Characterizing Mobile SoC for Accelerating Heterogeneous LLM Inference

Le Chen[1†], Dahu Feng[1§], Erhu Feng[✉†], Yingrui Wang[‡], Rong Zhao[§], Yubin Xia[†], Pinjie Xu[2‡], Haibo Chen[†]

[†] *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*

[§] *Tsinghua University* [‡] *SenseTime Research*

Abstract

With the rapid advancement of artificial intelligence technologies such as ChatGPT, AI agents, and video generation, contemporary mobile systems have begun integrating these AI capabilities on local devices to enhance privacy and reduce response latency. To meet the computational demands of AI tasks, current mobile SoCs are equipped with diverse AI accelerators, including GPUs and Neural Processing Units (NPUs). However, there has not been a comprehensive characterization of these heterogeneous processors, and existing designs typically only leverage a single AI accelerator for LLM inference, leading to suboptimal use of computational resources and memory bandwidth.

In this paper, we first summarize key performance characteristics of heterogeneous processors, SoC memory bandwidth, etc. Drawing on these observations, we propose different heterogeneous parallel mechanisms to fully exploit both GPU and NPU computational power and memory bandwidth. We further design a fast synchronization mechanism between heterogeneous processors that leverages the unified memory architecture. By employing these techniques, we present HeteroInfer, the fastest LLM inference engine in mobile devices which supports GPU-NPU heterogeneous execution. Evaluation shows that HeteroInfer delivers a 1.34× to 6.02× end-to-end speedup over state-of-the-art GPU-only and NPU-only LLM engines, while maintaining negligible interference with other applications.

CCS Concepts: • **Computer systems organization** → **Heterogeneous (hybrid) systems**; **System on a chip**; • **Human-centered computing** → **Mobile computing**; • **Computing methodologies** → *Machine learning*.

SOSP '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1870-0/25/10
https://doi.org/10.1145/3731569.3764808

Keywords: Mobile System-on-Chip, Large Language Model, Heterogeneous Computing

ACM Reference Format:
Le Chen, Dahu Feng, Erhu Feng, Yingrui Wang, Rong Zhao, Yubin Xia, Pinjie Xu, Haibo Chen. 2025. Characterizing Mobile SoC for Accelerating Heterogeneous LLM Inference. In *ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3731569.3764808

1 INTRODUCTION

Driven by the rapid evolution in large language models (LLMs), technologies such as ChatGPT [3, 12, 32, 38], AI agents [15, 52, 53, 62], and video generation [14, 54, 61, 65] have gained widespread adoption. Concurrently, as users increasingly prioritize the privacy of their personal data, there is a growing trend towards executing model inference on local devices like smartphones. To enable the efficient calculation of large language models on these mobile platforms, contemporary mobile System-on-Chip (SoC) manufacturers have integrated various AI accelerators, including GPUs and neural processing units (NPUs). These accelerators [10, 16, 24, 25, 36, 57–59] enhance capabilities for vector and matrix computations, aligning with the computational demands of AI applications. For example, Qualcomm's SoCs incorporate Adreno GPUs [44] and Hexagon NPUs [45] to address the computing needs of edge AI applications.

Prior studies [17, 18, 23] have proposed inference engines to leverage the computation capabilities of heterogeneous processors. However, these solutions are predominantly tailored for cloud infrastructures and discrete accelerators, rendering them incompatible with mobile platforms. Although some researches [34, 55, 56] have explored mobile inference engines, they still fall short in fully exploiting the computational power of mobile heterogeneous SoCs. The limitation stems from three main factors:

First, the pronounced performance disparity between mobile NPUs and GPUs. For instance, on the Snapdragon 8 Gen 3 platform [41], the GPU achieves approximately 1 TFLOPS in practice (with a theoretical peak of 2.8 TFLOPS), while the NPU delivers up to 10 TFLOPS (in actual) performance.

[1]The two authors contributed equally to this work and should be considered co-first authors.
[2]Pinjie Xu is Project Leader.

Thus, enforcing parallel execution of the GPU and NPU solely based on their raw computational power may not guarantee an improvement in end-to-end performance. *Second*, synchronization overhead between heterogeneous processors is substantial. On mobile platforms, GPU-NPU synchronization during LLM inference can take around 400 microseconds, which is comparable to or even exceeds the execution time of individual GPU or NPU kernels. *Third*, the decoding phase is inherently memory-intensive. Merely offloading computation to GPU and NPU offers limited benefits and may even degrade performance due to additional synchronization overhead. Consequently, designing a LLM inference engine that can efficiently coordinate all heterogeneous processors in real-world mobile devices remains a substantial and unresolved challenge.

After an in-depth analysis of the heterogeneous processors within mobile SoCs, we observe new opportunities to enhance LLM inference by leveraging distinctive performance characteristics for heterogeneous processors.

- ***Tensor-sensitive NPU performance.*** Although NPUs can achieve superior performance under ideal conditions, their efficiency is highly dependent on tensor characteristics such as order, size, and shape. If the characteristics of tensors involved in computation are not well-aligned with the NPU's hardware architecture, both utilization and performance may degrade significantly.
- ***Unified memory architecture across processors.*** Unlike discrete accelerators, mobile CPUs, GPUs, and NPUs adopt a unified memory architecture (UMA) within the system memory, featuring a shared address space that facilitates efficient inter-processor communication.
- ***Memory bandwidth improved with multiple processors.*** A single processing unit is insufficient to fully saturate the SoC's memory bandwidth. For example, GPU alone can utilize only 40 ∼ 45 GB/s of memory bandwidth in memory-intensive workloads. In contrast, employing two processing units concurrently can achieve a memory bandwidth of about 60 GB/s (theoretical memory bandwidth in SoC is 68 GB/s).

In this paper, we introduce HeteroInfer, the fastest mobile inference engine, designed to efficiently leverage all heterogeneous processing units in mobile SoCs. The CPU is employed as a control plane for synchronization and GPU kernel scheduling, as it is ill-suited for computing tasks due to the low energy efficiency. The NPU serves as the primary computing unit, handling the majority of computing tasks, while the GPU acts as a secondary computing unit to enhance the lower bound of NPU performance. To efficiently leverage these heterogeneous computing resources, HeteroInfer takes comprehensive account of the performance characteristics of the GPU and NPU, such as stage performance, order-sensitive performance and shape-sensitive performance.

To enable both layer-level and tensor-level GPU-NPU parallelism on real-world mobile devices, HeteroInfer further introduces three techniques. First, HeteroInfer applies different tensor partitioning strategies during both the prefill and decoding phases to facilitate tensor-level heterogeneous execution. Second, HeteroInfer employs a fast synchronization mechanism based on predictable kernel waiting times to achieve microsecond-level synchronization. Third, HeteroInfer incorporates a tensor partitioning solver that generates optimal partitioning solutions with the help of a hardware profiler.

We built an industrial-grade LLM inference engine on the Snapdragon 8 Gen 3 SoC, one of the most advanced mobile platforms that integrates Arm CPU, GPU and NPU. To leverage both the GPU and NPU, we developed optimized GPU kernels using OpenCL and integrated the NPU operators provided by Qualcomm's QNN [43] into our framework. We avoid using the activation quantization and sparsity techniques, as these techniques may decrease the accuracy of the model inference.

HeteroInfer is the first LLM engine to surpass 1000 tokens/sec in prefill phase and 50 tokens/sec in decoding phase, utilizing high-precision computation on mobile devices for billion-parameter scale LLMs. In end-to-end evaluations, HeteroInfer achieves a speedup ranging from 1.34× to 6.02× compared to SOTA GPU-only and NPU-only frameworks across various ML workloads. During the prefill phase, HeteroInfer accelerates the speed up to 3.69× over PI-2 [34] (NPU) and 8.68× compared to MNN [55] (GPU). When the sequence length does not align with the NPU graph shape, HeteroInfer delivers up to a 2.12× improvement over traditional padding-based methods. In the decoding phase, HeteroInfer consistently outperforms existing frameworks, achieving speedups between 1.50× and 2.53×. To evaluate performance interference, we run HeteroInfer concurrently with GPU-intensive workloads such as gaming. HeteroInfer effectively minimizes interference between LLM and gaming tasks, maintains stable frame rates (without FPS drop), and incurs only a 2.2% slowdown in the prefill phase and 17.7% during decoding.

2 BACKGROUND & RELATED WORK

2.1 LLM Inference

Large Language Model (LLM) inference refers to the process of utilizing a pre-trained model to generate outputs based on user input. It typically comprises two distinct phases: the prefill phase and the decoding phase. In the prefill phase, the LLM processes the user's input in a single batch to generate the first output token. Given the potentially long input sequence, this phase relies on matrix multiplication operations, rendering it computationally intensive. In contrast, decoding is a sequential and auto-regressive process, generating one token at a time. It primarily involves matrix-vector multiplications, resulting in a memory-intensive workload.

Table 1. We summarize the functionalities and limitations of current mobile-side inference engines as follows: **CPU, GPU, NPU** indicate support for various backends, accommodating both integer and floating operations. **Accuracy** indicates whether the model's accuracy is consistent with the original model.

Framework	CPU		GPU		NPU		NPU GEMM Type	Prefill	Decoding	Accuracy	Performance
	INT	FP	INT	FP	INT	FP					
llm.npu [56]	INT4	FP16/32	/	/	INT8	/	INT	CPU + NPU	CPU	Decrease (per dataset quantization)	High
Powerinfer2 [60]	/	W4A16	/	/	INT4	W4A16	INT / FLOAT	CPU + NPU	CPU	Decrease (change model structure)	High / Medium
Qualcomm-AI [42]	INT4/8	W4A16	/	FP16	INT4/8	/	INT	NPU	NPU	Decrease	High
MLC [34]	/	W4A16	/	W4A16	/	/	/	CPU / GPU	CPU / GPU	Not affect	Low
Llama.cpp [11]	INT4/8	W4A16	/	W4A16	/	/	/	CPU / GPU	CPU / GPU	Not affect	Low
Onnxruntime [33]	/	FP16/32	/	/	INT8/16	/	INT	CPU / NPU	CPU / NPU	Decrease	Medium
MNN-LLM [55]	INT8	W4A16	/	W4A16	/	/	/	CPU / GPU	CPU / GPU	Not affect	Medium
Ours	INT8	W4A16	INT8	W4A16	INT4/8	W4A16	FLOAT	GPU + NPU	GPU + NPU	Not affect	High

If the framework supports multiple quantization methods, list only the common ones. "W4A16" indicates that weights are stored as INT4 and computations are performed in FP16.

In contrast to cloud-side LLM inference (e.g. vLLMs [26], orca [63], etc. [2, 9, 28, 40, 48, 64, 67, 69]), which prioritizes high throughput as well as meeting the response-time Service-Level Objectives (SLOs) of different inference workloads, mobile-side LLM inference places a greater emphasis on minimizing end-to-end latency. The latency can be further divided into two parts: TTFT (Time to First Token) and TPOT (Time per Output Token). The former is primarily influenced by the speed of prefill phase processing, and the latter is associated with the token generation speed during the decoding phase.

2.2 Mobile-side Heterogeneous SoC

Considering the imperatives of personal privacy and security, there is a growing preference among individuals to deploy LLMs on local mobile devices instead of transmitting personal data to cloud services. Consequently, the mainstream vendors are actively enhancing their Edge-AI platform evolution, including mobile platforms such as Qualcomm's Snapdragon 8 Gen 3 [41], Apple's A18 [4], MediaTek's Dimensity 9300 [4], Huawei's Kirin 9000 [19], etc. Table 2 lists the parameter specifications of several mainstream mobile SoC platforms. To meet the substantial computational demands of LLMs, mobile platforms are evolving towards heterogeneous SoC architectures. In addition to traditional CPUs and GPUs, NPUs, which are dedicated processors optimized for ML workloads, are increasingly playing a critical role due to their superior computational power compared to CPUs and GPUs. Moreover, these heterogeneous processors often utilize a unified physical memory, distinguishing them from discrete heterogeneous systems.

Table 2. Specifications [35] of Mobile-side Heterogeneous SoC of mainstream vendors.

Vendor	SoC	GPU	GPU FP16	NPU	NPU INT8	NPU FP16
Qualcomm	8 Gen 3	Adreno 750	2.8 TFlops	Hexagon	34 Tops	17 TFlops
MTK	K9300	Mali-G720	4.0 TFlops	APU 790	48 Tops	24 TFlops
Apple	A18	Bionic GPU	1.8 TFlops	Neural Engine	35 Tops	17 TFlops
Nvidia	Orin	Ampere GPU	10 TFlops	DLA	87 Tops	None
Tesla	FSD	FSD GPU	0.6 TFlops	FSD D1	73 Tops	None

NPU FP16: Since the vendors have not disclosed the computational power of the NPU for FP16, we roughly estimate it to be half of the INT8 computational power.

2.3 Limitations of Mobile-side Inference Engines

With the development of heterogeneous mobile SoCs, numerous works have established frameworks for LLM inference to leverage the heterogeneous computational power at the edge, such as llm.npu [56], PowerInfer2 [60], MNN-LLM [55], MLC [34], etc [11, 13, 21, 31, 33, 51]. In Table 1, we provide a comprehensive comparison of existing approaches across multiple dimensions, including support for different backends, impact on model accuracy and end-to-end performance of LLM inference. Through this comparison, we identify two key aspects that existing works fail to address:

Parallelizing computation between GPU and NPU. Although both GPUs and NPUs can accelerate the computation of LLM workloads and exhibit varying affinities for different operators, existing research primarily focuses on leveraging a single accelerator within mobile SoC. For instance, MLC [34] and MNN-LLM [55] exclusively utilize the GPU for computation, whereas PowerInfer [60] and llm.npu [56] are limited to supporting the NPU backend. This challenge stems from the difficulty involved in synchronization and coordination among heterogeneous processors. Specifically, it is crucial to address how to efficiently synchronize data and effectively partition computational workloads across different backends.

Fully utilizing NPU computational power without accuracy loss. Being the most powerful computing unit on modern SoCs, NPUs have been utilized by frameworks such as llm.npu [56], Qualcomm-AI [42] to accelerate the inference process. However, these approaches primarily leverage the INT capabilities of NPUs for low-precision computations and do not fully exploit the available FLOAT capabilities. Utilizing INT4/8 for computations can lead to significant degradation in model accuracy (20% reported by Qualcomm-AI), especially for small-sized edge models. Although llm.npu seeks to address this issue by identifying tensor outliers and employing mixed-precision techniques, this approach introduces sensitivity to the input dataset, resulting in inference accuracy that is highly dependent on the user's input. Previous work [60] also tries to utilize the FLOAT capabilities of NPUs but often yields inefficient performance. Due to the

inherent hardware structure of NPUs, FLOAT performance can vary significantly based on tensor shape and ordering.

3 PERFORMANCE CHARACTERISTICS

To effectively utilize the heterogeneous processors, we start by analyzing the performance of the GPU, NPU and memory system. These accelerators exhibit diverse performance characteristics stemming from their unique hardware architectures. In particular, NPUs display significant performance variability across different tensor types and operators. Therefore, we conduct a comprehensive analysis of the architectural differences among these heterogeneous accelerators, and then summarize their performance characteristics.

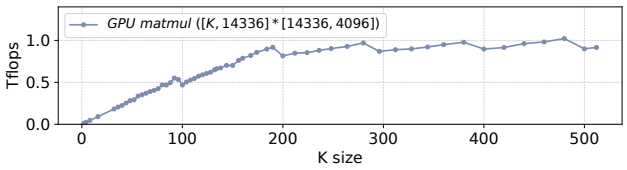

Figure 1. The GPU performance with varying tensor sizes.

3.1 GPU Characteristics

Characteristic GPU-1: Linear Performance. Figure 1 illustrates the performance of mobile GPUs with varying tensor sizes. When the tensor size is small, GPU computation is memory-bound. With the increase of tensor size, the total FLOPS increases linearly. Once the size surpasses a certain threshold, GPU computation turns to be computation-bound, and the total FLOPS stays stable.

Characteristic GPU-2: High-cost Synchronization. There are two primary types of synchronization overheads associated with mobile GPUs. The first type arises from the data copy. Since existing GPU frameworks still maintain a separate memory space for mobile GPUs, developers must utilize APIs such as `clEnqueueWriteBuffer` to transfer data from the CPU-side buffer to GPU memory. Secondly, the explicit synchronization command (`clFinish`) introduces a fixed latency of approximately 400 microseconds on our platform. This overhead results from blocking the execution of subsequent GPU kernels and synchronizing the states of the GPU driver. Upon completing the data transfer and executing the synchronization command, it ensures that all states are consistent between the CPU and GPU views, thereby enabling the consistent execution of subsequent CPU/GPU/NPU tasks.

3.2 NPU Characteristics

Although there are many different NPU implementations, matrix computation units (e.g., systolic arrays) serve as the most critical component inside NPUs. It leverages the data flow characteristics intrinsic to matrix computations, thereby minimizing the redundant load/store operations of model weights and activation. Figure 2 demonstrates a classical

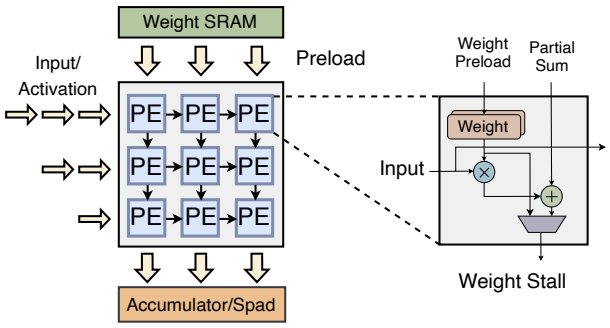

Figure 2. The hardware design for NPU: Systolic array with the weight stall computing paradigm.

systolic array design. In the computing flow of systolic array, weights are preloaded into each processing element (PE) prior to the computation. During the computation phase, a 'weight stall' mode is employed, where weights remain stationary while inputs or activations are fed into the systolic array. Finally, the computation results are output from the systolic array and are either stored in on-chip SRAM or directly forwarded to the subsequent systolic array unit. Due to this NPU computation paradigm, NPUs exhibit three distinct computational characteristics: stage performance, order-sensitive performance and shape-sensitive performance.

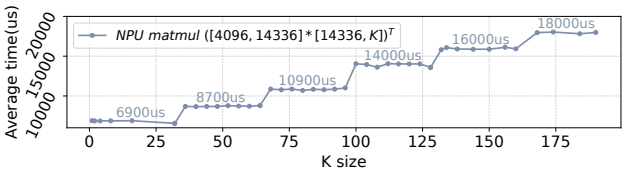

Figure 3. The stage performance of NPUs. The execution time of the Matmul operator across different tensor sizes.

Characteristic NPU-1: Stage Performance. Due to the fixed size of the hardware systolic array within NPUs, the dimensions of the tensor used by the Matmul operator may not align with the size of the matrix computation unit, which can lead to inefficient use of NPU computational resources. As shown in Figure 3, this misalignment results in a phenomenon referred to as *stage performance* across different tensor sizes. For instance, in the Snapdragon 8 Gen 3 SoC, NPU is equipped with multiple 32×32 systolic arrays. Thus, any computing tensor with dimensions smaller than 32 will exhibit the same computational latency, leading to significant performance degradation for certain tensor shapes. Furthermore, the compiler will partition tensors into tiles that align with the hardware size of the matrix computation unit. When tensor dimensions are not divisible by this size, the NPU compiler must introduce internal padding to align with the hardware requirement. This alignment results in a stage performance effect during NPU calculations.

Characteristic NPU-2: Order-sensitive Performance. In addition to stage performance, NPUs also exhibit order-sensitive computation behavior. Consider two tensors with

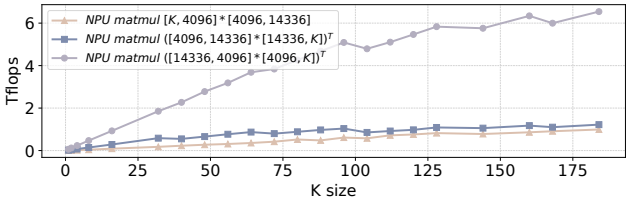

Figure 4. The order-sensitive and shape-sensitive performance of NPUs. The performance of the NPU is significantly influenced by the order and shape of the tensors.

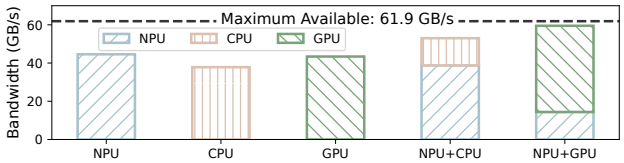

Figure 5. The total memory bandwidth with single and multiple processors. We execute the decoding workloads across different backends and measure the available memory bandwidth in the entire SoC.

dimensions $[M, N]$ and $[N, K]$, where $M > N > K$. A conventional matrix multiplication (Matmul) operation requires $2 \times M \times N \times K$ operations. If we reverse the order of these tensors, i.e., $[K, N] \times [N, M]$, the total number of computation operations remains unchanged. However, this can lead to significant performance degradation for the NPU, a phenomenon we refer to as *order-sensitive performance*. Figure 4 presents a specific example where the matrix multiplication operation of $[14336, 4096] \times [4096, K]$ achieves 6× performance improvement compared to $[K, 4096] \times [4096, 14336]$.

The primary reason for order-sensitive performance is that NPU leverages the weight stall computing to minimize memory load/store overhead. In the ideal situation, the weight tensor always fits perfectly within the hardware matrix computation unit, eliminating the need for additional memory operations. However, when the weight tensor is significantly larger than the input tensor, it needs to load the weight tensor from the memory into the matrix computation unit more frequently, increasing memory overhead during the NPU execution. As a result, although $[K, N] \times [N, M]$ and $[M, N] \times [N, K]$ involve the same number of computational operations, the larger size of $[N, M]$ results in inferior performance due to the extra memory operations involved. In the worst-case scenario for NPU computation, if the weight matrix is infinite, the matrix computation unit cannot take advantage of the weight-stall computing paradigm, and thus, the NPU performance may regresses to or even worse than the GPU performance.

Characteristic NPU-3: Shape-sensitive Performance. Similar to the order-sensitive performance, NPUs also exhibit *shape-sensitive performance* characteristic. Even when the input tensor is larger than the weight tensor, the NPU's efficiency is influenced by the ratio between row and column sizes. More specifically, when the row size of the input tensor exceeds the column size, NPU demonstrates a better performance (compare the blue line with the purple line in Figure 4). Since the column size of the input tensor is shared with the weight tensor, a larger column size results in a larger weight tensor, which undermines the advantages of the weight-stall computation paradigm as we mentioned above.

3.3 SoC Memory Bandwidth

Characteristic Memory-1: Underutilized Memory Bandwidth with Single Processor. Although mobile SoCs adopt a unified memory address space across heterogeneous processors, our measurements indicate that no single processor can saturate the SoC's memory bandwidth during LLM decoding. On the Snapdragon 8 Gen 3 platform, the theoretical peak memory bandwidth is 68 GB/s, while the maximum achievable bandwidth is approximately 61.9 GB/s, obtained only under continuous large-bulk memory operations (e.g., multi-threaded memcpy or NEON-load intrinsics on the CPU, and vector or image-buffer loads on the GPU). However, under decoding workloads, each individual processor (CPU, GPU, or NPU) achieves only 40–45 GB/s, as shown in Figure 5. In contrast, concurrent GPU–NPU execution raises aggregate bandwidth utilization to around 60 GB/s, with a 75/25 workload split that effectively overlaps GPU computation with NPU computation and synchronization overhead. These results suggest that, as contemporary mobile SoCs are equipped with multiple memory channels, co-execution of heterogeneous processors more fully utilizes the SoC's memory bandwidth, creating a new opportunity to accelerate memory-bound LLM decoding.

4 DESIGN

Given the distinct bottlenecks at different stages of LLM inference, our system adopts a stage-specific optimization strategy: during the prefill stage, the objective is to maximize the computational throughput of the SoC, whereas during the decoding stage, the focus shifts to maximize memory bandwidth utilization. Figure 6 illustrates the overall execution flow of a typical LLM using HeteroInfer. Considering the power constraints of mobile systems and the need to accommodate other concurrently running applications, we refrain from fully consuming all available computing units solely for LLM tasks. For instance, CPUs, in particular, are not ideal as computational backends due to their relatively low energy efficiency and, more critically, their significant involvement in general-purpose processing. Therefore, HeteroInfer leverages the CPU only as a control plane to handle tasks such as synchronization and GPU kernel scheduling.

4.1 Layer-level GPU-NPU Execution

We begin with a coarse-grained strategy that assigns computation tasks to the GPU and NPU based on their computing affinities at the layer level. For example, during the prefill phase, Matmul operators are allocated to the NPU,

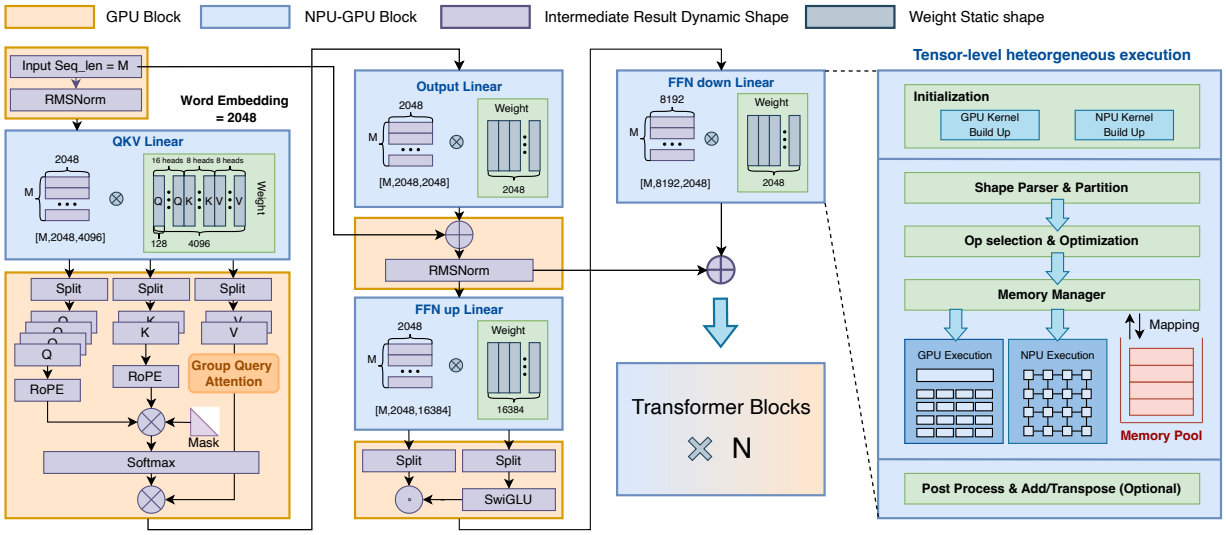

Figure 6. The overall execution flow of a typical LLM with HeteroInfer: Operators within the orange blocks are executed on the GPU backend, whereas operators within the blue blocks can be offloaded to heterogeneous backends. HeteroInfer additionally supports tensor-level heterogeneous execution through different tensor partitioning strategies.

which demonstrates superior performance in matrix multiplication computing operations. In contrast, RMSNorm and SwiGLU operators are executed more efficiently on the GPU. Besides, given that LLM models typically have larger weight tensors compared to user input tensors, we adapt to the characteristics of *NPU-2: order-sensitive performance* by exchanging the computation order of the input and the weight tensor, utilizing the following computational invariant: $[M, N] \times [N, K] \rightarrow [[K, N] \times [N, M]]^T$. As for the decoding phase, due to *NPU-1: stage performance*, the GPU becomes the primary computational unit, as it offers better performance in matrix–vector operations.

4.2 Tensor-level GPU-NPU Parallelism

While the layer-level approach leverages both the GPU and the NPU to accelerate LLM inference, it falls short of fully utilizing the capabilities of a heterogeneous SoC due to three key limitations:

- NPU performance degradation for specific tensor shapes.
- Underutilization of SoC memory bandwidth and computational power of heterogeneous processors.
- Static NPU graphs with high graph generation costs.

To overcome these limitations, HeteroInfer introduces tensor-level parallel execution, enabled by three different partitioning strategies.

4.2.1 Weight-centric partition with static shape

Problem. When considering different tensor shapes, the NPU fails to outperform the GPU in all situations. This unexpected performance result arises from two factors: First, when the sequence length is short, the NPU cannot fully utilize all available computational resources due to *NPU-1: stage performance*, resulting in performance that is similar to or even lower than the GPU's. Second, due to the dimensionality reduction matrix in the FFN-down layer, the column size of this matrix is larger than the row size (after transposition). This configuration is suboptimal for NPU execution, yielding only a 0.5× to 1.5× performance over the GPU, attributable to *NPU-3: shape-sensitive performance*.

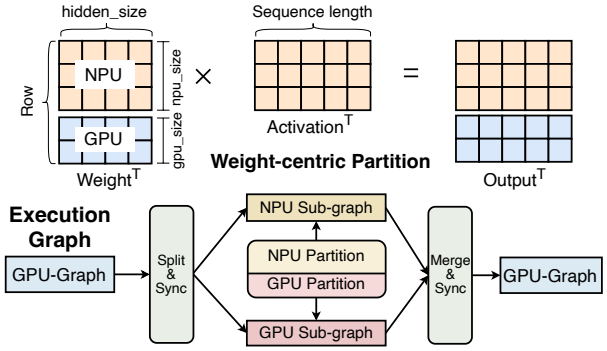

Figure 7. HeteroInfer partitions the weight tensor based on the row dimension, and dispatches computational loads to the different backends.

To address these performance limitations, we propose a *weight-centric partitioning* strategy to enable parallel execution between the GPU and NPU. To simplify the parallel strategy, we start by considering a fixed sequence length for user input, enabling the NPU to construct the computation graph based on static tensor shapes. As illustrated in Figure 7, this strategy partitions the weight tensor along its row dimension, assigning sub-tensors to the GPU and NPU for concurrent computation. The partition ratio is statically determined by an offline solver (described in §4.4).

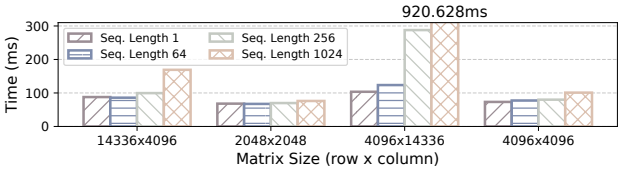

Figure 8. The NPU graph generation time for a single operator across various tensor shapes.

HeteroInfer explicitly incorporates synchronization points into the computation graph and manages the splitting and merging of intermediate results across different backends. With an ideal partition ratio, the GPU and NPU complete their respective workloads simultaneously, thereby minimizing end-to-end inference latency.

Although the same partitioning strategy is applied, the rationale behind acceleration in the prefill and decoding phases differs. During the prefill phase, the partitioning strategy leverages the GPU's computational resources to accelerate computation—especially in scenarios where the NPU suffers from performance degradation due to unfavorable tensor shapes. In contrast, during the decoding phase, the partitioning strategy is designed to address the underutilization of memory bandwidth when using a single computing unit (*Memory-1*). This approach focuses on maximizing the memory bandwidth of the SoC while minimizing memory contention through GPU-NPU parallelism.

4.2.2 Activation-centric partition with dynamic shape

Problem. In real-world scenarios, user input typically has a dynamic sequence length, while current mobile-side NPUs only support static graph execution. This limitation arises from the dataflow graph compilation method [1, 6, 50], which is widely adopted by contemporary mobile NPUs [20, 43]. To address this issue, a straightforward solution is to generate the computation graph at runtime. However, this graph generation process incurs a non-trivial overhead that is proportional to the size of the tensors [46, 66, 68], as shown in Figure 8. In contrast, the GPU framework offers a set of kernel implementations, each of which can be adapted to accommodate a variety of tensor shapes. This capability facilitates dynamic-shape kernel execution at runtime.

In order to support dynamic activation shapes on the NPU, a common approach is to select a set of predefined tensor shapes—such as powers of two—and pad the activation tensors to match these shapes. For example, if the input sequence length is 300, the inference engine pads it to 512 to avoid the overhead of generating a new NPU graph for the unmatched tensor shape. However, this padding introduces computational redundancy.

To mitigate this issue, we propose an *activation-centric partitioning* strategy that enables support for dynamic tensor shapes while minimizing the overhead caused by excessive padding. As shown in Figure 9, HeteroInfer offloads computation tasks involving dynamic-shape tensors to the GPU (according to *GPU-1: linear performance*), while retaining fixed-size tensor computations on the NPU. For instance, consider an activation tensor with a sequence length of 300, which can be partitioned into two segments: 44 and 256. Since the segment of size 256 adheres to a standard shape, its computation graph is pre-generated and can be executed directly on the NPU. In contrast, the remaining segment of size 44 does not conform to a predefined shape for the NPU backend; therefore, it can be processed by the GPU backend in parallel with the NPU.

To further balance the computational load between the NPU and GPU, we adopt a **multi-tensor activation partitioning** strategy. Considering that the GPU generally exhibits lower performance than the NPU, we partition the activation tensor along the sequence length dimension into multiple sub-tensors, each conforming to a standard shape, along with one additional sub-tensor that has an arbitrary shape. All sub-tensors with standard shapes are executed sequentially on the NPU, while the sub-tensor with a dynamic shape is offloaded to the GPU. This approach allows us to minimize the computational load on the GPU and effectively balance the execution time between the GPU and NPU.

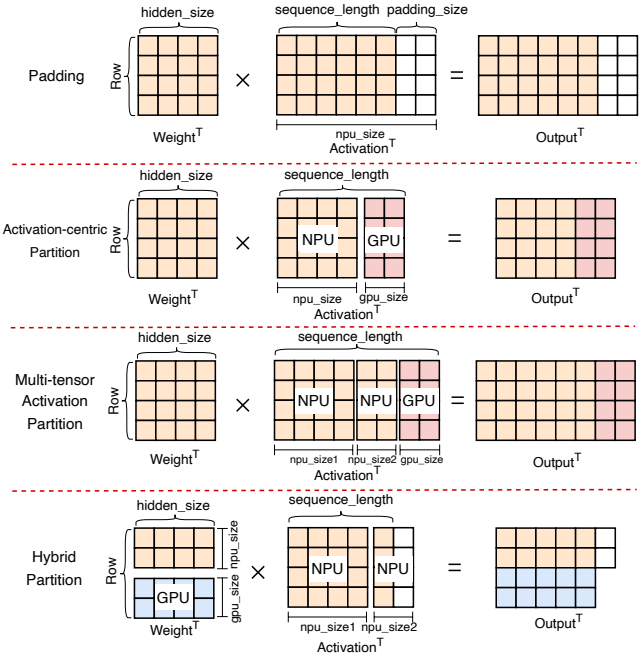

Figure 9. Due to the static computation graph of the NPU, HeteroInfer employs activation-centric partition to support the dynamic tensor shapes required by LLMs.

4.2.3 Hybrid partition

Problem. Although the activation-centric partitioning strategy facilitates support for dynamic tensor shapes, it results in suboptimal utilization of the computational resources of both the GPU and NPU due to *NPU-3: shape-sensitive performance*. Specifically, the sub-tensor offloaded to the GPU

may be either too large or too small to fully exploit its computational capabilities, or the tensor shape executed on the NPU may not be suitable for efficient NPU computation.

To address this issue, we propose a *hybrid partitioning* strategy that combines activation-centric partition with padding to handle dynamic tensor shapes, while leveraging weight-centric partition to offload tensors to both GPU and NPU, as illustrated in Figure 9. Given that the weight tensor is typically larger than the activation tensors, it allows for greater flexibility in partitioning the tensor into a more suitable shape for NPU computation. Consequently, the hybrid partitioning strategy not only maximizes the computational power of heterogeneous processors but also maintains system flexibility in accommodating dynamic user inputs.

4.3 Fast Synchronization

While GPU-NPU parallelism can reduce the execution time of certain operators, it may also introduce additional overhead due to synchronization between the GPU and NPU (*GPU-2: high-cost synchronization*). This issue arises because existing mechanisms, such as fences and `clFinish`, are not optimized for mobile SoCs. The overhead becomes particularly pronounced during the decoding phase, where the execution time of the Matmul operator is reduced to only a few hundred microseconds.

To mitigate this overhead, we employ two strategies. First, mobile SoCs provide a unified address space, which allows mapping a memory buffer into both host and device address spaces, eliminating the need for additional data transfers. In the HeteroInfer runtime, a dedicated memory pool is reserved for allocating the input and output tensors of each operator. Since different layers in LLMs share the same decoder block, this memory pool requires only a few buffer slots, which can be reused across layers. Furthermore, these buffer slots will not be reclaimed by the GPU / NPU driver, ensuring that the mapping between CPU and GPU / NPU address spaces is maintained throughout model inference.

Second, we exploit the predictable waiting times for GPU kernels to facilitate fast synchronization. Given that LLMs execute identical operations across each layer, the waiting times for GPU kernels tend to be consistent and predictable across different layers. We allow the synchronization thread to sleep for a predicted waiting time, followed by a polling mechanism to achieve precise synchronization. Since the minimum granularity of 'usleep' in mobile SoCs is approximately 80 to 100 microseconds, it cannot serve as an accurate synchronization mechanism. Consequently, once the synchronization thread awakens, it utilizes a small/middle CPU core to continuously monitor the output tensor of the last layer. A flag bit is added alongside the output tensor and is updated once the output tensor is completely populated. The CPU core only needs to poll this flag bit for a few microseconds and can immediately notify the NPU for subsequent execution as soon as the GPU kernel completes.

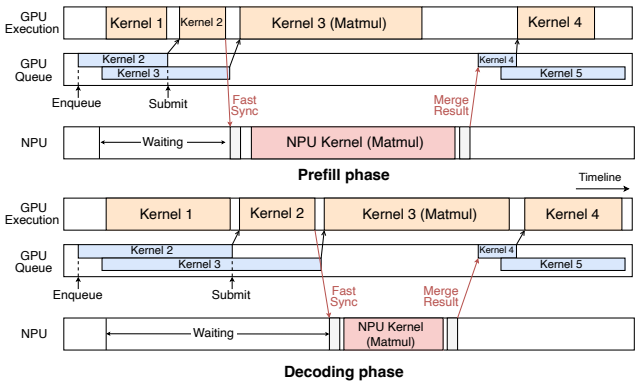

Figure 10. Fast synchronization during the prefill and decoding phases.

While both the prefill and decoding phases leverage GPU and NPU parallelism with fast synchronization, several distinctions exist between these two phases, as shown in Figure 10. In the prefill phase, the NPU exhibits superior computational capability, making it NPU-dominant. HeteroInfer effectively hides GPU execution time within NPU execution, but needs to delay the submission of the next GPU kernel until NPU execution is finished. Although this introduces a task submission overhead during GPU-NPU synchronization, this cost is approximately tens of microseconds and can thus be ignored in the prefill phase.

Conversely, in the decoding phase, the GPU outperforms the NPU because GPU kernel implementations obtain more stable and higher memory bandwidth, making this phase GPU-dominant. Here, we overlap NPU execution with the GPU execution. Upon NPU task completion, the subsequent GPU kernel is promptly enqueued. The GPU's inherent queue ordering ensures correct synchronization for GPU kernels without incurring additional submission overhead.

4.4 Putting It All Together

Figure 11 presents the overall architecture of HeteroInfer. Given a target SoC, the performance profiler first measures its performance matrices by executing operations on the actual NPU or GPU. The profiling results are then fed to the solver, which determines how to schedule and partition the workload across processors for running a given LLM on the specified SoC. The solver's output are subsequently used to generate the computation graph in advance. This entire process is performed offline. During online inference, the inference engine dynamically selects an appropriate execution strategy based on the requested sequence length.

Performance Profiler. The profiler executes the target operator with a variety of tensor shapes on both GPU and NPU to collect accurate performance metrics, including execution time, memory bandwidth, and synchronization overhead. The profiling space is constrained by three factors: (1) only weight tensor shapes from LLMs are considered; (2)

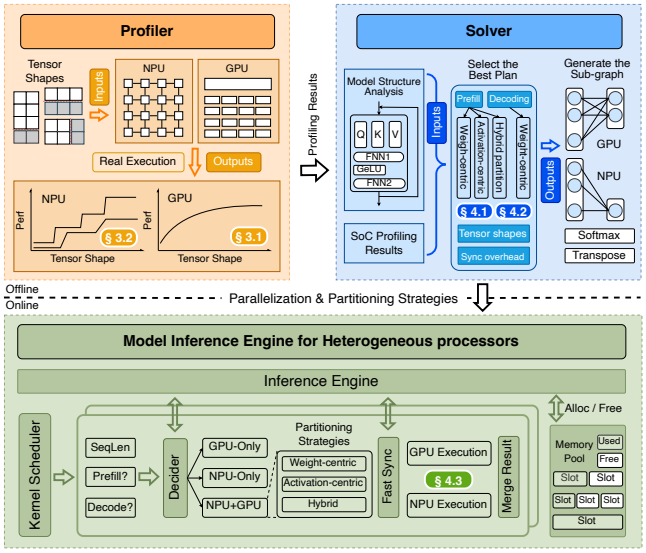

Figure 11. The overall architecture of HeteroInfer contains three components: Profiler, Solver and Inference Engine.

the NPU's stage performance characteristic impose minimum size requirements on sub-tensors; and (3) activation tensors are restricted to a predefined set of standard sequence lengths. Benefiting from this reduced set of tensor shapes, the profiling process can be completed efficiently, typically in less than 20 minutes.

Tensor Partitioning Solver. The solver determines the optimal tensor partitioning strategies for a given LLM based on profiling results. It first analyzes the overall model structure to locate operators where workloads can be partitioned across heterogeneous processors, such as those involved in attention projections and the FFN up/gate/down computations. For each operator, the solver enumerates all feasible parallelization strategies and selects the one that best overlaps computation with synchronization to minimize total latency, as formulated in the following equation. Since the prefill sequence length may vary arbitrarily while the profiler only provides measurements for standard sequence lengths, the solver estimates operator latency for variable-length sequences by leveraging the performance characteristics of *GPU-1: linear performance* and *NPU-1: stage performance.*

$$T_{\text{total}} = \min\Big(\max(T_{\text{GPU}}^{\text{partition1}}, T_{\text{NPU}}^{\text{partition2}}) + T_{\text{sync}} + T_{\text{copy}},$$
$$T_{\text{GPU}}^{\text{all}}, T_{\text{NPU}}^{\text{all}} + T_{\text{sync}} + T_{\text{copy}}\Big)$$
$$\text{s.t.} \quad \text{Partition1} + \text{Partition2} = \text{All}.$$

Table 3 illustrates an example of the solver's inputs and outputs. For tensor shapes in the decoding phase, a weight-centric partitioning strategy is adopted, with the GPU performing the majority of computation. This is because GPU usually outperforms NPU in the matrix-vector multiplication operation. While for tensor shapes in the prefill phase, the optimal partitioning strategy is more variable and shape-dependent. For instance, with a weight tensor of shape [4096, 4096], where a significant performance gap exists between GPU and NPU, an activation-centric partition is applied when the input sequence length falls within the range of 257-272. In contrast, for a weight tensor of shape [4096, 14336], the computational capabilities of the NPU and GPU are relatively comparable (approximately 3:2), primarily due to *NPU-3: shape-sensitive performance.* When the dynamic portion is relatively small—i.e., the prefill length only slightly exceeds a standard length—activation-centric partition may result in under-utilization of GPU resources. In such cases, a hybrid partitioning strategy is preferred.

Table 3. Input and output examples of the solver.

Weight Tensor Shape	Activation Tensor Shape	Latency (us)		Partitioning Strategy	Partition Ratio (GPU : NPU)
		GPU	NPU		
4096, 4096	4096, 1	511	693	Weight-centric	1 : 1
28672, 4096	4096, 1	1903	3886	Weight-centric	3 : 1
4096, 14336	14336, 1	1467	6506	No partition	GPU-only
4096, 4096	4096, 128	7306	912	No partition	NPU-only
4096, 4096	4096, [193-255]	$9k \sim 11k$	$1.2k \sim 1.9k$	Padding	NPU-only
4096, 4096	4096, 256	10841	1884	No partition	NPU-only
4096, 4096	4096, [257-272]	~ 11000	~ 1900	Activation-centric	Dynamic : 256
4096, 14336	14336, 256	35231	23445	Weight-centric	2 : 3
4096, 14336	14336, [257-384]	$35k \sim 50k$	$23k \sim 32k$	Hybrid	2 : 3 (Weight)

Latency values for tensor shapes with dynamic ranges are estimated using the solver's latency function.

Inference Engine. During execution, a control plane decider determines whether a kernel is executed on the NPU backend, the GPU backend, or using GPU-NPU parallelism, based on the solver's output and current states. When two adjacent kernels are allocated to different backends, the inference engine employs a fast synchronization mechanism to ensure data consistency. Upon completion of kernel execution on both backends, it merges intermediate results as needed. Besides, the inference engine manages a memory pool for host-device shared buffers, which are allocated or reclaimed as input and output tensors for each GPU/NPU kernel, bypassing the organization of the device driver.

5 EVALUATION

5.1 Experimental Setup

We implemented an industrial-grade edge LLM engine: HeteroInfer, by developing optimized GPU kernels in OpenCL and incorporating NPU support via the QNN-NPU library. The system supports both layer-level and tensor-level heterogeneous execution across the GPU and NPU. As for the model quantization, HeteroInfer employs the W4A16 (weight-only) quantization[1] [7, 8, 27, 30, 39] which balances the model accuracy (FLOAT computation) and storage overhead (INT4 for weight storage). We evaluate the performance of HeteroInfer on both Snapdragon 8 Gen 3 and Snapdragon 8 Elite, two of the most powerful mobile SoCs. For fair comparison with other works, all results in this paper are obtained on Snapdragon 8 Gen 3 unless otherwise noted. Compared to previous approaches [42, 56, 60] that leverage low-precision

[1]Only utilizes NPU's TOPS during the decoding phase, as NPU currently does not support W4A16 for decoding.

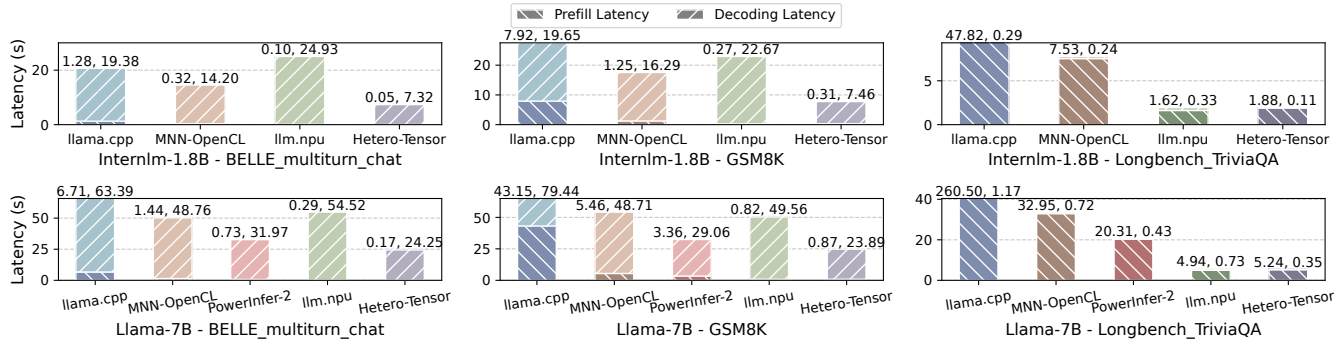

Figure 12. End-to-end latency comparison using real-world datasets on different models. The numbers labeled on each bar represent (prefill latency, decoding latency).

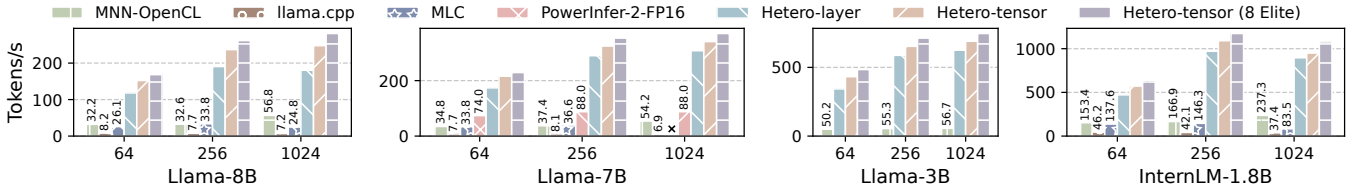

Figure 13. Prefill speed of MNN-OpenCL, llama.cpp, MLC, PowerInfer-2-FP16 and HeteroInfer across sequence lengths of 64, 256 and 1024 on multiple models.

computation and exploit model sparsity, HeteroInfer maintains the model accuracy and demonstrates even better performance through more efficient NPU utilization and GPU-NPU parallelism.

In our evaluation, we compare HeteroInfer with representative mobile LLM inference engines that utilize GPU-only and NPU-only accelerators, including llama.cpp (CPU), MLC (GPU), MNN (GPU), llm.npu (NPU), and powerinfer-2 (NPU). We use batch size of 1 across all experiments, as concurrent requests are not yet common in current mobile-device deployment scenarios. For end-to-end LLM workload tests conducted on the mobile platform, HeteroInfer demonstrates a performance improvement ranging from 1.34× to 6.02× across various workloads when compared to other SOTA solutions. More specifically, in the prefill stage, HeteroInfer achieves a performance enhancement of 3.29× to 24.9×, whereas in the decoding stage, it attains a performance improvement of 1.50× to 2.53×.

5.2 End-to-End Performance on Mobile Platform

Figure 12 compares the end-to-end latency of Hetero-tensor with other edge LLM engines across three representative tasks: multi-turn dialogue [29], simple QA [37], and long-text processing [5] (detailed configurations in Table 4). In the decoding-heavy multi-turn dialogue task with Llama-7B, Hetero-tensor achieves a 2.06× and 3.40× speedup over MNN and llm.npu, respectively, and even outperforms PowerInfer-2 (with sparse model) by 1.34×. This is enabled by optimized GPU kernels and efficient GPU-NPU parallelism that fully utilize SoC memory bandwidth. On GSM8K, where prefill and decoding are balanced, Hetero-tensor yields a 2.62× average improvement. In prefill-dominant workloads (Longbench), it delivers up to 6.02× speedup over MNN-OpenCL. Notably, Hetero-tensor still matches or surpasses llm.npu performance under such workloads, despite the latter utilizes mixed-precision computation which requires per-dataset quantization and may cause accuracy loss.

To better analyze the performance of HeteroInfer, we conduct detailed tests of both prefill and decoding stages.

Table 4. End-to-end benchmark configurations.

Task type	Dataset	Mean prefill tokens	Mean decoding tokens
Multi-turn dialogue	BELLE multiturn chat	54	374
Simple QA	GSM8K	296	340
Long text processing	LongBench-TriviaQA	1787	5

5.3 Prefill Performance

The evaluation of the prefill performance is conducted from two perspectives: fixed sequence lengths that fit in pre-defined graphs and dynamic sequence lengths that do not align with the NPU's static graph.

5.3.1 Fixed Sequence Length. Figure 13 compares the prefill performance across different frameworks with several fixed sequence lengths. For Llama-8B with sequence length of 256, Hetero-layer achieves 5.85×, 5.64× and 24.9× speedup compared to MNN-OpenCL, MLC and llama.cpp. Even compared to the NPU-only engine like PowerInfer-2-FP16, Hetero-layer achieves 3.29× speedup. These performance improvements arise from optimization mechanisms, including considerations of the affinity of NPU and GPU for different operators, as well as equivalent tensor order exchanges for Matmul computations.

Based on this, Hetero-tensor takes a step further and outperforms Hetero-layer by 30.2% on average (up to 40.8% when sequence length is 32). The prefill speed of Hetero-tensor reaches 247.9 tokens per second on Llama-8B and is up

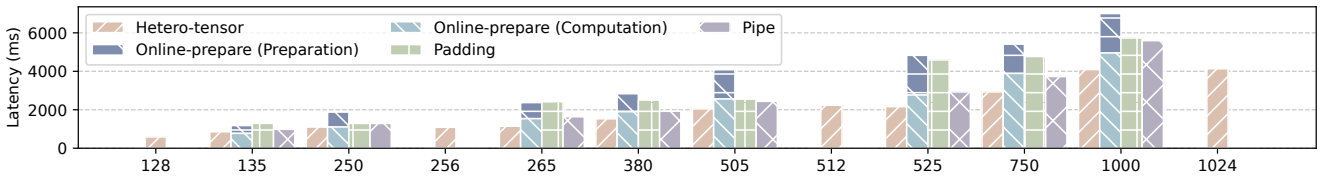

Figure 14. Prefill latency of Online-prepare, Padding, Pipe and Hetero-tensor on Llama-8B with dynamic sequence lengths.

to 1092 tokens per second on InternLM-1.8B. This is because Hetero-tensor enables GPU-NPU parallelism by partitioning the weight and activation tensors (especially for FFN-down) into shapes that are suitable for NPU computation, while offloading the remaining part for GPU computation. This method further enhances the computational efficiency of both GPU and NPU. On Snapdragon 8 Elite, Hetero-tensor achieves about 10.5% higher prefill performance than on Snapdragon 8 Gen 3, benefiting from the enhanced capabilities of both the GPU and NPU.

Compared with other inference engines [42, 56] that only leverage NPU INT computation and may sacrifice model accuracy (above 20% accuracy degradation for Qualcomm-AI), HeteroInfer fully exploits the NPU's computational power (FLOAT). Furthermore, with the efficient GPU-NPU collaboration, HeteroInfer achieves comparable or even better performance compared to these works. For instance, Hetero-tensor achieves 1092 tokens/s during the prefill phase with a sequence length of 256 on InternLM-1.8B, while llm.npu [56] attains only 564 tokens/s under a similar model size.

5.3.2 Dynamic Sequence Length. Since current mobile NPUs only support static computation graphs, predefining a graph for every possible sequence length is infeasible. A straightforward solution, termed *Online-prepare*, generates a new NPU computation graph at runtime for each sequence length. However, this approach is inefficient, as online graph preparation may cause significant performance degradation.

Besides the straightforward solution, there are other alternative solutions such as *Padding*, *NPU-pipe* and *Chunked-prefill*. Padding mechanism preloads graphs of standard sizes (e.g., 128, 256, 512) and pads the misaligned inputs to the nearest size. NPU-pipe mechanism partitions a dynamic-size computation graph into multiple standard-size computation graphs (FFN layer) and executes these standard graphs sequentially on the NPU. llm.npu adopts Chunked-prefill mechanism by splitting the input sequence into fixed-size chunks. The chunk size must be chosen carefully to fully utilize the computational power of NPU and avoid unnecessary overhead. For instance, Chunked-prefill can only achieve the maximum prefill performance until the sequence length is 1024, and its performance is degraded to half when the sequence length is shortened to 256 [56].

Figure 14 shows prefill latency of different methods under dynamic and misaligned sequence lengths. Both graph preparation and computation times are reported for Online-prepare. Hetero-tensor consistently outperforms all baselines, achieving 2.24×, 2.21×, and 1.35× speedups over Online-prepare, Padding, and NPU-pipe, respectively, at a sequence length of 525. Online-prepare generally exhibits the highest latency due to graph preparation overhead, which grows with sequence length and the number of NPU graphs (typically 4). For instance, at length 135, preparation takes 408.4 ms (34.6% of total latency), rising to 2050 ms at length 1000. Padding causes stepwise increases in latency, leading to inefficiencies. When sequence length slightly exceeds a standard size, the padding mechanism will introduce average 1.91× overhead compared with Hetero-tensor. NPU-pipe mitigates padding overhead by partitioning the whole computation graph into smaller subgraphs with standard sizes. Compared to the NPU-pipe approach, Hetero-tensor further reduces the prefill latency by 13.2% to 30.1% by adaptively choosing the optimal partitioning strategy and enabling simultaneous execution of different computation subgraphs.

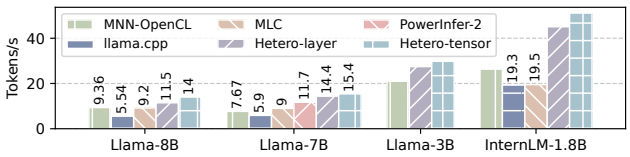

Figure 15. Decoding rate of MNN-OpenCL, llama.cpp, MLC, PowerInfer-2 and Hetero-tensor on different models. The prompt sequence length is 256.

5.4 Decoding Performance

Figure 15 presents the decoding speed of Hetero-tensor and other inference engines. Hetero-tensor reaches up to 14.01 tokens/s on Llama-8B, 29.9 tokens/s on Llama-3B and 51.12 token/s on InternLM-1.8B. On Llama-8B, Hetero-tensor achieves 1.50×, 2.53× and 1.52× speedup compared to MNN-OpenCL, llama.cpp, MLC, respectively. Even compared to PowerInfer-2 which utilizes the sparse model, Hetero-tensor still achieves 1.32× speedup, because sparse computations resulting in numerous random and small memory accesses, compromising the overall memory bandwidth.

Hetero-tensor is the only framework that utilizes both GPU and NPU in the decoding phase, and maximizes the SoC memory bandwidth. When NPU and GPU are running concurrently, the memory bandwidth increases from 43.3 GB/s (only GPU) to 59.5 GB/s, achieving 96% of the maximum available memory bandwidth. Given that the primary bottleneck during the decoding phase is memory bandwidth,

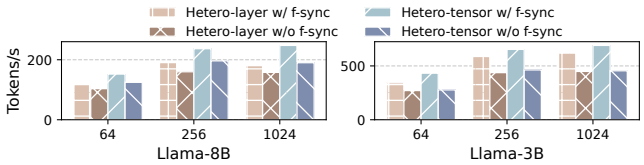

Figure 16. Prefill speed of Hetero-layer and Hetero-tensor with and without fast synchronization under sequence lengths of 64, 256 and 1024.

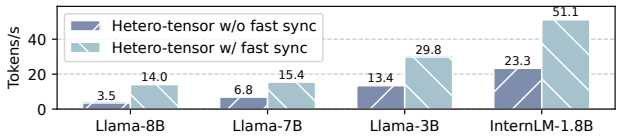

Figure 17. Decoding rate of Hetero-tensor running Llama-8B, Llama-7B, Llama-3B and InternLM-1.8B with and without fast synchronization. The prompt sequence length is 256.

we assert that Hetero-tensor is approaching the theoretical maximum decoding performance. Compared with Hetero-layer, Hetero-tensor further leverages weight-centric partitioning strategy for the Matmul operator, and achieves a 22.0% speedup in decoding stage for Llama-8B model. We also evaluate Hetero-tensor 's decoding performance on Snapdragon 8 Elite. Since the memory bandwidth of our selected Snapdragon 8 Elite device is the same as that of Snapdragon 8 Gen 3 device, the decoding performance remains similar.

5.5 Effect of Fast Synchronization

Prefill Performance. Figure 16 presents the prefill performance of Hetero-layer and Hetero-tensor with and without fast synchronization. On Llama-8B, fast synchronization yields average speedups of 15.8% for Hetero-layer and 24.3% for Hetero-tensor. Under a sequence length of 256, the prefill speed of Hetero-tensor increases from 196.44 tokens/s to 236.92 tokens/s. Due to GPU-NPU parallelism, Hetero-tensor is more sensitive to synchronization overhead, which can disrupt computational balance between the GPU and NPU.

Decoding Performance. Figure 17 presents the decoding performance of Hetero-tensor with and without fast synchronization. On Llama-8B, the decoding rate of Hetero-tensor speedups to 4.01× with fast synchronization. On other models, we observe comparable improvements of 2.2× speedup. The speedup in decoding phase is much higher than that in prefill phase, because the execution time of each kernel in the decoding phase is much shorter. Thus, the overhead of synchronization and GPU kernel submission is non-negligible.

5.6 Ablation Study

Since optimizations (NPU optimization, GPU-NPU parallelism, and fast synchronization) are only effective when applied jointly in the decoding stage, our ablation analysis focuses on the prefill stage. As illustrated in Figure 18, all of the techniques tailored to NPU and GPU characteristics effectively enhance prefill performance. When running the Llama-8B model, naïve NPU implementation is even 62.5%

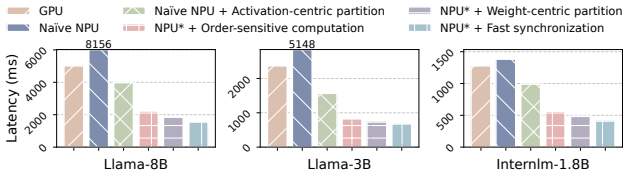

Figure 18. Ablation study of HeteroInfer when the prompt sequence length is 320. *NPU** stands for combination of all previous NPU techniques.

slower than GPU baseline due to the overhead of generating computation graphs on-the-fly. Applying activation-centric partition eliminates this overhead, resulting in a 2.05× speedup. Building on this, HeteroInfer further achieves a 1.79× improvement by rearranging the tensor ordering during computation to address the order-sensitive nature of NPU. When considering the shape-sensitive performance characteristics of NPUs, enabling weight-centric partition results in an additional 1.20× speedup. Finally, an efficient synchronization mechanism reduces GPU and NPU idle time during synchronization, leading to a further 1.19× speedup.

5.7 GPU Performance Interference

To evaluate the impact of HeteroInfer on system-level image rendering as well as interference with other GPU applications, we conduct experiments by running GPU-only, Hetero-layer and Hetero-tensor concurrently with a high-performance mobile game (League of Legends: Wild Rift), all graphical settings in the game are kept at default values.

As shown in Figure 19, when running concurrently with the game, the prefill speeds of Hetero-layer and Hetero-tensor decrease by only 0.5% and 2.2%, respectively, while the game's frame rate (FPS) remains unaffected. In contrast, running concurrently with the GPU-only baseline results in a severe FPS drop to zero—because the GPU kernels launched by the OpenCL runtime saturate the GPU submission queue, preventing the game's rendering tasks from completing within their required time budget. In comparison, Hetero-layer and Hetero-tensor offload only a small portion of computation to the GPU, preserving sufficient GPU resources for timely game rendering.

In the decoding phase, the FPS drop of the GPU-only baseline is alleviated, with the game maintaining 46 FPS, primarily because the GPU workload is lighter than in the prefill phase. Hetero-tensor continues to have no impact on FPS but

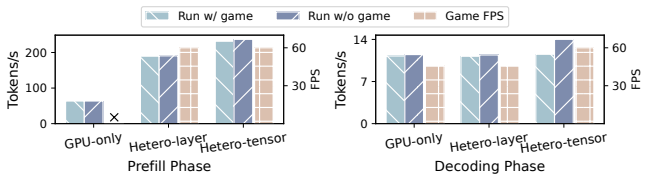

Figure 19. Prefill and decoding speed of GPU-only, Hetero-layer, and Hetero-tensor with and without concurrent game execution, along with the game's frame rate (FPS). Model: Llama-8B; prompt sequence length: 256.

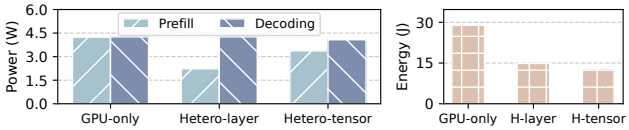

Figure 20. Power consumption during prefill and decoding, and end-to-end energy consumption for GPU-only, Hetero-layer, and Hetero-tensor (prompt sequence length: 256; output length: 32).

suffers a 17.7% reduction in decoding speed due to delayed GPU kernel execution, which prevents full overlap between NPU and GPU execution.

5.8 Energy Consumption

Figure 20 presents the power and energy consumption of GPU-only, Hetero-layer, and Hetero-tensor. During the prefill phase, Hetero-layer exhibits the lowest power consumption of 2.23W, primarily due to its reliance on the NPU for most computations. In the decoding phase, Hetero-tensor achieves the lowest power consumption by offloading computation to both the NPU and GPU. In terms of end-to-end energy consumption, Hetero-tensor is the most efficient, consuming 55% less energy than GPU-only and 12.8% less than Hetero-layer, owing to its faster execution.

6 DISCUSSION

Platform and Model Generality. Modern mobile SoCs commonly adopt similar hardware characteristics, such as unified memory architecture, an asynchronous GPU execution model, and a systolic-array-based NPU. Since the core observations and techniques of HeteroInfer are designed around these widely adopted features, the system can be ported to other mobile SoCs with minimal retuning. In terms of different models, even when the model architecture changes (e.g., MoE), the underlying operator types invoked during computation remain largely consistent. Our design focuses on operator-level optimization and enables GPU-NPU parallelism at the kernel granularity. As a result, the system is also applicable to models with diverse architectures and parameter configurations.

Model Quantization. HeteroInfer employs W4A16 (weight-only) quantization, striking a balance between memory footprint and computational accuracy. W4A16 quantization is the most widely used approach in real-world deployments. W4A16 quantizes model weights during storage and dequantizes them to FLOAT for computation. In contrast, other approaches [42, 56, 60] require quantization of both activations and weights to INT, which impairs inference accuracy.

Parallelism among Accelerators. While the parallelization of ML workloads is a well-established field, existing research[9, 63] has predominantly focused on distributing computations across homogeneous, discrete accelerators like NVIDIA GPUs and Google TPUs. Some prior approaches[22, 49], such as AccPar, have addressed heterogeneous systems. However, their scope is often confined to cloud-based scenarios, and the accelerators considered are typically from the same vendor lineage (e.g., TPUv2 and TPUv3), which feature similar hardware architectures and a unified software stack. In contrast, our work addresses the challenge of parallelization across fundamentally distinct accelerators integrated within a single mobile SoC.

Design of Future Edge AI Accelerators and Systems. Based on our experience in optimizing LLMs on commercial SoCs, we suggest that current edge AI SoCs can be further improved along following dimensions. First, unified GPU–NPU scheduling is needed. As future edge scenarios will involve increasingly complex multi-task workloads, including GPU-only, NPU-only, and GPU–NPU parallel tasks[47], the absence of a unified scheduling mechanism may lead to situations where tasks running on one processor simultaneously block the execution of tasks on both processors. Second, although current mobile SoCs adopt UMA, they often lack coherent memory management across different processing units. In our implementation, we establish shared memory between the CPU and GPU using OpenCL, between the CPU and NPU via QNN APIs, and further enable GPU-NPU shared memory. A unified API and memory management layer across CPU, GPU, and NPU would greatly simplify development and reduce unnecessary synchronization overhead. Third, a fast and lightweight synchronization library for heterogeneous processors would facilitate efficient overlap of computation and communication.

7 CONCLUSION

This paper introduces HeteroInfer, the fastest LLM inference engine optimized for heterogeneous processors in modern mobile SoCs. We conduct an in-depth analysis of the GPU and NPU hardware architectures, highlighting the NPU's tensor shape-sensitive performance characteristics. HeteroInfer leverages the GPU to improve the lower bound of NPU performance and enhance computational flexibility. It introduces several key techniques, including tensor partition, fast synchronization, and profiler-solver collaboration, to address the challenges of underutilized memory bandwidth and computing resources. HeteroInfer demonstrates a 1.34× to 6.02× speedup in real-world benchmarks, while maintaining negligible interference with other applications. Finally, this work offers new insights into the design of more efficient edge AI accelerators and heterogeneous SoCs.

Acknowledgments

We sincerely thank our shepherd and the anonymous reviewers of SOSP 2025, whose reviews, feedbacks, and suggestions have significantly strengthened our work. We also thank SenseTime for supporting this work. This work is supported in part by STI 2030—Major Projects 2021ZD0200300, China National Natural Science Foundation (No. 623B2074, 62088102, 62472279), Erhu Feng is the corresponding author.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA, July 2024. USENIX Association.

[3] Anthropic. Claude - right-sized for any task, the claude family of models offers the best combination of speed and performance. https://www.anthropic.com/, 2024. Referenced December 2024.

[4] Apple. A18. https://nanoreview.net/en/soc/apple-a18, 2024. Referenced December 2024.

[5] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding, 2024.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[7] Wenhua Cheng, Yiyang Cai, Kaokao Lv, and Haihao Shen. Teq: Trainable equivalent transformation for quantization of llms. *arXiv preprint arXiv:2310.10944*, 2023.

[8] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[9] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-Latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 135–153, Santa Clara, CA, July 2024. USENIX Association.

[10] Biji George, Om Ji Omer, Ziaul Choudhury, Anoop V, and Sreenivas Subramoney. A unified programmable edge matrix processor for deep neural networks and matrix algebra. *ACM Trans. Embed. Comput. Syst.*, 21(5), October 2022.

[11] Ggerganov. llama.cpp - LLM inference with minimal setup and state-of-the-art performance on a wide range of hardware. https://github.com/ggerganov/llama.cpp, 2023. Referenced December 2024.

[12] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, Hao Yu, Hongning Wang, Jiadai Sun, Jiajie Zhang, Jiale Cheng, Jiayi Gui, Jie Tang, Jing Zhang, Juanzi Li, Lei Zhao, Lindong Wu, Lucen Zhong, Mingdao Liu, Minlie Huang, Peng Zhang, Qinkai Zheng, Rui Lu, Shuaiqi Duan, Shudan Zhang, Shulin Cao, Shuxun Yang, Weng Lam Tam, Wenyi Zhao, Xiao Liu, Xiao Xia, Xiaohan Zhang, Xiaotao Gu, Xin Lv, Xinghan Liu, Xinyi Liu, Xinyue Yang, Xixuan Song, Xunkai Zhang, Yifan An, Yifan Xu, Yilin Niu, Yuantao Yang, Yueyan Li, Yushi Bai, Yuxiao Dong, Zehan Qi, Zhaoyu Wang, Zhen Yang, Zhengxiao Du, Zhenyu Hou, and Zihan Wang. Chatglm: A family of large language models from glm-130b to glm-4 all tools, 2024.

[13] Google. TensorFlow Lite - Google's high-performance runtime for on-device AI. https://tensorflow.google.cn/lite, 2017. Referenced December 2024.

[14] Wenyi Hong, Ming Ding, Wendi Zheng, Xinghan Liu, and Jie Tang. Cogvideo: Large-scale pretraining for text-to-video generation via transformers. *arXiv preprint arXiv:2205.15868*, 2022.

[15] Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14281–14290, 2024.

[16] Kuan-Chieh Hsu and Hung-Wei Tseng. Accelerating applications using edge tensor processing units. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Kuan-Chieh Hsu and Hung-Wei Tseng. Simultaneous and heterogenous multithreading. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 137–152, 2023.

[18] Kuan-Chieh Hsu and Hung-Wei Tseng. Shmt: Exploiting simultaneous and heterogeneous parallelism in accelerator-rich architectures. *IEEE Micro*, 2024.

[19] Huawei. Kirin-9000. https://www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-9000, 2020. Referenced December 2024.

[20] Huawei. Huawei hiai engine - about the service. https://developer.huawei.com/consumer/en/doc/hiai-References/ir-overview-0000001052569365, 2024. Referenced December 2024.

[21] Venkatraman Iyer, Sungho Lee, Semun Lee, Juitem Joonwoo Kim, Hyunjun Kim, and Youngjae Shin. Automated backend allocation for multi-model, on-device ai inference. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(3):1–33, 2023.

[22] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, et al. Whale: Efficient giant model training over heterogeneous {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 673–688, 2022.

[23] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479, 2020.

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. Heterogeneous dataflow accelerators for multi-dnn workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 71–83. IEEE, 2021.

[26] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[27] Jinhao Li, Jiaming Xu, Shan Huang, Yonghua Chen, Wen Li, Jun Liu, Yaoxiu Lian, Jiayi Pan, Li Ding, Hao Zhou, et al. Large language model inference acceleration: A comprehensive hardware perspective. *arXiv preprint arXiv:2410.04466*, 2024.

[28] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[29] LianjiaTech. Belle: Be everyone's large language model engine. https://github.com/LianjiaTech/BELLE, 2024.

[30] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024.

[31] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN model inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1025–1040, Renton, WA, July 2019. USENIX Association.

[32] Timothy R McIntosh, Teo Susnjak, Tong Liu, Paul Watters, and Malka N Halgamuge. From google gemini to openai q*(q-star): A survey of reshaping the generative artificial intelligence (ai) research landscape. *arXiv preprint arXiv:2312.10868*, 2023.

[33] Microsoft. Accelerated edge machine learning. https://onnxruntime.ai/, 2020. Referenced April 2023.

[34] MLC team. MLC-LLM - Universal LLM Deployment Engine with ML Compilation. https://github.com/mlc-ai/mlc-llm, 2023-2024. Referenced December 2024.

[35] Nanoreview. Online. https://nanoreview.net/en/soc-list/rating, 2023-2024. Referenced December 2024.

[36] Mohanad Odema, Luke Chen, Hyoukjun Kwon, and Mohammad Abdullah Al Faruque. Scar: Scheduling multi-model ai workloads on heterogeneous multi-chiplet module accelerators. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 565–579, 2024.

[37] OpenAI. Gsm8k (grade school math 8k). https://huggingface.co/datasets/openai/gsm8k, 2024.

[38] OpenAI. Introducing chatgpt. https://openai.com/index/chatgpt/, 2024. Referenced January 2024.

[39] Gunho Park, Baeseong Park, Minsub Kim, Sungjae Lee, Jeonghoon Kim, Beomseok Kwon, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. Lut-gemm: Quantized matrix multiplication based on luts for efficient inference in large-scale generative language models. *arXiv preprint arXiv:2206.09557*, 2022.

[40] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132, 2024.

[41] Qualcomm. Snapdragon 8Gen3 - Mobile Platform ignites endless possibilities. https://www.qualcomm.com/products/mobile/snapdragon/smartphones, 2023-2024. Referenced December 2024.

[42] Qualcomm. Llama-v3.1-8b-chat on qualcomm 8 elite. https://aihub.qualcomm.com/mobile/models/llama_v3_1_8b_chat_quantized?searchTerm=llama, 2024. Referenced December 2024.

[43] Qualcomm. Qnn. https://www.qualcomm.com/developer/software/qualcomm-ai-engine-direct-sdk, 2024. Referenced December 2024.

[44] Qualcomm. Qualcomm adreno gpu, game-changing speed and efficiency. https://www.qualcomm.com/products/technology/processors/adreno, 2025.

[45] Qualcomm. Qualcomm hexagon npu, powering the generative ai revolution. https://www.qualcomm.com/products/technology/processors/hexagon, 2025.

[46] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[47] Weihang Shen, Mingcong Han, Jialong Liu, Rong Chen, and Haibo Chen. {XSched}: Preemptive scheduling for diverse {XPUs}. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 671–692, 2025.

[48] Sudipta Saha Shubha and Haiying Shen. Adainf: Data drift adaptive scheduling for accurate and slo-guaranteed multiple-model inference serving at edge servers. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 473–485, 2023.

[49] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Accpar: Tensor partitioning for heterogeneous deep learning accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 342–355. IEEE, 2020.

[50] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.

[51] Tencent. NCNN - High-performance neural network inference computing framework optimized for mobile platforms. https://github.com/Tencent/ncnn, 2017. Referenced December 2024.

[52] Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *arXiv preprint arXiv:2406.01014*, 2024.

[53] Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158*, 2024.

[54] Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Yang Fan, Kai Dang, Mengfei Du, Xuancheng Ren, Rui Men, Dayiheng Liu, Chang Zhou, Jingren Zhou, and Junyang Lin. Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution. *arXiv preprint arXiv:2409.12191*, 2024.

[55] Zhaode Wang, Jingbang Yang, Xinyu Qian, Shiwen Xing, Xiaotang Jiang, Chengfei Lv, and Shengyu Zhang. Mnn-llm: A generic inference engine for fast large language model deployment on mobile devices. In *Proceedings of the 6th ACM International Conference on Multimedia in Asia Workshops*, MMAsia '24 Workshops, New York, NY, USA, 2024. Association for Computing Machinery.

[56] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. Fast on-device llm inference with npus, 2024.

[57] Yuqi Xue, Yiqi Liu, and Jian Huang. System virtualization for neural processing units. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 80–86, New York, NY, USA, 2023. Association for Computing Machinery.

[58] Yuqi Xue, Yiqi Liu, Lifeng Nai, and Jian Huang. V10: Hardware-assisted npu multi-tenancy for improved resource utilization and fairness. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[59] Yuqi Xue, Yiqi Liu, Lifeng Nai, and Jian Huang. Hardware-assisted virtualization of neural processing units for cloud platforms. *arXiv*

preprint arXiv:2408.04104, 2024.

[60] Zhenliang Xue, Yixin Song, Zeyu Mi, Xinrui Zheng, Yubin Xia, and Haibo Chen. Powerinfer-2: Fast large language model inference on a smartphone, 2024.

[61] Zhuoyi Yang, Jiayan Teng, Wendi Zheng, Ming Ding, Shiyu Huang, Jiazheng Xu, Yuanming Yang, Wenyi Hong, Xiaohan Zhang, Guanyu Feng, et al. Cogvideox: Text-to-video diffusion models with an expert transformer. *arXiv preprint arXiv:2408.06072*, 2024.

[62] Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. Ferret-ui: Grounded mobile ui understanding with multimodal llms. In *European Conference on Computer Vision*, pages 240–255. Springer, 2025.

[63] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[64] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.

[65] Pan Zhang, Xiaoyi Dong, Yuhang Zang, Yuhang Cao, Rui Qian, Lin Chen, Qipeng Guo, Haodong Duan, Bin Wang, Linke Ouyang, et al. Internlm-xcomposer-2.5: A versatile large vision language model supporting long-contextual input and output. *arXiv preprint arXiv:2407.03320*, 2024.

[66] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association, November 2020.

[67] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, Santa Clara, CA, July 2024. USENIX Association.

[68] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, Carlsbad, CA, July 2022. USENIX Association.

[69] Donglin Zhuang, Zhen Zheng, Haojun Xia, Xiafei Qiu, Junjie Bai, Wei Lin, and Shuaiwen Leon Song. MonoNN: Enabling a new monolithic optimization space for neural network inference tasks on modern GPU-Centric architectures. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 989–1005, Santa Clara, CA, July 2024. USENIX Association.