Hetero^{2}Pipe: Pipelining Multi-DNN Inference on Heterogeneous Mobile Processors under Co-Execution Slowdown

***Abstract*—The emerging multi-modal applications exemplified by multi-DNN inference have renewed interests for mobile intelligence. The goal is to utilize heterogeneous processors on-board to maximize throughput and resource utilization. Among a variety of options, building model-paralleled pipelines across different processors is a promising way. However, the existing efforts either focus on optimizing homogeneous DNN executions or simply ignore co-execution slowdown on the shared memory bus. Based on extensive empirical studies and insights with various degrees of resource contention, in this work, we introduce Hetero^{2}Pipe, a two-step pipeline planner based on dynamic programming, and contention-mitigated pipeline bubble minimization to make the problem tractable within manageable search space. The extensive evaluation across three commercial SoCs demonstrates 2-8× speedup compared to the state-of-the-art schemes.**

***Keywords*-Pipeline parallelization; emerging multi-DNN inference; co-execution memory interference; consumer mobile devices.**

I. Introduction

Recently, we have witnessed tremendous applications of on-device intelligence, that drives performance beyond the real-time processing speed for mobile devices. For example, inference of MobileNetV2 on Snapdragon 778G at 76 FPS, ResNet50 (FP16) at 30 FPS on CPUs, and over 100 FPS for ResNet50 on the Kirin 990 Neural Processing Unit (NPU). On the other hand, the emerging multi-modal applications are reshaping system design with a transition from the canonical *single-DNN inference* towards *multi-DNN inference* for different downstream tasks and applications. For example, a scene understanding app could be comprised of YOLOv4 for object detection [1], FaceNet [2], Age/GenderNet [3] for facial, age and gender recognition and ViT-GPT2 for scene-to-text captioning [4]. Although heterogeneous processors such as embedded GPU and NPU create new opportunities to share the workloads, the existing computing paradigm on mobile devices is still CPU-centric with serial execution and resource under-utilization [5].

Problem. Pipeline parallelism is a typical way to improve resource utilization, as evidenced by its recent success in model-paralleled training of large language models [6]–[8], in which the model is partitioned onto different GPUs to overcome the memory limit and training data are fed in micro-batches to form a pipeline to maximize resource utilization. For mobile devices, we are curious to answer a similar question:

§*"Given an array of heterogeneous mobile processors, can we migrate the paradigm of model-paralleled training from cloud GPUs to support multi-DNN inference on mobile devices? If yes, what adaptations are needed?"*

Challenges. Although multi-DNN inference is free from the long-range dependencies of backpropagation in model-paralleled training (i.e., inference does not need additional memory space for gradient storage like training), we are still facing a cohort of new challenges with multi-faceted heterogeneity across the micro-architecture as well as the algorithmic levels from *processors* and *models* (hence the name Hetero^2). Specifically, mobile processors exhibit distinct processing power, thermal and throttling behaviors with diverse programmability and operator support from the CPU, GPU and specialized processors such as NPU or TPU. For programmability, most mobile inferences are executed on the CPU cores, but subject to high throttling [9]; embedded GPUs such as ARM Mali and Qualcomm Adreno are accelerated by OpenCL, with fewer GPU cores and optimization compared to Nvidia GPUs powered by CUDA; NPU is the ideal processor, but it only supports very limited number of operators – any model with an unsupported operator would have to fall back to the CPU, which creates additional overhead of memory copy.

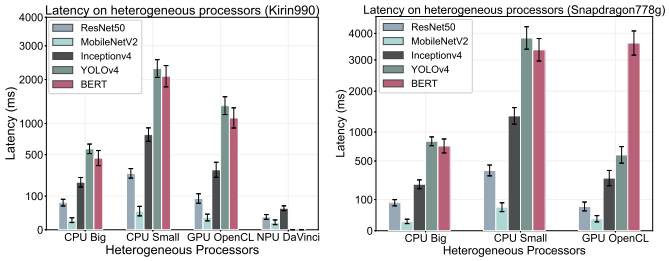


Fig. 1: Processing latency of different models on heterogeneous processors. For mobile SoCs, the Big CPU cores are generally on par with OpenCL GPU, while the Small cores pose high performance degradation. NPU has the fastest speed but very limited support for a diverse set of operators and an error is reported due to unsupported operators in the MNN framework for both YOLOv4 and BERT [10].

More importantly, unlike GPU clusters with high-bandwidth interconnections (NVLink over 200 Gbps), mobile SoCs couple computing units on the same die with shared bus and memory controller. Energy efficiency also demands low bandwidth designs with active memory frequency throttling based on workload intensity. These have effectively reduced the memory bandwidth below 20 Gbps [11]. Different from cloud GPUs, co-execution of inference on heterogeneous processors simultaneously would lead to competition for shared resources on the memory bus with various degrees of slowdown [12], [13]. For different models, intermediate dimensions directly affect data locality and hardware affinity [14], that vary significantly between different generations of model architectures such as Convolutional and Transformer networks as shown in Fig. 1. For example, while small convolutional kernels have better computational efficiency and data locality, the multi-head attention and quadratic computational complexity in Transformers necessitate large matrix multiplications that often exceed the L2 cache size of mobile CPUs, thereby causing high memory access, processing latency

and energy overhead. Such heterogeneity makes it difficult to find an appropriate mapping of workloads to different processors.

Hence, the response to our previous query remains to further tackle a series of intertwined modeling and system design questions of how to split each model into pipeline stages and map them to heterogeneous processors optimally? How to efficiently model resource contention under manageable profiling efforts? How to balance workloads across different processors under potential memory interference? The main inefficiency of pipeline execution stems from unbalanced loads between different stages because throughput is always bottlenecked by the slowest stage. Though there is a long list of works on model-paralleled training in cloud GPUs [6]–[8], [15], their formulation and solution are generally limited to homogeneous GPUs. In contrast, mobile devices exhibit unique challenges of memory interference when workloads are co-executed on different processors. This diminishes the optimization effort of pipeline schedules – in fact, very few of the existing works have actually considered planning under co-execution slowdown [12]–[14], especially on mobile devices. As a result, to pipeline heterogeneous processors, the resonance between resource under-utilization and runtime contention poses a dilemma: increasing the utilization often exacerbates the contention problem and load unevenness across different stages.

Solution. This paper proposes a new approach, called Hetero^{2}Pipe, that organizes multi-DNN requests on heterogeneous processors with a contention mitigation strategy. Upon realizing the fact that a single-step problem formulation in most of the previous works such as [6], [7], [16] cannot fully capture the dual heterogeneity in our system, we decouple pipeline planning into a more tractable two-step optimization problem along the *horizontal* and *vertical* directions of the processing pipeline. First, the horizontal model slicing is optimally solved by dynamic programming. Then we propose a work stealing strategy to align execution time across different stages in order to minimize the pipeline bubbles along the vertical direction. We also supplement the vertical optimization with a new *contention-mitigation* strategy based on the polynomial-time Linear Assignment Problem. The main contributions of this paper are summarized as follows:

- ✧ **Motivation**: We conduct extensive empirical studies to analyze the impact the dual heterogeneity of processors and network models, and resource contention on shared memory bus through performance counters. We discover counter-intuitive phenomenons that some lightweight models are actually memory-bound, thereby leading to high memory interference to concurrent executions. Based on these observations and measurements, we characterize such model-specific contention footprints via an effective regression model, without external efforts to profile a large number of co-execution combinations.
- ✧ **Methodology**: We propose a two-step pipeline planning approach to minimize the pipeline bubbles. To our best knowledge, this is the first work that considers co-execution slowdown in pipeline planning for resource-constrained edge devices. Our strategy incorporates a contention window to re-arrange incoming request at the minimum displacement cost. It transforms the complex contention mitigation problem into the classic Linear Assignment Problem with polynomial-time solutions and employs working stealing between different pipeline stages to minimize pipeline bubbles and improve resource utilization.
- ✧ **Evaluation**: We provide extensive evaluations on three commodity SoCs of Snapdragon 778G, 870 and Kirin 990 with large model combinations from the earliest CNN models, YOLOv4 object detectors [1] to the latest transformer architectures such as BERT [17] and ViT [18]. The results demonstrate 4-8$\times$ and 2-3$\times$ speedup compared to vanilla MNN [10] and Pipe-it [19]. Confronting the competitive SOTA scheme of Band [20], our strategy also achieves additional 5% gain due to extra pipelining optimizations.

II. Background and Related Works

A. Heterogeneous Mobile Computing

ARM Big.LITTLE is the de-facto standard to achieve energy-efficiency in mobile devices. However, with the increasing computational demand and limited thermal-area budget, mobile CPUs are no longer capable of handling the variety of on-device computational loads alone. Thus, considerable efforts from the vendors are devoted to strengthening the computational power with heterogeneous processors such as embedded GPU and NPU, with a growing number of works exploring the design space on heterogeneous processors. Pipe-it leverages the Big and Small CPU cores for fine-grained partition of the convolutional models with a flexible pipeline of different CPU cores [19]. However, as validated by this work, an in-cluster partition of the CPU cores leads to evictions and conflicting cache misses down the cache hierarchy. MASA proposes a technique to reduce the peak memory footprint and page faults while executing multiple DNN inference [21]. Blasnet supports real-time DNN model inference on CPU-GPU platforms through block-level model optimization and scheduling [22], but lacks a pipelining strategy to handle multi-DNN requests.

Another acceleration strategy is called intra-operator partition. μLayer develops a cooperative strategy leveraging the mobile CPU-GPU to partition the DNN in a channel-wise manner [23]. Intra-operator partition has been also implemented by [5], [24] on different granularities. For example, NN-Stretch proposes branch-parallelism that transforms a network into multiple paralleled branches on different processors [5]. However, the intermediate results from different processors are deemed to be merged with additional overhead of significant communication/memory copy per split. The closest work to ours is Band [20], which takes advantage of the superiority of the NPU and rolls back unsupported operators to the CPU or GPU. However, it only involves impromptu subgraph-processor mapping with an unoptimized strategy without effective pipeline planning.

B. Pipeline Planning

Model partition is a common strategy to distribute workloads on various computing units. A handful of previous efforts have explored DNN partitioning with Dynamic Programming [6]–[8], [16], [25], Branch and Bound [26] and other heuristics/solvers [19], [27]–[30], that a majority of them focus on building an efficient pipeline for model-paralleled training [6]–[8], [27], [28]. Gillis considers partitioning a large model

TABLE I: State-of-the-art methods for on-device inference

Related Work	Processors	multi-DNN	DNN Hetero.	Pipeline	Contention	Algorithm
Pipe-it [19]	CPU	✓	×	✓	×	Local Search
MASA [21]	CPU	✓	✓	×	×	BinPacking
EdgePipe [25]	CPU	✓	×	✓	×	DP
Gillis [16]	CPU	✓	×	✓	×	DP
B&B Parition [26]	CPU, GPU	×	×	×	×	Branch&Bound
μLayer [23]	CPU, GPU	×	×	×	×	—
TVMPipe [30]	CPU, GPU	✓	×	✓	×	Exhaustive Search
PICO [31]	CPU	✓	×	✓	×	DP
DART [32]	CPU, GPU	✓	×	✓	×	DP
BlasNet [22]	CPU, GPU	✓	×	×	×	DARTS
Band [20]	CPU, GPU, NPU	✓	✓	×	×	Greedy
Hetero^2 Pipe (Ours)	**CPU, GPU, NPU**	✓	✓	✓	✓	**DP+Work Stealing**

across multiple serverless functions and reduces the memory footprint for each function [16]. EdgePipe and PICO propose distributed frameworks on multiple heterogeneous devices with dynamic programming-based optimal partitioning strategy [25], [31]. A simple two-stage pipeline is considered in [30] that the optimal solution is found by searching over all possible device orders. In addition, a genetic algorithm is proposed to schedule the task graph for maximal throughput in [29]. However, these works only focus on optimizing inference requests with homogeneous model structures. DART leverages data parallelism for CPU/GPU platforms [32], but co-execution contention is not considered and the implementation is only based on homogeneous models as well. In this work, we consider a more difficult and realistic scenario of multiple heterogeneous model requests on different processors with co-execution slowdown and bubble minimization, which extends towards a much larger search space that has not been studied before. A detailed comparison is available in Table I.

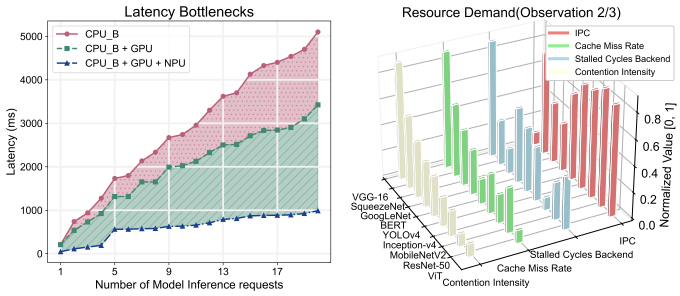
III. Resource Contention on Heterogeneous SoC

We first motivate our study by showcasing the benefits of utilizing the heterogeneous processors in Fig. 2(a). Obviously, for multi-DNN inferences, canonical implementations with serial processing on the Big CPU cores is subject to large queuing delay and introducing heterogeneous processing power alleviates the performance bottleneck. However, co-execution leads to resource competition and the dependencies on model combinations pose unique contention profiles.

Our initial finding is that the interference between CPU-GPU is much higher than CPU-NPU or GPU-NPU. For example, co-executing YOLOv4 and BERT results in 18%, 21% slowdown on CPU-GPU, while only 3%, 4.5% on CPU-NPU and 2%, 2.3% on GPU-NPU. This is possibly due to specialized design of NPU and dedicated memory path, which is less prone to resource contention. For CPU and GPU co-execution, the slowdown ratio is summarized below.

Observation 1 (Slowdown Consistency on CPU/GPU) The slowdown ratios across the CPU and GPU are generally consistent, i.e., equal-priority execution suffers from identical slowdown across the CPU-GPU. In other words, for different co-execution model pairs, it is less likely to have a large slowdown on the CPU but little on the GPU, or vice versa.

In fact, this is guaranteed by the fairness-aware scheduling policies in most commercial SoCs, and the memory controller often prioritizes requests with higher row-hit to maximize the total bandwidth [12]. As a result, when such a high row-buffer hit rate is no longer sustained under contention, a slowdown is observed on both CPU-GPU even though the peak memory



(a) (b)

Fig. 2: Empirical results on Kirin990 SoCs: (a) The queuing delay accumulates with serial execution on the CPU big cores (CPU_B) and bringing heterogeneous processors reduce the performance bottlenecks substantially; (b) Resource demands of executing different models (*Observation 2/3*). y-axis is ranked according to the *contention intensity*, explained in Eq. (1).

bandwidth is not reached. Thus, it is sufficient to measure the resource demands from solo executions as a proxy to indicate co-execution slowdown [13]. For DNNs, in addition to compute intensities, resource contention originates from the inner structures such as tensor dimensions, operators and precedence relations, and we describe the impact from these latent factors as explained below.

TABLE II: Slowdown comparison of SqueezeNet and ViT

Model	Processor	Solo-Exec Time(ms)	Co-Exec Time(ms)	Slowdown
SqueezeNet	CPU_B	13.46	17.02	26.43%
BERT	GPU	1109.23	1233.36	11.22%
BERT	CPU_B	553.91	670.31	21.01%
SqueezeNet	GPU	18.83	21.12	12.16%
ViT	CPU_B	453.37	504.56	11.29%
BERT	GPU	1109.23	1171.39	5.60%
BERT	CPU_B	553.91	613.57	10.77%
ViT	GPU	1474.53	1612.87	9.38%

We read the Processor Monitor Unit (PMU) of `perf event` from the CPU to indicate the interference on the GPU [1]. In particular, we examine `Instructions Per Cycle (IPC)`, `Cache Misses Rate` and `Stalled Cycles Backend` shown in Fig. 2(b). 1) IPC measures how efficiently the CPU executes instructions; higher values indicate better efficiency and less time spent on accessing external memory, thus less interference to other processes. 2) High cache miss rate indicates poor locality with higher memory access, which contends for limited memory bandwidth. This could be caused by sub-optimal implementation of matrix multiplications where the tensors do not fit into the L2 cache. 3) High backend stall suggests that the CPU frequently waits for resources, which is exacerbated by resource contention.

Observation 2 (Contention from Heavyweight Models). Matrix multiplication (`MatMul`) with large dimensions have

[1]Embedded GPUs such as ARM Mali and Qualcomm Adreno also lack the variety of performance counters compared to CPUs.

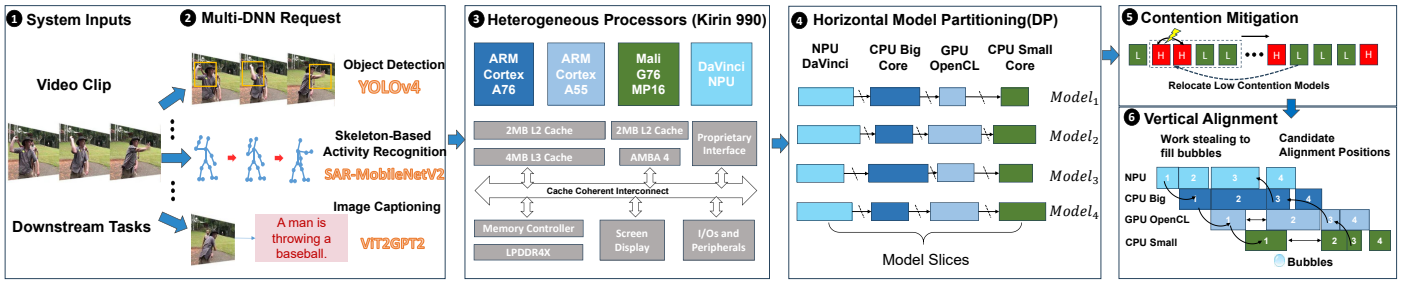


Fig. 3: An example of major steps of Hetero^{2}Pipe: ❶ Multimedia inputs such as images and text;❷ Multi-DNN requests defined by the downstream tasks; ❸ Execution on a heterogeneous platform of Kirin 990 SoC with shared interconnections; ❹ Partition of each model inference along the horizontal directions independently; ❺ Interleave the high contention models by re-arrangements; ❻ Vertical alignments to reduce the pipeline bubbles by work stealing.

lower data locality and is often memory-bound. These include the Fully connected (FC) layers in most CNN models such as the VGG family as well as the multi-head attention layers in Transformers such as BERT.

Our experiment shows that the FC layers in VGG/AlexNet alone have 2-4× higher cache miss compared to CONV layers on ARM Cortex A76. Similarly, multi-head attention layers with 768×768 `MatMul` and the Layer Normalization layers with 768×3072 `MatMul` in BERT also impose high memory access, but the uniform intermediate dimensions of Transformers make model partition and pipeline planning more straightforward compared to convolutional models.

Observation 3 (Contention from Lightweight Models). Intuitively, lightweight models measured by FLOPs should incur less contention to their co-executing peers. However, we discover surprising outliers that models such as SqueezeNet (4.8MB) and GoogleNet (23 MB) have relatively high resource demands, indicated by `IPC`, `Stalled Cycles Backend` and `Cache Miss Rate`. As shown in Table II, during co-execution, SqueezeNet imposes an additional 10% slowdown compared to large models like ViT of 70× in size.

Thus, it is imperative to develop a method to quantify *contention intensity* so the demanding model requests are interleaved temporally. Thus, we leverage the effective `perf events` as features $\mathbf{X} = \{x_1, x_2, x_3\}$ to learn a regression of *contention intensity* $\mathbf{Y}$ with an α-regularization term to alleviate overfitting,

$$\mathbf{W} = \arg\min_{w} \frac{1}{2}(\mathbf{XW} - \mathbf{Y})^{\top}(\mathbf{XW} - \mathbf{Y}) + \frac{1}{2}\alpha \left\|\mathbf{W}\right\|_2^2 \quad (1)$$

where the weight matrix can be calculated as $\mathbf{W} = (\mathbf{X}^{\top}\mathbf{X} + \alpha\mathbf{I})^{-1}\mathbf{X}^{\top}\mathbf{Y}$, $\mathbf{I}$ is the identity matrix. Thus, for new inference requests, we could quickly estimate its *contention intensity* given the `perf event` statistics as shown in Fig. 2(b).

IV. System Model and Problem Formulation

System Model. We consider consumer mobile devices with three typical heterogeneous processors: CPU, GPU and NPU, where the CPU consists of Big/Small clusters. The GPU/NPU are considered as a single unit and cannot be partitioned further [20], [33]. Different from CUDA platforms, the CPU is accelerated by ARM NEON (SIMD for multi-core CPUs) [34]; GPU acceleration is enabled by cross-compiling with OpenCL [35] and NPU inference is launched via a proprietary API interface [36]. The streaming inference requests are comprised of a set of $\mathcal{M}$ heterogeneous models. We organize the processors in a descending order of their processing power (processing speed: NPU $\gg$ CPU Big $\geq$ GPU $\gg$ CPU Small). For NPU, if a single operator is not supported, it leads to processing error and a viable way is through operator fallback [20]. Hetero^{2}Pipe also supports this by forwarding the sub-model to the CPU Big cores/GPU.

The entire system architecture is shown in Fig. 3: ❶ the system takes multimedia inputs such as images and text; ❷-❸ launch multi-DNN requests for the downstream applications on a heterogeneous SoC; ❹ partition each model along the horizontal directions independently; ❺ interleave the high contention models by processing re-ordering; ❻ reduce the pipeline bubbles by work stealing in the vertical direction. To dispatch the models to different processors, we perform model slicing as defined below.

Definition 1 (Model Slicing). Define a K-way partition $P = \{p_1, \cdots, p_K\}$ that splits the model into layer slices and distributes to K heterogeneous processors,

$$\mathcal{P} \rightarrow \left\{\{l_0, \ldots, l_{p_1-1}\}, \{l_{p_1}, \ldots, l_{p_2-1}\}, \ldots, \{l_{p_{K-1}}, \ldots, l_{p_K}\}\right\},$$

where K is also the *pipeline depth*. Since it is computationally intensive to provide a layer-wise granularity for slicing large models, we consider a coarse-grained model slicing strategy of K slices.

Definition 2 (Execution time). The total execution time $T_k(\cdot)$ of a model slice $\{l_{p_k}, \cdots, l_{p_{k+1}-1}\}$ on processor k is defined as,

$$T_k^i(l_{p_k}, \cdots, l_{p_{k+1}-1}) = \underbrace{T_k^e(l_{p_k}, \cdots, l_{p_{k+1}-1})}_{\text{solo execution time}} + \underbrace{T_k^c(l_{p_k}, l_{p_{k+1}-1})}_{\text{memory copy time}} + \underbrace{T_k^{\text{co}}(l_{p_k}, \cdots, l_{p_{k+1}-1} | l_{p_i}, \cdots, l_{p_{i+1}-1}, i \in \mathcal{M}/k)}_{\text{co-execution slowdown}} \quad (2)$$

where the first term is the pure execution time when the model slice is executed alone on processor k. The second term represents the memory copy time while copying the input/output tensor between different memory addresses on the unified memory architecture for different processors. The last term represents slowdown given two or more processors are co-running different model slices. A caveat in pipeline planning is the sequential interdependence between consecutive stages. If the finishing times are misaligned, they could lead to substantial pipeline bubbles and resource wastage as defined below.

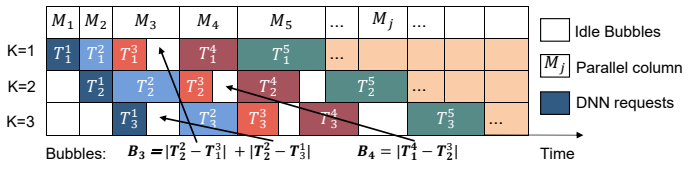


Fig. 4: An example of a three-stage pipeline.

Definition 3 (Pipeline Bubbles). Similar to [6], we define pipeline bubbles as the idling time of processors due to pipeline stall (Fig.4), i.e., when a model slice completes on processor k, but waits for processor $k+1$ to finish processing. Specifically, denote sub-model i on processor k as M_k^i and its processing latency as T_k^i for short. Denote the concurrent workloads as $\mathcal{M}_j$, $j = 1, 2, \ldots, |\mathcal{M}| + K - 1$, which is represented by different columns of simultaneous execution of model slices in the vertical direction. Formally, the bubble size $|B_j|$ for the j-th sub-models on different processors can be written as,

$$|B_j| = \sum_{M_k^i \in \mathcal{M}_j} \left(\max_{\forall M_k^i \in \mathcal{M}_j} \{T_k^i\} - T_k^i \right) \tag{3}$$

In addition to the dependency and misalignment due to load unbalancing, the co-execution slowdown would also exacerbate pipeline bubbles. Different from model-paralleled training [6]–[8], we also optimize bubbles accumulated at the tail of the pipeline for inference. The following property demonstrates an empirical relation between pipeline bubbles and the overall latency.

Property 1 (Bubble vs. Latency). There is a linear relation between pipeline bubbles and the overall latency (empirically evaluated by Fig. 12 in Appendices). Hence, optimizing the overall latency and throughput are equivalent to minimizing the pipeline bubbles.

Problem Formulation. Our goal is to find a pipelining plan such that the total inference delay is minimized. We formulate it into a two-step optimization problem in the *horizontal* and *vertical* directions: horizontally, we partition each model *independently* to balance workloads across heterogeneous processors and minimize the makespan; vertically, we consider heterogeneous model slices across K processors based on the horizontal solutions to reduce pipelining bubbles with contention mitigation.

Horizontal Direction (P1). For each model $\mathcal{M}$, the horizontal optimization aims to find partition strategy P such that the maximum execution time (makespan) is minimized, which effectively balances the computational loads across different processors,

$$\mathbf{P1}: \mathcal{P} = \arg\min_{\{p_1, \cdots, p_k\}} \max_{1 \le k \le K-1} T_k^i(l_{p_k}^i, \cdots, l_{p_{k+1}-1}^i), \tag{4}$$

where $|\mathcal{P}| = K$ and $\forall i = \{1, \cdots, m\} \in \mathcal{M}$. Since the horizontal formulation optimizes different models independently, we further consider the vertical direction across different pipelining stages.

Vertical Direction (P2). Since horizontal optimization performs optimal partition for each model/processor independently, it leaves workloads across different processors unbalanced with pipeline bubbles. The vertical optimization problem aims to minimize the sum of bubbles,

$$\mathbf{P2}: \min \sum_{j=1}^{|\mathcal{M}|+K-1} |B_j| \tag{5}$$

s.t.

$$\sum_{k=1}^{K} |\{l_{p_k}^{i-k+1}, \cdots, l_{p_{k+1}-1}^{i-k+1}\}| \le C_{\text{mem}} \tag{6}$$

$$T_k^{\text{co}}(l_{p_k}, \cdots, l_{p_{k+1}-1} | l_{p_j}, \cdots, l_{p_{j+1}-1}, j \in \mathcal{M}/k) > 0 \tag{7}$$

$$t_{p_{k-1}}^i + T_k^i(l_{p_{k-1}}^i, \cdots, l_{p_k-1}^i) \le t_{p_k}^i \tag{8}$$

Constraint (6) imposes that concurrent execution of model slices is bounded by the memory capacity C_{mem} to avoid page faults and performance degradation due to memory swaps [21]. Constraint (7) states that co-execution slowdown is non-negligible on realistic edge devices. Constraint (8) ensures the precedence of executing consecutive model slices down the pipelines, i.e., processor k must wait for the tensors from processor $k-1$ from the previous stage.

V. Contention-Aware Heterogeneous Pipeline Execution

In this section, we optimize inference execution on mobile processors by considering both *model* and *processor* heterogeneity describe in the following. For completeness, we also analyze the solution search space in the Appendices.

A. Horizontal Model Partitioning

Horizontal optimization can be obtained by solving $|\mathcal{M}|$ dynamic programming problems independently. With a slight abuse of notation, define $T_k^e(i, j)$ as the sum of $T_k^e(l_i, \ldots, l_j) + T_k^c(l_i, l_j)$ that combines the solo execution and memory copy time in Eq. (2) for each model.

Solution via Dynamic Programming. For the n-layer network, the partition problem has an optimal sub-structure. Define $S^*(j, k)$ as the min-max execution time in the optimal partition from layer 0 to j on k processors, where $1 \le k \le K$ and $0 \le j \le n-1$. $S^*(j, k)$ has the following optimal sub-structure:

$$S^*(j,k) = \begin{cases} T_k^e(0, j), & k = 1 \\ \max\limits_{i \le j < n} T_k^e(i, i), & k = j \\ \min\limits_{i \le j \le n-k} \max\{S^*(i-1, k-1), T_k^e(i, j)\}, & \text{otherwise,} \end{cases}$$

where the first two are boundary conditions and the last one is the recurrence equation. By utilizing the sub-problems of $S^*(i-1, k-1)$ and $T_k^e(i, j)$, we can iteratively obtain the optimal partition $S^*(j, k)$.

The entire procedure is detailed in Algorithm 1. We first initialize a table S^* as the latency for the partition decisions. For each row j, the optimal solution of a single partition, $S^*(j, 1)$, is computed using $T_0^e(0, j)$ and serves as the basis for subsequent recursive state transitions. For the remaining partitions, from $k = 2$ to p, we partition from $[0, i]$ with an initial value of $i = 2$. Iterating over each row from $j = 0$ to $n-1$, the maximum partition cost is identified via $\max(S^*(i-1, k-1), T_{k-1}^e(i, j))$. This bifurcation in strategy is due to if the division time for $[0, i]$ into $k-1$ partitions exceeds the inference time for $[i, j]$, the optimal partition cost $S^*(j, k)$ is equivalent to $S^*(i-1, k-1)$, thereby reducing unnecessary computations.

Algorithm 1: Horizontal Model Partitioning

Input: $T^e_k(i,j),\ k \in \mathcal{K}$. ▷ Inference time of submodel (i,j) on k-th processors
Output: $\mathcal{S}^* = \{S_1, S_2, \ldots, S_{K-1}\}$. ▷ Optimal splitting points
Initialize $\forall j \in [0, n-1], S^*[j][1] \leftarrow T^e_0(0,j)$
for $k = 2$ *to* p **do**
  $i \leftarrow 2$
  for $j = 0$ *to* $n-1$ **do**
    $S^*_{max} \leftarrow \max(S^*(i-1,k-1), T^e_{k-1}(i,j))$
    if $S^*_{max} = S^*(i-1,k-1)$ **then**
      $S^*(j,k) \leftarrow S^*(i-1,k-1)$
    else
      while $T^e_{k-1}(i+1,j) \geq S^*(i,k-1)$ **do**
        $i \leftarrow i+1$
      $S^*(j,k) \leftarrow \min(S^*(i,k-1), T^e_{k-1}(i,j))$
$S_i \leftarrow 0$
for $S_j = 1$ *to* $n-1$ **do**
  if $T^e_k(S_i, S_j) \geq S^*(n-1,p)$ **then**
    $\mathcal{S} \cup S_j$
    $S_i \leftarrow S_j$
    $k \leftarrow k+1$
return $\mathcal{S}^* = \{S_1, S_2, \ldots, S_{K-1}\}$

When $T^e_{k-1}(i,j) \geq S^*(i-1,k-1)$, further partitioning is required. Hence, the goal is to locate a balance point i, such that the segment sum from i to j ($T^e_{k-1}(i,j)$) minimizes the discrepancy with the optimal solution obtained by dividing the first i rows into $k-1$ partitions. The state transition is completed by setting $S^*(j,k) = \min(S^*(i,k-1), T^e_{k-1}(i,j))$. To analyze the complexity, we leverage the monotonicity property to reduce the search space.

Property 2 (Monotonicity). $T^e_k(i,j)$ satisfies the following monotonicity condition:

- $T^e_k(i+1,j) < T^e_k(i,j) < T^e_k(i,j+1),\ 0 \leq i \leq j < n-1,\ 1 \leq k \leq K$.
- $T^e_k(i,j) = 0$, only if $j < i$.

Based on the monotonous property of $T^e_k(i,j)$, the algorithm updates the value of i based on the condition $T^e_{k-1}(i+1,j) \geq S^*(i,k-1)$. Once $S^*(n-1,p)$ is obtained, we can backtrack through table S^* to identify the optimal set of partition points.

Computation Complexity. For n layer model, binary search takes $O(nK)$ iterations. We leverage prefix sum to optimize the computation of $T^e_k(i,j)$ in $O(1)$. With the monotonicity property of $T^e_k(i,j)$, we reduce the time complexity from $O(n^2K)$ to $O(nK)$. In the worst case, when $K = n$, Algorithm 1 takes $O(n^2)$ time. For serial processing of the $|\mathcal{M}|$ models, $O(|\mathcal{M}|nK)$ time is required.

B. Mitigation of Co-Execution Interference

Interference on the memory bus occurs when the resource-demanding workloads are co-located to each other in the pipeline, i.e., when slices from the resource-intensive models are mapped to the same temporal axis. A solution is to re-order the input sequence such that the demanding workloads are temporally interleaved, while preserving the original execution order as much as possible. According to the analysis in Sec. III, network models exhibit different degrees of contention. Thus, we first use a percentage threshold to split the inference requests into high ($\mathcal{H}$) and low ($\mathcal{L}$) contention (also denoted by `H` and `L`).

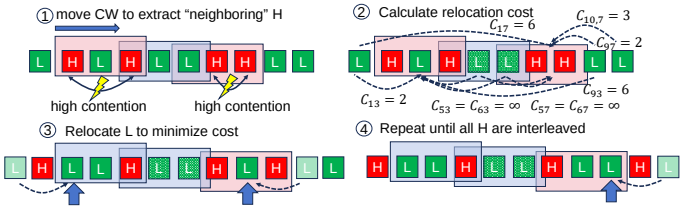


Fig. 5: Illustration of the contention mitigation algorithm.

To bound the radius of mutual influence, we define *contention window* below.

Definition 4 (Contention Window). For each model j to be pipelined on K processors, by looking forward, the contention window spans from j to $j+K-1$.

Denote $|\mathcal{H}_j|$ and $|\mathcal{L}_j|$ as the number of high and low contention model slices in contention window j. Once the number of high contention model slices is larger than two, there would be a temporal overlap between two or more model slices with high contention. For example, `HH` for $K = 2$; `HLH`, `HHL`, `LHH`, `HHH` all have two high contention model slices for $K = 3$, and so on. Thus, we seek to relocate a low-contention model slices `L` so that the number of high contention is less than two, which is illustrated by the next property.

Property 3 (Contention Mitigation). If u and v are the indices of two `H` in $\mathcal{H}_j$ and the contention distance is $d = |v-u|$. We need to relocate at least $K-d$ number of `L` from $\mathcal{L}$ to make the number of high contention less than two.

Relocation via Linear Assignment. We transform such contention mitigation problem into the classic Linear Assignment Problem (LAP), i.e., relocate $i \in \mathcal{L}_j$ to $\mathcal{H}_j$ such that the total moving cost c_{ij} is minimized,

$$\textbf{P3}: \min \sum_{i=0}^{|\mathcal{L}|-1} \sum_{j=0}^{|\mathcal{H}|-1} c_{ij} x_{ij}, \text{ where } \sum_{i=0}^{|\mathcal{L}|-1} x_{ij} = \sum_{j=0}^{|\mathcal{H}|-1} x_{ij} = 1 \quad (9)$$

and x_{ij} is a 0-1 decision variable. Note that the assignment cost is calculated as,

$$c_{ij} = \begin{cases} \infty, & i \in [j-K+1, j+K-1], \\ & i \to |\mathcal{H}|_j \overset{if}{\Longrightarrow} |\mathcal{H}|_i \geq 2 \\ |j-i|, & \text{otherwise.} \end{cases} \quad (10)$$

It states when i is either in the left or right contention window of j; or after i is assigned to j, it leads to high contention. We do not consider these $i \in \mathcal{L}$ and set the $c_{ij} \to \infty$; otherwise, the cost is set to the contention distance between i and j. We can solve the problem by the *Kuhn–Munkres Algorithm* in $\mathcal{O}(|\mathcal{M}|^3)$ [37].

The entire procedure is visually illustrated in Fig. 5. For each input sequence, ① slide the contention window to extract all the neighboring `H`; ② calculate the relocation cost according to Eq. (10); ③ find the min-cost assignments so that the total displacement cost is minimized by the Kuhn–Munkres Algorithm; ④ stop until all neighboring `H` are at least K apart or there is no sufficient `L` for selection. The time complexity is bounded by $\mathcal{O}(|\mathcal{M}|^3|\mathcal{H}|)$ and the procedure is summarized in Algorithm 2.

Algorithm 2: Contention Mitigation

Input: Input sequence $\mathcal{M}$, their *contention intensity* $\mathcal{C}_I$, processors K, function of Kuhn–Munkres Algorithm $\mathcal{F}_{KM}(\cdot)$.
Output: Contention-mitigated model sequence $\hat{\mathcal{M}}$.
$\mathcal{C}_I \to \mathcal{H} \bigcup \mathcal{L} \subseteq \mathcal{M}$, $\{\mathcal{H}_j \geq 2\}_{\forall j \in \mathcal{M}} \to \mathcal{S}_H$, $\mathcal{L} \to \mathcal{S}_L$.
while $\mathcal{S}_H, \mathcal{S}_L \neq \emptyset$ **do**
  for $\forall i \in \mathcal{S}_H, \forall j \in \mathcal{S}_L$, $i \notin [j-K+1, j+K-1]$ **do**
    if $i \to |\mathcal{H}_j| \overset{if}{\Longrightarrow} |\mathcal{H}_i| \geq 2$ **then**
      $c_{ij} \leftarrow \infty$ ▷ Exclude infeasible solutions
    else
      $c_{ij} = |j - i|$ ▷ Set cost to displacement distance
  $(i^* \to |\mathcal{H}_j|) \leftarrow \mathcal{F}_{KM}(\mathcal{S}_H, \mathcal{S}_L, c_{ij})$. ▷ Find mapping
  $\mathcal{S}_L \leftarrow \mathcal{S}_L - i$, $\mathcal{S}_H \leftarrow \mathcal{S}_H - j$, $|\mathcal{H}_j| \leftarrow \mathcal{H}_j| - 1$.
return $\hat{\mathcal{M}}$.

Algorithm 3: Vertical Alignment by Work Stealing

Input: $\forall k \in \mathcal{K}$, $i \in \mathcal{M}$, execution time $T_k^i(l_{p_k}, \cdots, l_{p_{k+1}-1})$ from **Algorithm 1**, output sequence $\hat{\mathcal{M}}$ from **Algorithm 2**, contention windows $|CW|$ of size K.
Output: Updated partitioning $\hat{\mathcal{P}}$ of $\hat{\mathcal{M}}$.
for $0 \leq u \leq |\mathcal{M}|$ **do**
  $\mathcal{M}_{cw} \leftarrow |CW|_u$ ▷ Extract models in CW indexed by u
  $i_c \leftarrow \arg\max_{i \in \mathcal{M}_{cw}} \sum_{k=1}^{K} T_k^i(l_{p_k}, \cdots, l_{p_{k+1}-1})$ ▷ Critical path
  for $1 \leq i \leq K - i_c$ **do**
    for $1 \leq j \leq k-1$ **do**
      Assign layers $\ell \in M_{k-j}^{i_c+i} \to M_{k-j+1}^{i_c+i}$ till $T_{k-j}^{i_c+i} - T_k^{i_c} \to 0$ ▷ Work stealing right
  for $1 \leq i \leq i_c - 1$ **do**
    for $1 \leq j \leq K-k$ **do**
      Assign layers $\ell \in M_{k+j}^{i_c+i} \to M_{k+j-1}^{i_c+i}$ till $T_{k+j}^{i_c+i} - T_k^{i_c} \to 0$ ▷ Work stealing left
  $u \leftarrow u + K$ ▷ Sliding CW by step K
return Updated partitioning $\hat{\mathcal{P}}$ of $\hat{\mathcal{M}}$.

C. Vertical Alignments by Work Stealing

After we successfully re-arrange the input sequence, the final step is to perform cross-stage alignment in the vertical direction to minimize pipeline bubbles. Our main strategy is to identify the *critical path* with the longest processing delay and utilize *work stealing* to adjust workloads in neighboring stages in order to amortize the pipeline bubbles. Work stealing is a common technique in multi-core processing that decouples tasks from the executing threads to allow work units to move between thread contexts [38]. Here, if a bubble is present between the two execution stages, the lesser would request additional work from its next corresponding stage until the stages are re-balanced.

The main procedure consists of two phases: 1) Perform work stealing within each contention window (CW) and slide it till the end of the sequence; 2) Conduct local search to minimize the tail bubbles. In 1), for models $\mathcal{M}_{cw} \subseteq \mathcal{M}$, we first find the critical path $i_c = \arg\max_{i \in \mathcal{M}_{cw}} \sum_{k=1}^{K} T_k^i(l_{p_k}, \cdots, l_{p_{k+1}-1})$. Then we perform work stealing in a layer-wise granularity, e.g., if $T_{k-1}^{i_c+1} - T_k^{i_c} > 0$, we re-allocate layers from model slice $M_{k-1}^{i_c+1}$ to $M_k^{i_c+1}$ to so that $T_{k-1}^{i_c+1} \approx T_k^{i_c}$. The objective is to vertically align the execution time across different stages so that the pipeline bubbles are minimized for $k = \{1, \cdots, K\}$,

$$\min \sum_{i=1}^{m} |B_k^i| = \sum_{i=1}^{K-i_c} \sum_{j=1}^{k-1} |T_{k-j}^{i_c+i} - T_k^{i_c}| + \sum_{i=1}^{i_c-1} \sum_{j=1}^{K-k} |T_{k+j}^{i_c-i} - T_k^{i_c}|. \tag{11}$$

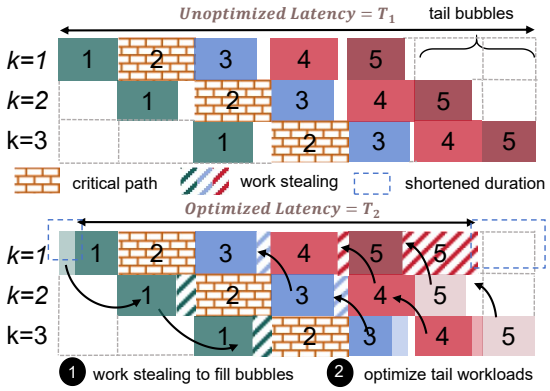


Fig. 6: Example of the vertical alignment via work stealing. The second model is the critical path and the arrows indicate the flow of work stealing.

As shown in Fig. 6, this not only reduces under-utilization at the beginning, but also gradually drains the workloads towards the end of the pipeline by filling the intermediate bubbles, which in turn reduces the total makespan. In the second phase, different from pipelined training with dependencies on the gradients [6], [7], inference execution allows us to further reduce the bubbles on the tail via re-allocating workloads by local search, e.g., a fast greedy approach would be simply assigning all the workloads to the fastest processor, or we can perform an exhaustive search since the search space is only K.

Time Complexity. The time complexity of vertical alignment is $\mathcal{O}(\frac{\mathcal{M}}{K} \cdot nK + \frac{|\mathcal{M}|}{K} \cdot K^2) = \mathcal{O}(|\mathcal{M}|(n+K))$. The analysis takes the worst-case that in each CW, $\mathcal{O}(nK)$ number of layer alignments are needed and the sequence consists of $\frac{\mathcal{M}}{K}$ steps. In summary, the total complexity of the Hetero^{2}Pipe planner is $\mathcal{O}\big(|\mathcal{M}|(nK+n+K) + |\mathcal{M}|^3|\mathcal{H}|\big)$, where n represents the (average) number of layers, K is the number of processors. We can see that the overall time complexity is primarily determined by the number of inference requests $|\mathcal{M}|$, which is further governed by how often the pipelining plan is made. In case of more inference requests, the planner should be scheduled more frequently to avoid enlarged search space.

VI. Experimental Results

A. Environment Setup

Hardware. We employ three heterogeneous SoCs for our experiments:

- **Kirin 990.** The SoC contains a CPU with 2-core A76 @2.86GHz, 2-core A76 @2.09GHz, and 4-core A55 @1.86GHz, a GPU with 16-core Mali-G76, and a NPU with the DaVinci Architecture.
- **Snapdragon 778G.** This SoC features a CPU with 1-core A78 @2.40GHz, 3-core A78 @2.20GHz, and 4-core A55 @1.90GHz, and a GPU with Qualcomm Adreno 642L.

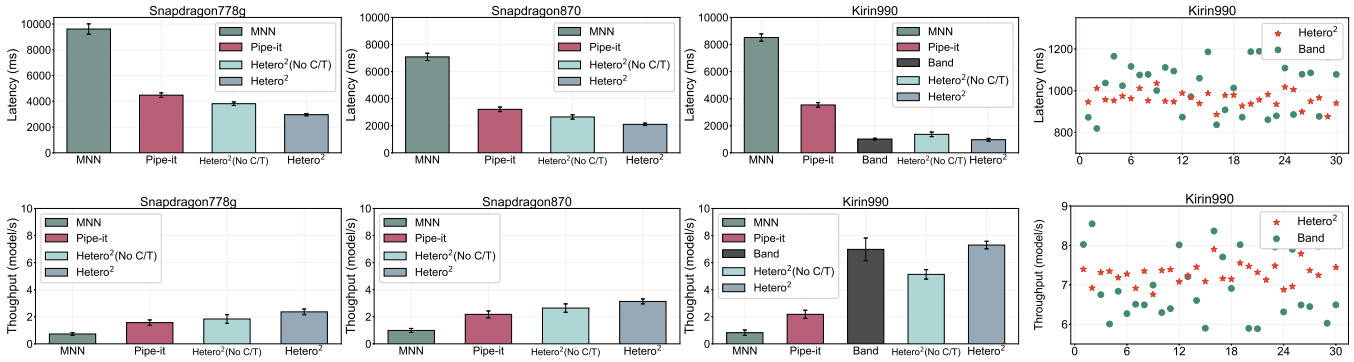


Fig. 7: Comparison of the overall performance from samples of 100 random model combinations on Snapdragon 778G, Snapdragon870 and Kirin990 SoCs. The top and bottom charts represent *Latency* and *Throughput*, respectively. The rightmost scatter plots compare Band and our scheme by showing the solution distributions from a random subset of 30% sequence combinations.

✧ **Snapdragon 870.** It consists of a CPU with 1-core A77 @3.20GHz, 3-core A77 @2.42GHz, and 4-core A55 @1.80GHz, and a GPU with Qualcomm Adreno 650.

We utilize the `ADB` interface to execute the program and monitor the kernel events. Additionally, we bind the process to CPU cores using the UNIX `taskset` command. The order of the processors is arranged by computational capability from high to low.

Software. We implement Hetero^{2}Pipe and other benchmarks in the MNN software library [39]. MNN is a versatile, lightweight, industrial-grade deep learning framework that supports both inference and training with a large variety of operators. We cross-compile on ARMv8 with the Android NDK r25c version using ARM NEON, OpenCL and the HiAI suite for the CPU, GPU and NPU, respectively. We launch the pipeline program under the Android `/data/local/tmp` folder from the ADB Shell.

Inference Models. To simulate real applications, we consider a combination of 10 representative DNNs: AlexNet, VGG16, GoogLeNet, Inceptionv4, ResNet50, YOLOv4, MobileNetV2, SqueezeNet, BERT and ViT. The collection scales from over-parameterized convolution networks with large filters, branching and residual connections, depth separable convolutions, and object detection to the latest transformer-based architectures. We use pre-trained models in their ONNX format and convert them to MNN format with `MNNConvert`.

Baseline. We compare Hetero^{2}Pipe with the following baselines.

✧ **MNN v2.6.0** [10]: since the CPU still outperforms the embedded GPU in most mobile consumer devices, this represents the vanilla CPU-centric implementation on the Big cores.
✧ **Pipe-it** [19]: it pipelines inputs across different CPU cores for homogeneous DNNs. We adapt the core partitioning strategy for heterogeneous DNNs and select the fastest core combination of four Big and four Small cores to avoid cache incoherence across the CPU clusters.
✧ **Band** [20]: we implement the co-processor fallback mechanism to migrate unsupported NPU operations to CPU/GPU.
✧ **Hetero^{2}Pipe (No C/T)**: the proposed Hetero^{2}Pipe without considering *contention mitigation* and *tail bubble* optimization.

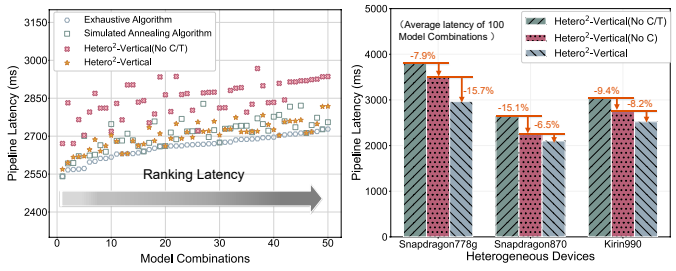
B. Comparison of Overall Performance

First, we compare the overall performance of Hetero^{2}Pipe on three SoC platforms in Fig. 7 with the setup of four processors (CPU Big cores, OpenCL GPU, CPU Small cores and NPU) and various combinations of multi-DNN networks. The system throughput is defined as the number of completed model inferences per unit time: Throughput $=$ # Model/Latency.

Compared to MNN [10], Hetero^{2}Pipe accelerates inference by $4.2\times$ on average, achieving up to $8.8\times$ speedup on the Kirin 990 platform due to NPU acceleration. Compared to Pipe-it [19], Hetero^{2}Pipe accelerates inference by $2\times$ on average, achieving up to $3.7\times$ speedup on the Kirin 990 SoC. It is also observed that with particular contention mitigation and tail bubble optimization, Hetero^{2}Pipe outperforms the unoptimized "No C/T" version by $1.3\times$ on average.

Comparison to Band [20]. Band can be considered a competitive SOTA scheme that orchestrates the fastest NPU on-board. It prioritizes model inference on high-performance processors based on operator supportability, and falls back to secondary ones for unsupported operators which achieves efficient parallel inference through dynamic processor switching. The two scatter plots in Fig. 7 compare solution points between Band and Hetero^{2}Pipe. For clarity, we extract a random subset of 30% model sequence to visualize the solution distributions. On average, our scheme surpasses Band by 5% with less solution variance (admittedly, Band is able to achieve better performance in certain cases). Band can be regarded as a special pipeline design that also efficiently harnesses NPU for parallel collaboration among heterogeneous processors. In general, the two schemes are similar conceptually as they both take advantage of heterogeneous processors. The difference resides in the parallelism optimization as Band does not purposely optimize pipelines. This may lead to bubbles and resource idling during operator fallback whereas the following requests are far from being aligned. On the system-level, because Band is fallback-driven, it leads to constant new memory allocation and data transfer, which may leave memory fragments that further limit its sustainability in the long run. In our scheme, the NPU execution stops and delegates to the next pipeline stage either because

of unsupported operators or because a model partition decision has been made.



(a) (b)

Fig. 8: Ablation studies of vertical optimization over 100 random model combinations. (a) Comparison with exhaustive search, simulated annealing and no contention mitigation/tail bubble optimization. (b) Average latency achieved by removing components from Hetero^{2}Pipe.

C. Ablation Studies for Vertical Optimization

The vertical optimization consists of several components. To demonstrate their effectiveness, we conduct additional ablation studies and compare them with *exhaustive search* and meta-heuristics such as *simulated annealing*. Fig.8(a) shows 100 samples of random model combinations and we organize the model sequences on the x-axis in an ascending order of the latency values. Our scheme ranks very close to the solution found by exhaustive search (only 4% away from the optimality) and outperforms simulated annealing with much lower complexity. Fig.8(b) further validates the consideration of contention mitigation and tail bubble optimization with progressive reduction of latency when both factors are accounted.

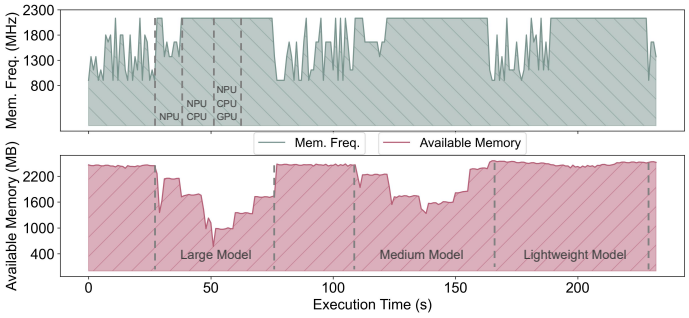


Fig. 9: Visualization of memory frequency and footprint due to pipeline executions on Kirin990. The upper and bottom plots trace the memory frequency and available memory respectively.

D. Memory Footprint

A side effect of pipeline execution on edge devices is the increase of burden in the memory subsystems due to the running multiple pipeline stages concurrently. On one hand, co-execution demands full memory bandwidth, thus often throttling the memory controller to the maximum memory frequency. On the other hand, if the peak memory exceeds the physical memory capacity, it leads to page faults, virtual memory swapping, and substantial performance slowdown [21]. While memory frequency is managed by the proprietary driver from the vendor, it is essential to guarantee the peak memory is under the physical memory capacity (Constraint (6)). Fig.9 demonstrates the memory frequency and usage on the Kirin 990 platform by building pipelines proportional to the FLOPs: large models (BERT, ViT, YOLOv4) over 300 MB, medium models (Inceptionv4, ResNet50, AlexNet) between 100-300 MB, and lightweight models (SqueezeNet, MobileNetV2, GoogLeNet) under 100 MB. We notice that single-stage execution on the NPU does not require full memory bandwidth indicated by frequency. Once the CPU/GPU are involved, memory frequency is running at the maximum state. For the initial available memory around 2.5GB, the 3-stage pipeline brings the available memory down to 500 MB and adding one more large model would likely overwhelm the capacity. Fortunately, our contention mitigation mechanism could avoid concurrent executions of the large models because the large models tend to cause higher interference.

VII. Conclusion

This paper proposes Hetero^{2}Pipe, a pipeline planner for multi-DNN inference on the platforms with heterogeneous processors under co-execution slowdown. We first characterize memory interference for different models on the processors and investigate through the performance monitoring module. Based on the findings, we formulate the pipeline organization problem into a two-step optimization with dynamic programming-based model partition horizontally and work stealing across different stages for load re-balancing vertically. Our extensive experiments over a large collection of models on three representative mobile SoCs demonstrate the effectiveness of the proposed mechanism compared to the existing mechanisms.

VIII. Appendices

We follow up with more discussions and evaluations on a set of interesting phenomenons and insights from our implementations below.

A. Search Space of Processor Pipelines

Consider three typical heterogeneous mobile processors: CPU, GPU and NPU. The CPU consists of C_b Big cores and $C - C_b$ Small cores; the GPU/NPU is considered as a single unit and cannot be partitioned further on mobile devices. We first calculate the number of possible partitions S_p for a single model in Eq. (12) with P stages and denote $P' = P - 2$ as the number of stages for the CPU cores where the two stages are reserved for GPU and NPU.

$$S_P = \sum_{P_b^{\min}}^{P_b^{\max}} (4D_bD_s + 3D_b + 3D_s) + 1 \tag{12}$$

where,

$$\begin{aligned} &D_b = \binom{C_b - 1}{P_b - 1}, \quad D_s = \binom{C - C_b - 1}{P' - P_b - 1}, \\ &P_b = \{1, \cdots, C_b\}, \quad P' - P_b = \{1, \cdots, C - C_b\}, \\ &P_b^{\min} = \max(1, P' + C + C_b), \\ &P_b^{\max} = \min(C_b, P' - 1). \end{aligned} \tag{13}$$

P_b and $P' - P_b$ are the stages for the Big and Small cores, respectively. D_b and D_s are the combinations on the Big and

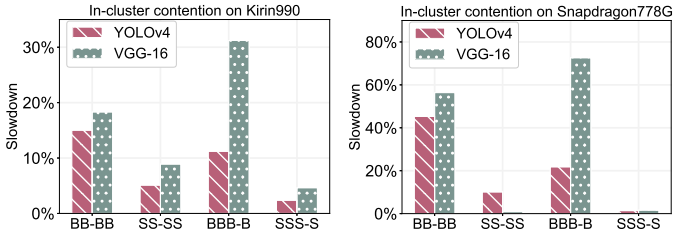


Fig. 10: In-cluster contention between CPU cores on Kirin990 and Snapdragon778G causes as high as 70% slowdown on the performance cores, which is downplayed by [19]. "BB-BB" represents when YOLOv4 and VGG-16 are co-executed on the two separate Big cores, and so on for "SS-SS", "BBB-B" and "SSS-S". To avoid contention during further partitioning of the Big and Small cores, we consider all four Big/Small cores together.

Small cores. For $|\mathcal{M}|$ models with n layers each, the entire search space is,

$$S = \prod_{i=1}^{|\mathcal{M}|} \sum_{P=2}^{C+2} \binom{n-1}{P-1} \cdot S_P. \tag{14}$$

Example. In a typical consumer device with an eight-core CPU, a GPU and NPU, we identify a total of 449 feasible pipelines for P between 2 and 10 from Eq. (12). Moreover, for MobileNetV2 with 28 convolutional layers, there are over 3.6B distinct split points from Eq. (14). For different models, e.g., {MobileNetV2, VGG16, Bert}, the search space grow exponentially.

Remarks on Intra-Cluster Contention. Different from [19] that partitions the workloads on a per-core granularity, we consider the per-cluster granularity of the four Big and Small cores as a whole because our experiments indicate a large intra-cluster contention that causes as high as 70% slowdown due to conflicting L2 cache miss shown in Fig. 10. This simplification helps reduce the search space.

B. Thermal Behaviors

We demonstrate complex thermal behavior of different processors at run-time. As shown in Fig. 11, for continuous inference workloads, the CPU reaches above $60°C$ with a noticeable slowdown, while the GPU/NPU have much better thermal controls within the $50°C$ limit, partially due to lower core frequencies. This observation is consistent with [40] which found that GPU is less prone to throttling compared to the CPU. To alleviate the thermal impact on processing speed, in this work, we conduct all the experiments at the thermal limits when frequency scaling and temperature have reached a steady state.

C. Relations between Pipeline Bubbles and Latency

Our empirical evaluation indicates a general linear relationship between the bubble size and the overall latency in Fig. 12 in which the latency values are found by exhaustive search. The multi-DNN sequence combinations determine the slopes of the linear relation that corresponds to different bubble sizes when various DNN models are co-executed.

D. Batching of Lightweight Models

In practice, we may have multiple requests from different applications running lightweight models such as continuous

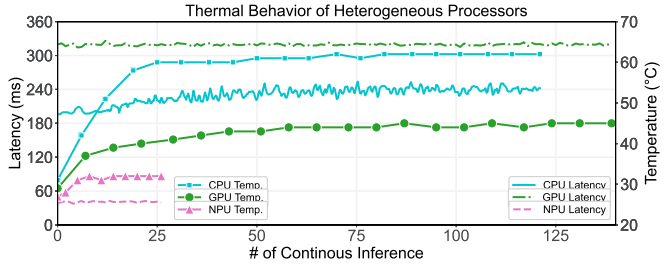
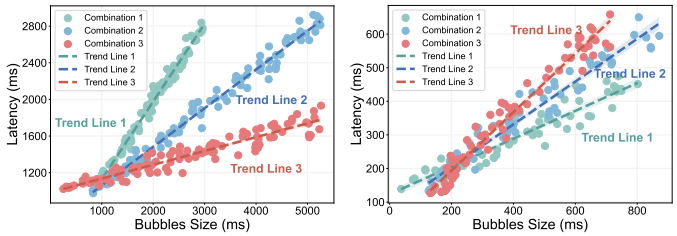


Fig. 11: Processing latency of different models on heterogeneous processors. The Big CPU cores are generally on par with OpenCL GPU and the Small cores pose high performance degradation. NPU has the fastest speed but very limited support for a diverse set of operators [20].



(a) (b)

Fig. 12: Empirical relations between bubble size and overall latency. a) Constructing a five-network pipeline on three processors (ViT, Alexnet, YOLOv4, BERT, MobileNetV2 on CPU Big cores, GPU, CPU Small cores). b) Constructing a three-network on three processors pipeline (InceptionV4, ResNet-50, Squeezenet on NPU, CPU Big cores, GPU).

classification of video frames with MobileNetV2/SqueezeNet. According to our experiment, the execution time of a single inference between such lightweight models against heavyweight models such as BERT is about 20-40× time difference, plus excessive overhead from kernel launching and data transfer from global memory to on-chip buffers while loading the model. Thus, the vertical alignment of a single lightweight model is inefficient and costly due to its short duration. A quick workaround is to adopt *batching* to close the gap between light and heavyweight models, as well as hide the memory copy time of loading a single model. Fig. 13 shows the experimental results of combining inference requests into batches on different mobile processors compared to the CUDA GPU. Due to limited on-chip memory,

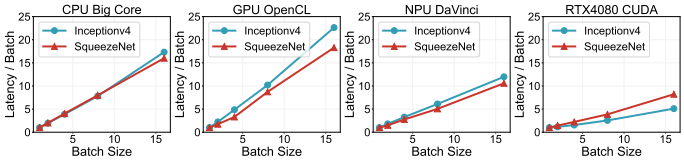


Fig. 13: Batch size can be used to align the execution time of light/heavyweight models. The vertical axis represents the rate of change in inference latency as batch size increases. The linear relation indicates that the computational resources are being fully utilized. Due to limited on-chip memory in mobile devices [33], the execution time climbs almost linearly with the increased batch size.

the execution time can be modeled as an *affine function* regarding batch size for appropriate coefficients calculated for different processors.

REFERENCES

[1] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.

[2] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.

[3] G. Levi and T. Hassner, "Age and gender classification using convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2015, pp. 34–42.

[4] "Vit2gpt image captioning," accessed on July, 2024. [Online]. Available: https://huggingface.co/nlpconnect/vit-gpt2-image-captioning

[5] J. Wei, T. Cao, S. Cao, S. Jiang, S. Fu, M. Yang, Y. Zhang, and Y. Liu, "Nn-stretch: Automatic neural network branching for parallel inference on heterogeneous multi-processors," in *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*, 2023, pp. 70–83.

[6] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[7] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM symposium on operating systems principles*, 2019, pp. 1–15.

[8] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," in *International Conference on Machine Learning*. PMLR, 2021, pp. 7937–7947.

[9] C. Wang, Y. Yang, and P. Zhou, "Towards efficient scheduling of federated mobile devices under computational and statistical heterogeneity," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 394–410, 2021.

[10] "Alibaba mnn framework," accessed on July, 2024. [Online]. Available: https://github.com/alibaba/MNN

[11] C. Wang and H. Wu, "Energy optimization for federated learning on consumer mobile devices with asynchronous sgd and application co-execution," *IEEE Transactions on Mobile Computing*, pp. 1–16, 2024.

[12] Y. Xu, M. E. Belviranli, X. Shen, and J. Vetter, "Pccs: Processor-centric contention-aware slowdown model for heterogeneous system-on-chips," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1282–1295. [Online]. Available: https://doi.org/10.1145/3466752.3480101

[13] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 62–75.

[14] Z. Liu, J. Leng, Z. Zhang, Q. Chen, C. Li, and M. Guo, "Veltair: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 388–401.

[15] Z. Luo, X. Yi, G. Long, S. Fan, C. Wu, J. Yang, and W. Lin, "Efficient pipeline planning for expedited distributed dnn training," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 340–349.

[16] M. Yu, Z. Jiang, H. C. Ng, W. Wang, R. Chen, and B. Li, "Gillis: Serving large neural networks in serverless functions with automatic model partitioning," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, 2021, pp. 138–148.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[18] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[19] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-throughput cnn inference on embedded arm big.little multicore processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2254–2267, 2020.

[20] J. S. Jeong, J. Lee, D. Kim, C. Jeon, C. Jeong, Y. Lee, and B.-G. Chun, "Band: coordinated multi-dnn inference on heterogeneous mobile processors," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, 2022, pp. 235–247.

[21] B. Cox, J. Galjaard, A. Ghiassi, R. Birke, and L. Y. Chen, "Masa: Responsive multi-dnn inference on the edge," in *2021 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2021, pp. 1–10.

[22] N. Ling, X. Huang, Z. Zhao, N. Guan, Z. Yan, and G. Xing, "Blastnet: Exploiting duo-blocks for cross-processor real-time dnn inference," in *SenSys '22: Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, 2022, pp. 91–105.

[23] Y. Kim, J. Kim, D. Chae, D. Kim, and J. Kim, "μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–15.

[24] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[25] Y. Hu, C. Imes, X. Zhao, S. Kundu, P. A. Beerel, S. P. Crago, and J. P. N. Walters, "Pipeline parallelism for inference on heterogeneous edge computing," *arXiv preprint arXiv:2110.14895*, 2021.

[26] T. Feltin, L. Marchó, J.-A. Cordero-Fuertes, F. Brockners, and T. H. Clausen, "Dnn partitioning for inference throughput acceleration at the edge," *IEEE Access*, vol. 11, pp. 52 236–52 249, 2023.

[27] N. K. Unnikrishnan and K. K. Parhi, "Layerpipe: Accelerating deep neural network training by intra-layer and inter-layer gradient pipelining and multiprocessor scheduling," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–8.

[28] J. H. Park, G. Yun, M. Y. Chang, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y.-r. Choi, "{HetPipe}: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 307–321.

[29] D. Kang, J. Oh, J. Choi, Y. Yi, and S. Ha, "Scheduling of deep learning applications onto heterogeneous processors in an embedded device," *IEEE Access*, vol. 8, pp. 43 980–43 991, 2020.

[30] D. Velasco-Montero, B. Goossens, J. Fernández-Berni, Á. Rodríguez-Vázquez, and W. Philips, "A pipelining-based heterogeneous scheduling and energy-throughput optimization scheme for cnns leveraging apache tvm," *IEEE Access*, vol. 11, pp. 35 007–35 021, 2023.

[31] X. Yang, Z. Xu, Q. Qi, J. Wang, H. Sun, J. Liao, and S. Guo, "Pico: Pipeline inference framework for versatile cnns on diverse mobile devices," *IEEE Transactions on Mobile Computing*, vol. 23, no. 4, pp. 2712–2730, 2024.

[32] Y. Xiang and H. Kim, "Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 392–405.

[33] J. Lee, Y. Liu, and Y. Lee, "Parallelfusion: towards maximum utilization of mobile gpu for dnn inference," in *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, 2021, pp. 25–30.

[34] "Arm neon," accessed on July, 2024. [Online]. Available: https://developer.arm.com/Architectures/Neon

[35] "Arm mali gpu opencl developer guide," accessed on July, 2024.

[36] "Kirin 990," accessed on July, 2024. [Online]. Available: https://www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-990

[37] H. W. Kuhn and J. Munkres, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[38] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 356–368.

[39] C. Lv, C. Niu, R. Gu, X. Jiang, Z. Wang, B. Liu, Z. Wu, Q. Yao, C. Huang, P. Huang, T. Huang, H. Shu, J. Song, B. Zou, P. Lan, G. Xu, F. Wu, S. Tang, F. Wu, and G. Chen, "Walle: An End-to-End, General-Purpose, and Large-Scale production system for Device-Cloud collaborative machine learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 249–265. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/lv

[40] S. Wang, G. Ananthanarayanan, and T. Mitra, "Optic: Optimizing collaborative cpu–gpu computing on mobile devices with thermal constraints," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 38, no. 3, pp. 393–406, 2018.