number of cores in *Little CPU* of the platform	For platforms with different number of cores in *Little CPU*
n	Number of frames	To set the number of frames that run the inference (is useful for measuring the average performance metrics)
frequency	Frequency of the layers/sub-graphs	Frequency indexes separated with '-' for layers/sub-graphs
host_gpu	The host processor for *GPU* device	Either 'B' or 'L' that means *big* or *Little CPU*, respectively
host_npu	The host processor for *NPU* device	Either 'B' or 'L' that means *big* or *Little CPU*, respectively
profile	Set the level of profiling	The possible profiling levels are Level 0: reporting the overall latency and throughput, Level 1: execution and transfer time of the sub-graphs, and Level 2: execution and transfer time for each layer

ARM-CO-UP provides scalable and efficient sub-graph management by creating sub-graphs within the core of the ARM-CL. Consequently, introducing new models becomes straightforward, eliminating any need to alter the existing model code. *ARM-CO-UP* refines the NPU sub-graph once it has established the sub-graphs. It accomplishes this by substituting its internal nodes with a singular NPU node.

5.3 Sub-Graph Management

Graph Manager within *ARM-CO-UP* sets up and executes the sub-graphs. Original *ARM-CL* works with a single graph representing the model. However, in *ARM-CO-UP*, multiple sub-graphs are set up on different processor types, and these sub-graphs can execute in Pipeline or Switch mode.

Setup. In the setup phase, after setting up the backend context, the *Graph Manager* creates and initializes all tensors associated with each edge of the sub-graph on their respective target processors. It generates a *Memory Manager* for each processor and tasks the manager to handle the tensors buffer within the corresponding backend context. Meanwhile, for each node, *ARM-CO-UP* leverages the function factory of the backend context, creating optimized functions tailored for individual nodes. This meticulous process generates a specific workload for each sub-graph. Further, the NPU backend creates the function for the NPU node based on the user-defined NPU type.

```
enum class NPUTypes{
    RockPi,
    Khadas,
};
const NPUTypes selectedNPU = NPUTypes::RockPi;
create_npu_function<NPU<selectedNPU>>(node);
```

ARM-CO-UP adds the execution tasks represented by node function input and output tensors to the workload. The input tensor's accessor handles the loading and pre-processing of the input data (image), while the output tensor accessor is responsible for post-processing and interpreting the output. *ARM-CO-UP* extends the workload by adding the `Receiver` and `Sender` tensor of the `Receiver` and `Sender` nodes, respectively. These extended objects facilitate the synchronization and transfer of data. The subsequent phase involves memory allocation for `Constant` tensors, which house essential parameters such as weights and biases. Once allocated, *ARM-CO-UP* begins the loading process for these trained static parameters. These static parameters require a one-time load

during the setup phase, preparing the system for inference across varied input images. Since *ARM-CO-UP* sets up sub-graphs in parallel, it reduces the loading time.

In parallel, *ARM-CO-UP* addresses the NPU segments of each sub-graph (extracted earlier in the pre-setup phase). Leveraging the distinctive naming convention, the pertinent NPU node — tasked with executing that specific segment — identifies and retrieves its associated segment. With the workloads for each sub-graph now defined and the model's static parameters duly loaded, *ARM-CO-UP* stands poised to execute inference on the provided input images.

Run the Inference. *ARM-CO-UP* framework loads the input data and executes the inference workload functions in the run step. In terms of execution, the graph management run component of the *ARM-CO-UP* includes the execution of inter-connection tensors involving the receiver and sender. Each sub-graph host, pinned to a specified CPU core, handles the sub-graph workload autonomously. This host thread activates the tensor accessors for input and receivers, ensuring data is seamlessly fetched from the primary input or dispatched from sender sub-graphs, as needed. *ARM-CO-UP* systematically schedules the workload functions of the sub-graph on the target processor once it has prepared all data elements. Subsequently, it engages the tensor accessor for all outputs and senders, facilitating the post-processing of output results and the data forwarding to the receiving sub-graphs to continue the inference. The run procedure extends to the execution of the receiver-tensor function at the beginning (in addition to the input tensor functions) and the sender-tensor function at the end of each sub-graph (in addition to the output-tensor function). The `Receiver` and `Sender` tensors transfer data between sub-graphs and synchronize their execution due to the dependencies. The sender-tensor holds the address of the receiver tensors to which it sends the data. Once the data is ready, the sender-tensor calls the `send_data` function of all associated receiver tensors and passes the data as an input argument. This function checks if the receiver can receive the data. If it is ready, then the sender-tensor transfers data to the receiver-tensor. If the receiver is unprepared, the sender-tensor places the data in the buffer of the receiver-tensor before returning the function.

On the other hand, the *Graph Manager* initiates the execution of a sub-graph by calling the functions of the `Input` and `Receiver` tensors. The `Receiver` tensor then calls the `receive_data` function, which checks if data is in the buffer. If data is available, the `Receiver` tensor fetches it; otherwise, the `Receiver` tensor sets the ready flag and waits for the sender to send the data. Though the execution of each sub-graph is managed by a separate host thread, harnessing a multi-threaded approach, these sub-graphs represent sequential segments of the complete model, necessitating consecutive execution. This structure precludes parallel execution of sub-graphs for a single input image. However, the design does permit simultaneous processing of successive input images, leveraging a pipeline structure.

By incorporating these enhancements, *ARM-CO-UP* enables efficient and collaborative utilization of various processor types, enhancing performance, energy efficiency, and flexibility.

6 VALIDATION

Determining the optimal mapping of every layer to the appropriate processor is crucial in achieving desired performance outcomes within the constraints of a given application. *ARM-CO-UP* offers a versatile set of features to navigate this massive search space effectively while enabling the fine-tuning of inference performance for specific targets. The key feature of the *ARM-CO-UP* is the ability to map layers to processors, allowing for desired allocation and optimization based on application objectives. The profiling functionality is the foundation for exploring the mapping search space in the *ARM-CO-UP* framework. In addition, *ARM-CO-UP* provides options for selecting the running mode (Pipeline or Switch) to facilitate cooperative processor utilization, adjusting processor frequencies at a granular level for power management, specifying the number of threads for CPUs, and designating the host CPU for GPU and NPU devices.

Table 3. Details for the experimental setup.

Experimental Setup	
Board	*Rock Pi N10*
SoC	*RK3399PRO*
CPU	*ARM big.Little*
Little CPU	Dual-core *ARM Cortex-A72+*
Little CPU L1 cache	48 KB/32 KB I/D cache
Little CPU L2 cache	1 MB
big CPU	Quad-core *ARM Cortex-A53*
big CPU L1 cache	32 KB/32 KB I/D cache
big CPU L2 cache	512 KB
GPU	Quad-core *ARM Mali-T860 MP4* GPU
GPU L2 cache	256 KB
Main NPU	*Rockchip* NPU
Interconnect	*CCI500*
Memory	LPDDR3 3 GB (NPU 1 GB + CPU 2 GB)
OS	*Android* 8.1
Framework	*ARM-CL* v18.03

Table 4. Extra-functional characteristics of *AlexNet* in Pipeline mode.

Layers Mapping	Host		Frequency (GHz)			Time (ms)	Energy (mJ)
	GPU (G)	NPU (N)	*Little* CPU (L)	*big* CPU (C)	GPU (G)		
GLNNNNNN	B	L	1.2	1.416	0.6	34.99	243.46
GNNNNNNB	B	L	0.408	1.608	0.6	40.26	226.39
GNNNNNNL	B	L	0.816	1.2	0.4	42.88	208.89
NNNNNNNB	B	L	0.6	1.008	0.2	47.21	191.55

6.1 Evaluation

We assess the performance and efficiency of the *ARM-CO-UP* framework on a real platform running popular CNNs. Table 3 details the *Rock Pi N10* embedded board used in this work. Our measurements inherently encapsulate all performance and power overheads. However, we make a deliberate effort to delineate these overheads in a dedicated section to provide a clearer understanding.

ARM-CO-UP framework seamlessly integrates the NPU with the CPUs and GPU, offering versatility across both Pipeline and Switch modes. In Pipeline mode, we demonstrate the integration of the non-quantized NPU with *Little* CPU, *big* CPU, and GPU to boost throughput and energy efficiency. In Switch mode, we highlight the integration of the *Little* CPU, *big* CPU, and GPU to enhance energy efficiency. *ARM-CO-UP* also supports quantized NPU in Pipeline and Switch modes, introducing potential accuracy trade-offs. However, our primary focus is not to explore all possible trade-offs but to showcase the framework's capabilities.

We aim to demonstrate the framework's efficacy in enhancing performance and power efficiency across various CNN models. We begin by leveraging Pipeline mode to boost throughput and energy efficiency, followed by latency and energy efficiency in Switch mode. Our evaluation also includes a comparative analysis with the *Pipe-it* [24] framework to provide context. Throughout our assessment, we showcase performance and energy efficiency results using the GA algorithm, emphasizing the framework's effectiveness. It's important to note that while we provide insights into the framework's capabilities, we do not explore all trade-offs comprehensively in this paper. Instead, we offer capabilities for researchers to analyze trade-offs in performance, energy efficiency, and accuracy, facilitating the discovery of optimal solutions.

6.1.1 Pipeline mode. In Pipeline mode, the *ARM-CO-UP* framework empowers users to map desired graph segments onto designated processors and execute them concurrently across successive frames. Users define the mapping, and the framework autonomously manages execution, including partitioning, NPU preparation, data transfer, and inter-processor conversion. Building upon this framework, we employ a multi-objective

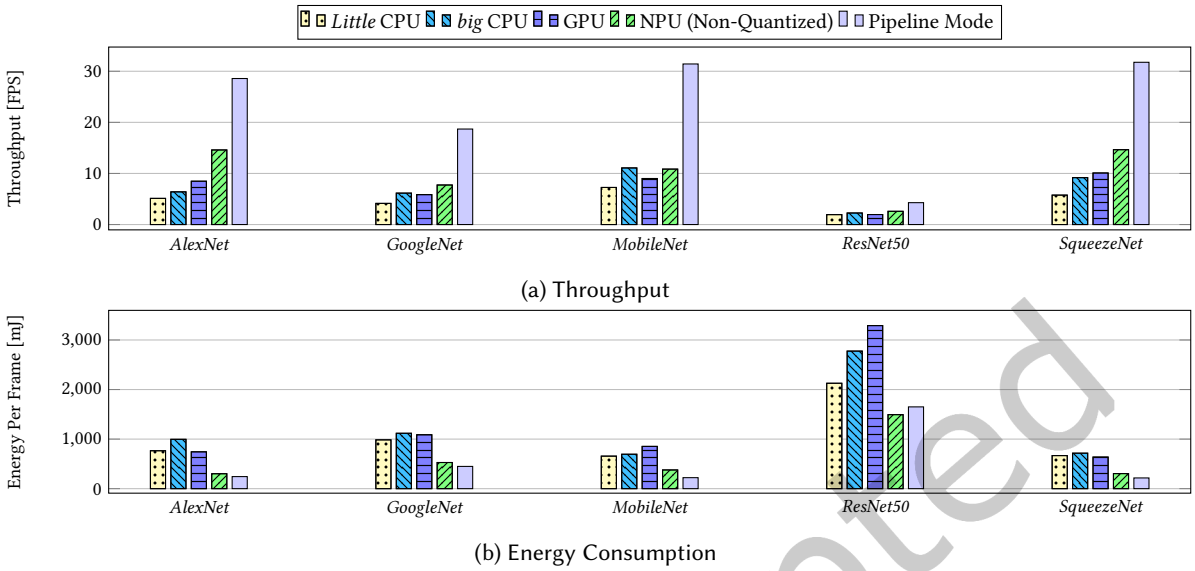


Fig. 13. Pipeline mode versus single-processor inference comparison for different CNNs.

Table 5. Comparison of top-1 accuracy and inference time between non-quantized and INT8-quantized models for different image classification CNNs running on NPU.

CNN	Top-1 Accuracy (%)		Inference Time (ms)	
	Non-Quantized	INT8-Quantized	Non-Quantized	INT8-Quantized
AlexNet	56.072	55.912 [-0.160]	139.8	29.7
GoogleNet	67.912	67.796 [-0.116]	129.2	26.7
MobileNet	68.362	67.354 [-1.008]	136.5	29.4
ResNet50	68.266	67.768 [-0.498]	181.0	38.8
SqueezeNet	56.984	56.592 [-0.392]	119.0	25.5

genetic algorithm *NSGA-II*) to explore the design space, varying mapping (partitioning and mapping) and power settings to identify near-optimal solutions for throughput and efficiency. The multi-objective genetic algorithm leverages the power and performance profile feature of *ARM-CO-UP* to evaluate a design space encoded within chromosomes. It commences with a random population of 100 design points, executing configurations on the platform using *ARM-CO-UP*. It then evaluates each point with measurements obtained through *ARM-CO-UP*'s power and performance profiling features. Subsequently, it selects efficient population points and generates offspring using mating operations.

Table 4 presents the pipeline results for the *AlexNet*, presenting layer-to-processor mappings (L, B, G and N for *Little* CPU, *big* CPU, GPU, and NPU, respectively), host selection for GPU and NPU devices, DVFS settings for each processor, and the resulting throughput and energy efficiency. We assess the Pipeline mode across five CNNs in Figure 13. Figures 13a and 13b illustrate the throughput and energy efficiency of the pipeline in these CNNs, along with the throughput of each processor. The framework explores the design space to discover near-optimal pipeline solutions within this expansive landscape. Despite the overheads, the Pipeline mode significantly enhances both throughput and energy efficiency.

We primarily present the pipeline results obtained from non-quantized models on the NPU. The framework supports the utilization of quantized models on the NPU in both pipeline and switch modes. Users can select

Table 6. Comparison of mean mAP and inference time between non-quantized and INT8-quantized YOLO-V3 model for object detection task running on NPU.

CNN	mAP (%)		Inference Time (ms)	
	Non-Quantized	INT8-Quantized	Non-Quantized	INT8-Quantized
YOLO-V3	68.780	64.709 [-4.070]	3328.1	438.1

Table 7. Processing time of *GoogleNet* layers on *Little* CPU, *Big* CPU, GPU, and best processor amongst them.

Layer	*Little* CPU Time (ms)	*Big* CPU Time (ms)	GPU Time (ms)	Minimum Time (ms)
0	57.6	35.0	40.2	35.0
1	34.5	23.9	31.2	23.9
2	12.9	13.3	10.2	10.2
3	24.1	28.6	23.3	23.3
4	9.0	10.8	7.8	7.8
5	10.1	12.9	8.3	8.3
6	11.0	13.6	10.0	10.0
7	11.8	14.5	10.6	10.6
8	16.3	20.5	14.5	14.5
9	7.5	8.0	7.1	7.1
10	10.8	11.7	10.1	10.1
Sum	205.4	192.8	173.1	160.7

Table 8. Switching overhead in the Switch mode.

Graph	*AlexNet*	*GoogleNet*	MobileNet	ResNet50	SqueezeNet
Best Mapping	GGGGGGGG	GGBBBBBBBBB	BBBBBBBBBBBBB	BBLLLLLLLLLLLLLL	GGGGGGGGLL
Inference Time	120.5 ms	161.5 ms	106.8 ms	381.7 ms	100.0 ms
Switching Time Overhead	0	0.8 ms (0.5%)	0	1.3 ms (0.3%)	0.6 ms (0.6%)
Inference Energy	740 mJ	1047 mJ	678 mJ	2215 mJ	618 mJ
Switching Energy Overhead	0	4 mJ (0.4%)	0	7 mJ (0.3%)	5 mJ (0.8%)

quantization for specific NPU partitions, enabling the NPU tools to perform quantization during the NPU preparation step. While quantization enhances run-time performance, it may incur a trade-off with a reduction in accuracy. We evaluate the accuracy and inference time for quantized and non-quantized models running on the NPU. Table 5 illustrates the top-1 accuracy and inference time for image classification using quantized and non-quantized models. Model quantization to INT8 occurs with the vendor-specific conversion tools available from *Radxa*. We assess accuracy using 50,000 images from the *ImageNet* validation set, while inference time is measured during model execution on the NPU using the *ARM-CO-UP* framework. Furthermore, Table 6 presents the mean Average Precision (mAP) and inference time for object detection using the *YOLO-V3* model. We assess accuracy here using the *COCO* validation dataset with 5,000 images.

6.1.2 Switch mode. We present latency results by comparing the inference performance with the best single processor for the entire CNN in Switch mode. Table 7 delineates the execution time of each layer of GoogleNet on both the big CPU and GPU, highlighting variability in layer performance across the processors. Some layers exhibit faster execution on the big CPU, while others perform better on the GPU. Moreover, Table 7 identifies the quickest processor for each layer. Switch mode necessitates a switch for executing each layer with the optimal processor, transitioning to the GPU from the third layer. The *sum* row of the last column represents the summation of layer execution times on their respective best processors, totalling 160.7 ms for GoogleNet.

Subsequently, we perform inference with the desired mapping using the run command option within the framework to assess the overall inference time and energy per frame, incorporating switching overhead. These outcomes encapsulate synchronization and transfer overheads during processor transitions. Table 8 delineates the overall inference time and energy per frame, alongside switching time and energy overhead. As depicted in

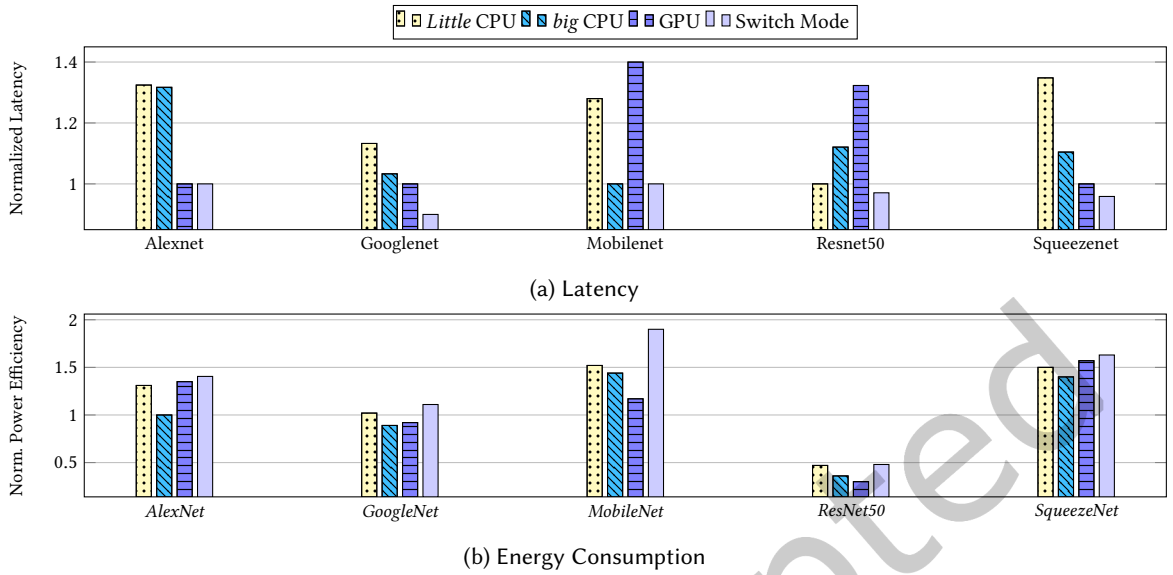


Fig. 14. Switch mode versus single-processor inference comparison for different CNNs

the table, the execution time for *GoogleNet* amounts to 161.5 ms, incurring a 0.8 ms switching overhead for data transfer from the second to the third layer on the GPU.

For *AlexNet* and *MobileNet*, the optimal mapping entails running all layers on the GPU and *big* CPU, respectively, obviating switching. Conversely, *SqueezeNet* exhibits minimal switching overhead compared to other networks. We attribute this observation to switching occurring in the final layers, which entail smaller data sizes for transmission.

Figure 14 compares the switching method and the best single processor inference. Figure 14a shows an overall latency improvement despite the synchronization and data transfer overheads. In addition to latency, we delved into the power efficiency of the Switch mode to achieve a target latency. Initially, we harnessed the profiling feature of the *ARM-CO-UP* framework to analyze the time and power consumption during layer execution and data transfer. We developed models to predict inference time and power consumption using the data. Subsequently, employing a GA, we navigated the design space to identify the near-optimal power-efficient (using DVFS) configuration for a target latency. Using time and power models within the GA framework significantly expedites the exploration of design points, ensuring thorough coverage. Figure 14b illustrates the power efficiency of the switch mode compared to the power efficiency attained when utilizing a single processor for inference. This analysis provides valuable insights into the efficacy of the Switch mode in enhancing power efficiency.

6.1.3 DVFS Settings. *ARM-CO-UP* integrates an internal power manager to oversee the voltage and frequency configurations for processors, supporting DVFS in both per-processor and per-layer modes. Through accessible APIs and command options, users can specify DVFS settings, which *ARM-CO-UP* implements at setup- and run-time for per-processor and per-layer modes, respectively. The per-processor DVFS feature allows users to explore settings across pipeline stages. At setup time, *ARM-CO-UP* configures frequency and voltage settings for the *Little* CPU, *big* CPU, and GPU. Leveraging this capability, we investigated the impact of DVFS on performance and energy consumption. Table 11 presents inference time and energy consumption per frame across various DVFS settings for the processors using a fixed layer mapping. The mapping and host selection remain fixed,

Table 9. The available DVFS levels for different processors.

index	*Little* CPU		*big* CPU		GPU	
	Frequency (MHz)	Voltage (mV)	Frequency (MHz)	Voltage (mV)	Frequency (MHz)	Voltage (mV)
1	408	800	408	800	200	800
2	600	800	600	800	300	800
3	816	850	816	825	400	825
4	1008	925	1008	875	600	925
5	1200	1000	1200	950	800	1100
6	1416	1125	1417	1025		
7			1608	1100		
8			1800	1200		

Table 10. The min and max DVFS delay for DVFS-capable processors when transitioning to higher (up) and lower (down) frequency levels.

Transition	*PE*	*Min Delay (µs)*	*Frequency*		*Max Delay (µs)*	*Frequency*	
			i	*i+1*		*i*	*i+1*
Up	L	296	0	1	4211	0	2
	B	193	0	1	3811	6	7
	G	657	0	1	4461	2	4
Down	L	109	4	3	193	3	0
	B	91	7	3	1413	4	1
	G	670	4	1	1464	4	2

Table 11. Extra-functional analysis of *AlexNet* running in Pipeline mode.

Layers mapping	*Host*		*Frequency (GHz)*			*Time (ms)*	*Energy (mJ)*
	GPU	NPU	*Little* CPU	*big* CPU	GPU		
GNNNNNNL	B	L	1.008	1.608	0.6	40.45	224.67
GNNNNNNL	B	L	0.6	1.608	0.6	40.5	224.42
GNNNNNNL	B	L	1.2	1.416	0.6	40.52	220.68
GNNNNNNL	B	L	0.816	1.416	0.6	40.57	220.22
GNNNNNNL	B	L	0.816	1.2	0.6	40.95	214.92
GNNNNNNL	B	L	1.2	1.416	0.4	41.14	212.99
GNNNNNNL	B	L	1.008	1.008	0.6	41.37	211.18
GNNNNNNL	B	L	0.6	1.008	0.6	41.54	210.43
GNNNNNNL	B	L	0.816	1.008	0.6	41.58	209.77
GNNNNNNL	B	L	0.6	1.2	0.4	42.8	209.39
GNNNNNNL	B	L	0.816	1.2	0.4	42.88	208.89

allowing for focused analysis of DVFS effects on the GPU, Little CPU (first and third stages), and big CPU (second stage) processors.

The effect of DVFS on each pipeline stage hinges on whether the stage serves as a bottleneck. Decreasing the frequency could reduce energy consumption per frame when a stage is not a bottleneck. Conversely, if a stage acts as a bottleneck, decreasing the frequency may increase energy consumption due to prolonged inference times per frame. This nuanced understanding underscores the importance of considering pipeline dynamics and workload characteristics when optimizing DVFS settings within the *ARM-CO-UP* framework. Users can effectively improve energy consumption and performance by discerning bottleneck stages and adjusting DVFS settings.

In per-processor mode, DVFS incurs no overhead on inference as settings are configured once at setup time and remain constant throughout run-time. This stability ensures efficient management of DVFS without run-time adjustments, thus streamlining the optimization process within the *ARM-CO-UP* framework. The per-layer DVFS

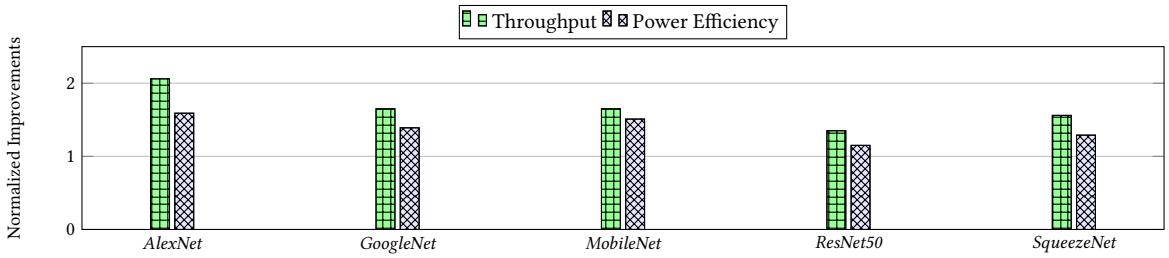


Fig. 15. Throughput and power efficiency of *ARM-CO-UP* normalized to *Pipe-it* [24] for different CNNs.

feature enables dynamic voltage and frequency adjustments during run-time and is especially beneficial for Switch mode operations. At the start of each layer, the framework triggers the requested DVFS settings for the underlying processor in a non-blocking approach. Experimental observations have revealed a slight delay between the triggering of the DVFS settings and the actual change in the hardware, as detailed in Table 10 for transitions to higher or lower frequency settings in the GPU and CPUs.

This delay does not disrupt the inference process or incur additional waiting overhead due to the non-blocking procedure. Inference continues seamlessly, with the computation smoothly transitioning to the requested DVFS settings as they apply gradually to the hardware. This mechanism ensures uninterrupted operation and facilitates the examination of DVFS effects on inference.

6.1.4 Comparative Evaluation. Figure 15 shows the CNN inference throughput and power efficiency for *ARM-CO-UP* compared with *Pipe-it* [24]. *ARM-CO-UP* and *Pipe-it* work on top of *ARM-CL*. However, *Pipe-it* supports only the *Little* and *big* CPU. Therefore, we limit the results in Figure 15 to only *ARM big.Little* asymmetric multi-core CPU in *Rock PI N10*. Figure 15 shows *ARM-CO-UP* provides, on average, 1.67x and 1.49x higher throughput and power efficiency over *Pipe-it*, respectively. The throughput and power efficiency gains originate from more efficient sub-graph-based implementation in *ARM-CO-UP*, which induces much less overhead than the thread migration-based implementation in *Pipe-it*.

6.2 Case Study

We leverage the capabilities of the *ARM-CO-UP* framework to present intuitive analytical results. Our case study analysis focuses on the timing and power consumption of the *MobileNet* CNN, utilizing the profiling feature of the framework. The analysis applies to other CNNs in a similar manner. Initially, we delve into the power efficiency (Frames Per Second per Watt - FPS/Watt) of individual layers for exploring various DVFS levels through the *ARM-CO-UP* power manager. Subsequently, we conduct a comparative analysis of the execution time, energy consumption, and power efficiency across different processors for each layer. This comparative study sheds light on the performance and efficiency of processors concerning diverse layers within the *MobileNet* architecture. Users can utilize these results to facilitate the identification of the optimal inference configuration based on the specific objectives and constraints of their application.

Figure 16 illustrates the power efficiency of *MobileNet* layers across various frequency levels for the *Little* CPU, *big* CPU, GPU, and NPU. Notably, for the NPU, the exploration is conducted concerning the host (*big* CPU) frequency, given that the NPU lacks DVFS level adjustment capabilities (see Figure 16d). This experiment leverages the profiling feature at the layer granularity, coupled with the power manager, to measure the time and power consumption of layers under different DVFS settings. Subsequently, we calculate the power efficiency of each layer.

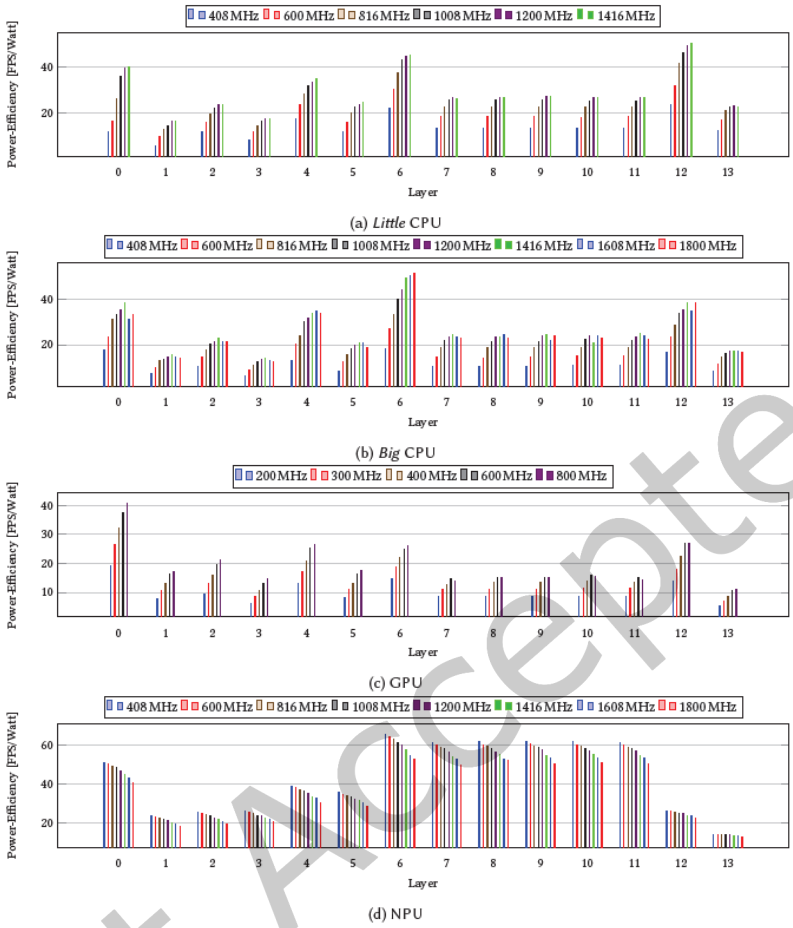


Fig. 16. Power efficiency vs. Frequency for *MobileNet* layers.

The results reveal a general trend across the *Little* CPU (Figure 16a), *big* CPU (Figure 16b), and GPU (Figure 16c), wherein increasing the frequency enhances the power efficiency of layers. We attribute this phenomenon to the reduction in time, which outweighs the increase in power, resulting in higher power efficiency and reduced final energy consumption. However, the dynamics differ for the NPU (Figure 16d). Given that the explored frequency is not that of the NPU itself but rather the host CPU (*big* CPU), the results indicate that, when running layers on the NPU, the optimal choice for maximizing power efficiency is to minimize the frequency of the host CPU.

In the context of the NPU, it is crucial to delineate the distinct role of the host CPU, particularly the Big CPU cluster, which orchestrates the loading of input data into the NPU and initiates its execution. It is important to note that altering the frequency and voltage settings of the host CPU does not directly impact the execution time of a layer on the NPU. Instead, any increase in the CPU frequency and voltage settings amplifies the HMPSoC's power consumption, as the CPU voltage heightens during NPU execution, consequently increasing power consumption and compromising overall power efficiency.

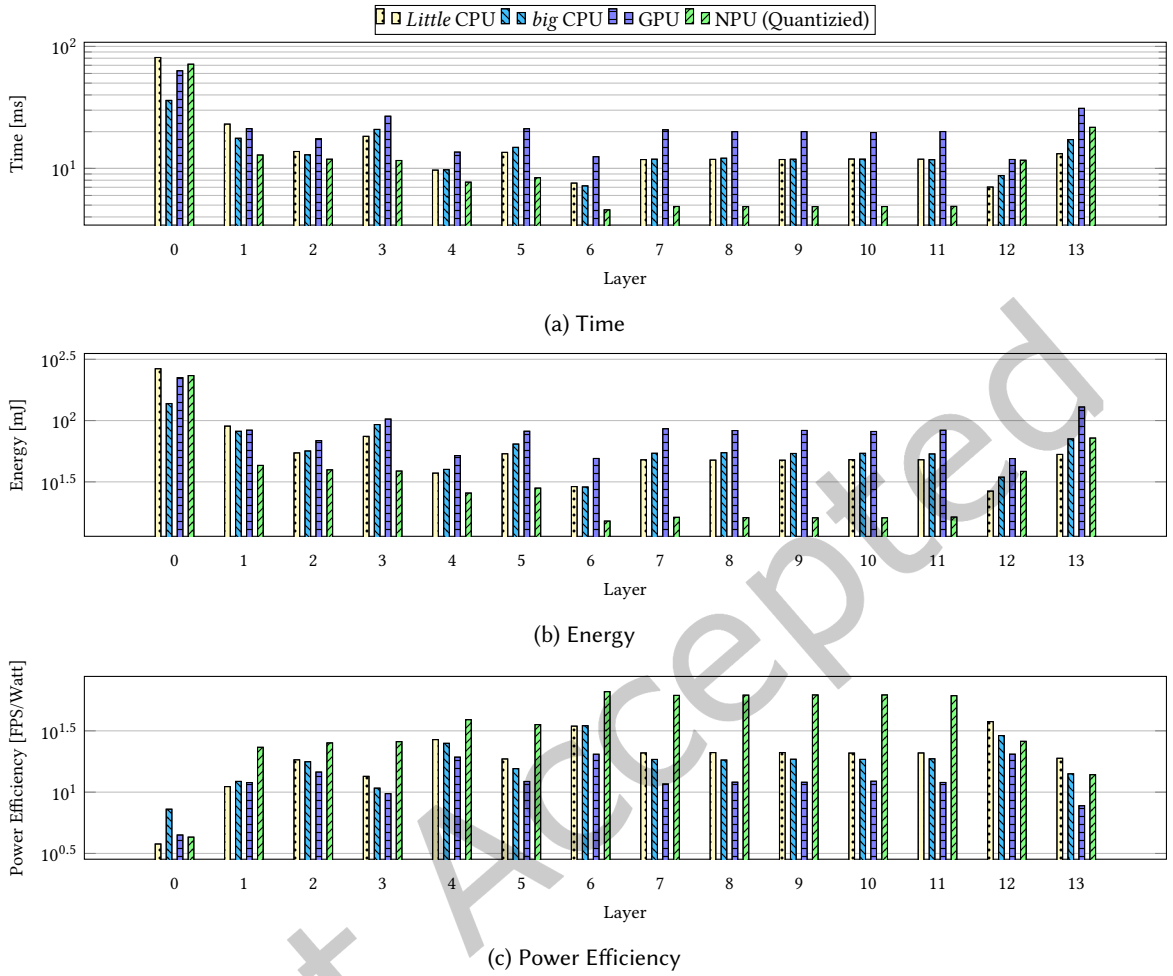


Fig. 17. Different characteristics of *MobileNet* layers on different HMPSoCs processors.

In the subsequent assessment, we delve into the efficiency of the *Little* CPU, *big* CPU, GPU, and NPU across different layers of the *MobileNet*, with the results of this comparative analysis presented in Figure 17. For this analysis, 60 random images were drawn from the *ImageNet* dataset for evaluation and profiling by *ARM-CO-UP*. We then averaged the results to generate the graphs. We use each image for inference individually, i.e. with the batch size setting as one. This analysis encompasses the examination of execution time (Figure 17a), energy consumption (Figure 17b), and power efficiency (Figure 17c). The examination of execution time depicted in Figure 17a reveals that, in most cases, the NPU exhibits the highest performance, significantly outperforming other processors. The first layer is an exception, where the NPU's superiority is mitigated by the substantial contribution of loading and preparing input data. The NPU effectiveness is reduced by the overhead of transferring data, especially for initial layers with large data sizes. Furthermore, the results indicate comparable performance between the CPU and GPU, with the CPU outperforming in some layers and the GPU excelling in others. This

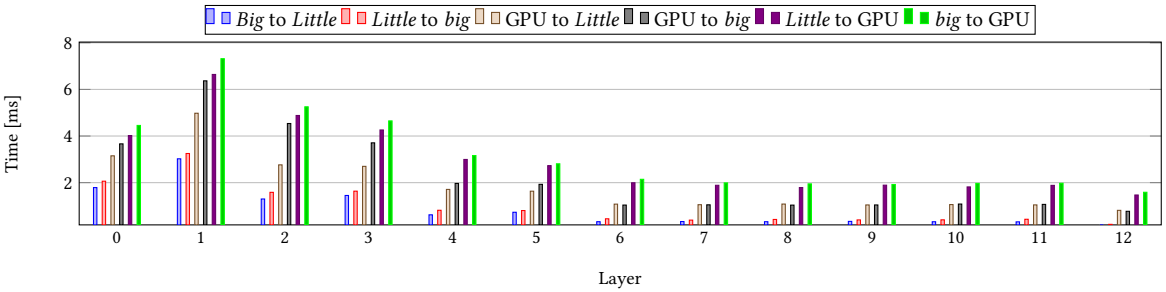


Fig. 18. Inter-processor transfer times for *MobileNet* layers

nuanced performance distinction underscores the intricacies of layer-specific computational requirements and the adaptability of the processors to varying tasks within the *MobileNet* architecture.

The results for energy consumption (Figure 17b) and power efficiency (Figure 17c) underscore that the NPU not only surpasses the CPU and GPU in terms of performance (execution time) but also exhibits optimal energy consumption and power efficiency. This observation implies that the NPU is the most effective processor for performance and power efficiency. However, when the target is throughput, using the Pipeline mode, incorporating both the CPU and GPU contributes to increased throughput compared to utilizing only the NPU. A trade-off exists between accuracy and performance/power efficiency, even in the Pipeline mode. This observation is because NPU computations are executed in a quantized version, leading to a drop in accuracy. Striking a balance between these factors becomes pivotal in optimizing the overall system performance based on specific application requirements and objectives.

Figure 18 investigates the transfer time between different source and destination processors for *MobileNet* layers. These timings serve as a measure of overhead incurred during the switching process between different processors. Notably, the transfer time is contingent upon the data transfer size. The data transfer involves direct movement, conducted in switch mode (without buffering), from the output of one layer in the source processor to the input of the next layer in the target processor. This process encompasses synchronization overhead, accentuating the communication overhead during the transition between successive layers. Figure 18 illustrates that this transfer time tends to decrease as we progress deeper into the network. This observation stems from the fact initial layers typically involve larger data sizes for transfer, while deeper layers exhibit a decrease in data size. Observations suggest that transitioning to other processors in deeper layers, especially for those with higher switching costs, is more advantageous.

Furthermore, the transfer time for *MobileNet* ranges approximately from one to five milliseconds, as depicted in Figure 18. This range, when compared to the order of execution times for layers on processors (Figure 17a), indicates that tolerating overhead for switching between processors every couple of layers could be beneficial. The relatively low overhead associated with switching stems from the integrated nature of these processors on a single chip. This insight into transfer times provides valuable guidance for optimizing the layer distribution across processors, accounting for the trade-off between computational efficiency and the cost of inter-processor communication.

A comprehensive examination of the execution costs at various layers enables seamless integration with design space exploration algorithms. This examination underscores the adaptability of the framework, empowering users to customize the inference process according to their specific application requirements through optimization and search algorithms.

7 CONCLUSION AND FUTURE WORK

We have presented *ARM-CO-UP*, an advanced framework that addresses the limitations of existing inference frameworks for edge devices. By cooperatively utilizing *ARM* asymmetric multi-core CPUs, *GPU*, and NPU, *ARM-CO-UP* achieves optimal inference performance and energy efficiency. *ARM-CO-UP* overcomes drawbacks of state-of-the-art, such as no support for accelerators and CNNs with complex models. It introduces a refined implementation within the core of *ARM-CL*, extending it extensively. It supports fine-grained partitioning at the layer level, allows layer-level DVFS, automatically handles branches within partitions, and includes a generic NPU node for effortless integration of desired accelerators. These features make *ARM-CO-UP* a promising solution for deep learning workloads on resource-limited embedded devices.

REFERENCES

[1] Ehsan Aghapour, Dolly Sapra, Andy Pimentel, and Anuj Pathania. 2022. CPU-GPU Layer-Switched Low Latency CNN Inference. In *2022 25th Euromicro Conference on Digital System Design (DSD)*. 324–331. https://doi.org/10.1109/DSD57027.2022.00051

[2] Ehsan Aghapour, Dolly Sapra, Andy D. Pimentel, and Anuj Pathania. 2023. PELSI: Power-Efficient Layer-Switched Inference. In *2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 12–17. https://doi.org/10.1109/RTCSA58653.2023.00011

[3] Hung-Yang Chang, Seyyed Hasan Mozafari, Cheng Chen, James J. Clark, Brett H. Meyer, and Warren J. Gross. 2023. PipeBERT: High-throughput BERT Inference for ARM Big.LITTLE Multi-core Processors. *Journal of Signal Processing Systems* 95, 7 (01 Jul 2023), 877–894. https://doi.org/10.1007/s11265-022-01814-y

[4] Jessie Y. C. Chen and Jennifer E. Thropp. 2007. Review of Low Frame Rate Effects on Human Performance. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 37, 6 (2007), 1063–1076. https://doi.org/10.1109/TSMCA.2007.904779

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799* 11, 20 (2018).

[6] Xiaotian Guo, Andy D. Pimentel, and Todor Stefanov. 2023. Automated Exploration and Implementation of Distributed CNN Inference at the Edge. *IEEE Internet of Things Journal* 10, 7 (2023), 5843–5858. https://doi.org/10.1109/JIOT.2023.3237572

[7] Jussi Hanhirova, Teemu Kämäräinen, Sipi Seppälä, Matti Siekkinen, Vesa Hirvisalo, and Antti Ylä-Jääski. 2018. Latency and Throughput Characterization of Convolutional Neural Networks for Mobile Computer Vision. In *Proceedings of the 9th ACM Multimedia Systems Conference* (Amsterdam, Netherlands) *(MMSys '18)*. Association for Computing Machinery, New York, NY, USA, 204–215. https://doi.org/10.1145/3204949.3204975

[8] Alexander Hoffman, Anuj Pathania, Philipp H. Kindt, Samarjit Chakraborty, and Tulika Mitra. 2020. BrezeFlow: Unified Debugger for Android CPU Power Governors and Schedulers on Edge Devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218542

[9] Eunjin Jeong, Jangryul Kim, and Soonhoi Ha. 2022. TensorRT-Based Framework and Optimization Methodology for Deep Learning Inference on Jetson Boards. *ACM Trans. Embed. Comput. Syst.* 21, 5, Article 51 (oct 2022), 26 pages. https://doi.org/10.1145/3508391

[10] EunJin Jeong, Jangryul Kim, Samnieng Tan, Jaeseong Lee, and Soonhoi Ha. 2022. Deep Learning Inference Parallelization on Heterogeneous Processors With TensorRT. *IEEE Embedded Systems Letters* 14, 1 (2022), 15–18. https://doi.org/10.1109/LES.2021.3087707

[11] Andreas Karatzas and Iraklis Anagnostopoulos. 2023. OmniBoost: Boosting Throughput of Heterogeneous Embedded Devices under Multi-DNN Workload. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[12] Andreas Karatzas and Iraklis Anagnostopoulos. 2023. OmniBoost: Boosting Throughput of Heterogeneous Embedded Devices under Multi-DNN Workload. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247989

[13] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. 2004. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture* (München, Germany) *(ISCA '04)*. IEEE Computer Society, USA, 64.

[14] Tyrone Tai-On Kwok and Yu-Kwong Kwok. 2008. On the design, control, and use of a reconfigurable heterogeneous multi-core system-on-a-chip. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–11. https://doi.org/10.1109/IPDPS.2008.4536165

[15] Zhaoying Li, Dhananjaya Wijerathne, Xianzhang Chen, Anuj Pathania, and Tulika Mitra. 2022. ChordMap: Automated Mapping of Streaming Applications Onto CGRA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 2 (2022), 306–319. https://doi.org/10.1109/TCAD.2021.3058313

[16] Xiangzhong Luo, Di Liu, Shuo Huai, Hao Kong, Hui Chen, and Weichen Liu. 2021. Designing efficient DNNs via hardware-aware neural architecture search and beyond. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 6 (2021), 1799–1812.

[17] Svetlana Minakova, Dolly Sapra, Todor Stefanov, and Andy D Pimentel. 2022. Scenario based run-time switching for adaptive CNN-based applications at the edge. *ACM Transactions on Embedded Computing Systems (TECS)* 21, 2 (2022).

[18] Svetlana Minakova, Erqian Tang, and Todor Stefanov. 2020. Combining Task- and Data-Level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Alex Orailoglu, Matthias Jung, and Marc Reichenbach (Eds.). Springer International Publishing, 18–35.

[19] Dolly Sapra and Andy D Pimentel. 2020. Constrained evolutionary piecemeal training to design convolutional neural networks. In *Trends in Artificial Intelligence Theory and Applications. Artificial Intelligence Practices: 33rd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2020, Kitakyushu, Japan, September 22-25, 2020, Proceedings 33*. Springer, 709–721.

[20] Wonik Seo, Sanghoon Cha, Yeonjae Kim, Jaehyuk Huh, and Jongse Park. 2021. SLO-aware inference scheduler for heterogeneous processors in edge platforms. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 4 (2021), 1–26.

[21] Mingwen Shao, Junhui Dai, Jiandong Kuang, and Deyu Meng. 2021. A dynamic CNN pruning method based on matrix similarity. *Signal, Image and Video Processing* 15, 2 (2021), 381–389.

[22] Junzhong Shen, You Huang, Zelong Wang, Yuran Qiao, Mei Wen, and Chunyuan Zhang. 2018. Towards a Uniform Template-Based Architecture for Accelerating 2D and 3D CNNs on FPGA. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 97–106. https://doi.org/10.1145/3174243.3174257

[23] Dawei Sun, Shaoshan Liu, and Jean-Luc Gaudiot. 2017. Enabling embedded inference engine with arm compute library: A case study. *arXiv preprint arXiv:1704.03751* (2017).

[24] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. 2020. High-Throughput CNN Inference on Embedded ARM Big.LITTLE Multicore Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2254–2267. https://doi.org/10.1109/TCAD.2019.2944584

[25] Siqi Wang, Anuj Pathania, and Tulika Mitra. 2020. Neural Network Inference on Mobile SoCs. *IEEE Design & Test* 37, 5 (2020), 50–57. https://doi.org/10.1109/MDAT.2020.2968258

[26] Dhananjaya Wijerathne, Zhaoying Li, Anuj Pathania, Tulika Mitra, and Lothar Thiele. 2022. HiMap: Fast and Scalable High-Quality Mapping on CGRA via Hierarchical Abstraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 10 (2022), 3290–3303. https://doi.org/10.1109/TCAD.2021.3132551

[27] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 331–344. https://doi.org/10.1109/HPCA.2019.00048

[28] Di Wu, Yu Zhang, Xijie Jia, Lu Tian, Tianping Li, Lingzhi Sui, Dongliang Xie, and Yi Shan. 2019. A High-Performance CNN Processor Based on FPGA for MobileNets. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 136–143. https://doi.org/10.1109/FPL.2019.00030

[29] Hsin-I Wu, Da-Yi Guo, Hsu-Hsun Chin, and Ren-Song Tsay. 2020. 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)*. 46–49. https://doi.org/10.1109/AICAS48895.2020.9073977

[30] Sean Young, Zhe Wang, David Taubman, and Bernd Girod. 2021. Transform Quantization for CNN Compression. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).