Article

Not peer-reviewed version

HeteroLLM: Accelerating Large Language Model Inference on Mobile SoCs with Heterogeneous AI Accelerators

Le Chen , Dahu Feng , Erhu Feng , Yingrui Wang , Rong Zhao , Yubin Xia * , Haibo Chen , Pinjie Xu

Posted Date: 13 January 2025

doi: 10.20944/preprints202501.0901.v1

Keywords: Machine learning; Inference engine; Heterogeneous system

Article

HeteroLLM: Accelerating Large Language Model Inference on Mobile SoCs with Heterogeneous AI Accelerators

Le Chen [1,†], Dahu Feng [2,†], Erhu Feng [1], Rong Zhao [2], Yingrui Wang [3], Yubin Xia [1,*], Haibo Chen [1] and Pinjie Xu [3,‡]

1 Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
2 Tsinghua University
3 SenseTime Research
* Correspondence: xiayubin@sjtu.edu.cn
† The two authors contributed equally to this work and should be considered co-first authors.
‡ Project Leader.

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 mobile SoC, including heterogeneous processors, unified memory, synchronization, etc. Drawing on these observations, we propose different tensor partition strategies to fulfill the distinct requirements of the prefill and decoding phases. We further design a fast synchronization mechanism that leverages the unified memory address provided by mobile SoCs. By employing these techniques, we present HeteroLLM, the fastest LLM inference engine in mobile devices which supports both layer-level and tensor-level heterogeneous execution. Evaluation results show that HeteroLLM achieves 9.99× and 4.36× performance improvement over other mobile-side LLM inference engines: MLC and MNN.

1. Introduction

Driven by the rapid evolution in large language models (LLMs), technologies such as ChatGPT [1–4], AI agents [5–8], and video generation [9,9–13] 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 [14–21] enhance capabilities for vector and matrix computations, aligning with the computational demands of AI applications. For example, Qualcomm's smartphones incorporate Adreno GPUs and Hexagon NPUs to address the computing needs of edge AI applications. Furthermore, by integrating different computational units within a single SoC, these processors can utilize a unified physical memory, thus obviating the explicit data copying.

To fully leverage the computing resources in heterogeneous systems, prior researches [22–24] have proposed inference engines designed for heterogeneous processors. However, these solutions are not compatible with current mobile platforms. First, traditional synchronization methods for GPUs and NPUs can incur significant overhead during LLM inference, particularly during the decoding phase, where each kernel executes only hundreds of microseconds. Second, NPUs exhibit significantly higher performance than GPUs. For instance, while the GPU in the Qualcomm 8 Gen 3 [25] platform delivers approximately 1 TFLOPS (in actual, theoretical 2.8 TFLOPS) of computing power, its NPU can reach up to 10 TFLOPS (in actual) performance. Enforcing parallel execution of the GPU and NPU, as suggested by previous approaches, may actually degrade end-to-end performance. We also notice that some previous studies [26–29] utilize model sparsity with mixed-precision techniques. These approaches assign high-precision but limited operations to the CPU, while offloading low-precision, high-volume operations to the NPU. The performance and accuracy of these methods heavily depend on sparsity in activations and weights, whereas recent research [30,31] points out that LLMs are increasingly exhibiting dense characteristics. Therefore, designing an efficient inference engine for all heterogeneous processors (i.e., CPU, GPU and NPU) in real-world mobile devices remains a significant challenge.

Through an in-depth analysis of the heterogeneous processors within mobile SoCs, we have identified several distinctive characteristics based on their hardware architecture.

- **Tensor-sensitive NPU performance:** Although NPUs can exhibit superior performance under optimal conditions, their efficiency is highly dependent on tensor factors such as the order, size and shape. If tensors do not align with the NPU's hardware structure, the performance benefit of 'weight-stall' computation cannot be fully realized, potentially causing performance to regress to GPU levels.
- **Static NPU graph with a high generation cost:** Existing mobile NPUs only support static computation graphs, which are incompatible with dynamic workloads of LLMs. Due to inherent constraints in the NPU architecture, generating an optimal graph for NPU is more complex than for GPU. Moreover, the overhead of graph generation is non-negligible and related to the size of each tensor, making it impractical to generate the NPU graph during runtime.
- **Memory bandwidth restriction for a single processor:** A single processing unit is insufficient to fully saturate the SoC's memory bandwidth. For example, GPU alone can utilize only $40 \sim 45$ 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 (maximum memory bandwidth in SoC is 68 GB/s).

These distinctive performance traits unlock new opportunities for enhancing GPU and NPU parallelism: *By strategically leveraging the GPU, it can compensate for the NPU performance limitation in specific scenarios.*

In this paper, we introduce HeteroLLM, 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 LLM tasks due to 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, HeteroLLM 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, HeteroLLM further introduces three techniques. First, HeteroLLM applies different tensor partition strategies during both the prefill and decoding phases to facilitate tensor-level heterogeneous execution. Second, HeteroLLM implements a fast synchronization mechanism based on predictable kernel waiting times to achieve microsecond-level synchronization. Third, HeteroLLM incorporates a tensor partition solver that generates optimal partition solutions using a hardware profiler and a runtime decider.

We have implemented a prototype of HeteroLLM on the Qualcomm 8 Gen 3 SoC, one of the most advanced mobile platforms, integrating Arm CPU, GPU and NPU. Our system builds upon the baseline of PPL [32], a state-of-the-art LLM inference engine supporting both CPU and GPU. To incorporate the NPU, we integrated NPU operators provided by Qualcomm's QNN [33] into our framework. As for the inference accuracy, we avoid using the activation quantization and sparsity techniques, as these techniques are orthogonal to our approach. HeteroLLM is the first LLM engine to surpass 1000 tokens per second in prefill phase using **FLOAT** calculations on mobile devices for billion-scale LLMs. In the prefill phase, compared to other SOTA inference engines, just using the layer-level heterogeneous execution can improve the speed up to 7.27× over MLC [34] and 3.18× over MNN [35]. Furthermore, tensor-level heterogeneous execution delivers nearly a 40% performance improvement compared to layer-level execution. When the sequence length is misaligned with the shape of NPU graph, tensor-level approach achieves up to 2.12× improvement than padding approach. In the decoding phase, HeteroLLM is capable of generating 23.4% more tokens than GPU-only. When running concurrently with GPU-intensive workloads, HeteroLLM minimizes the interference between LLM inference and the rendering tasks (without FPS drop), with only a 7.26% slowdown for LLM tasks.

2. Background & Related Work

Table 1. Specifications [36] 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	73 Tops	36 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.

2.1. LLM Inference

Large Language Model (LLM) inference refers to the process of using a pre-trained model to generate predictions or outputs based on new input data. Generally, it consists of two distinct phases: the prefill phase and the decoding phase. During the prefill phase, the LLM processes the user's input in a single batch, generating the first token. Due to the potentially large sequence length of the user's input, this phase relies on matrix multiplication operations, rendering it computationally intensive. Conversely, the decoding phase is distinguished by the sequential and auto-regressive manner, with each subsequent token being produced one at a time. Due to the KV Cache [37–39] support, this phase requires matrix-vector multiplication, resulting in a memory-intensive workload.

In contrast to cloud-side LLM inference (e.g. vLLMs [40], orca [41], etc. [42–49]), 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, which denotes the latency until the generation of the first token, is primarily influenced by the speed of prefill phase processing. The latter, which indicates the time required to produce each subsequent token, is associated with the token generation speed during the decoding phase.

2.2. Mobile-Side Heterogeneous SoC

Table 2. 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. **Sparse activation** refers to sparsity reliance on quantized activations. **Accuracy** indicates whether the model's accuracy is consistent with the original model.

Framework	**CPU**		**GPU**		**NPU**		**NPU GEMM Type**	**Independency on Sparse Activation**	**Accuracy**	**Performance**
	INT	**FP**	**INT**	**FP**	**INT**	**FP**				
MLLM-NPU [50]	INT4	FP16/32	/	/	INT8	/	INT	✗	Depend on activation	High
Qualcomm-AI [51]	INT4/8	W4A16	/	FP16	INT4/8	/	INT	✓	Decrease	High
MLC [34]	/	W4A16	/	W4A16	/	/	/	✓	✓	Low
Llama.cpp [52]	INT4/8	W4A16	/	W4A16	/	/	/	✓	✓	Low
Onnxruntime [53]	/	FP16/32	/	/	INT8/16	/	INT	✓	Decrease	Medium
MNN [35]	INT8	W4A16	/	W4A16	/	/	/	✓	✓	Medium
Ours	INT8	W4A16	INT8	W4A16	INT4/8	W4A16	FLOAT	✓	✓	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.

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 [25], Apple's A18 [54], MediaTek's Dimensity 9300 [54], Huawei's Kirin 9000 [55], etc. Table 1 lists the parameter specifications of several mainstream mobile SoC platforms. To support the massive computational power required by LLMs, mobile platforms are evolving towards to the heterogeneous SoC. In addition to the conventional CPUs and GPUs, NPUs are increasingly playing a critical role in these platforms. Generally, these heterogeneous processing units can share a unified physical memory, which is significantly different from discrete heterogeneous systems.

2.3. Mobile-Side Inference Engine

With the growing demand for LLMs, prior works have developed mobile-side inference engines, such as ONNX-Runtime [53], Llama.cpp [52] MNN [35], NCNN [56], OpenVino [57], TFLite [58], MLC [34], PPL [32] and etc [59,60]. Considering the complexity, diversity and incompatibility of mobile-side devices, it is challenging to establish a unified and comprehensive software ecosystem. To address these challenges, these mobile-side inference engines generally utilize the ONNX format [53] as input, and then perform a series of optimizations, such as graph optimization and operator fusion to construct the model's runtime graph. To support various devices, inference engines typically abstract mobile accelerators into different backends like CPU, GPU and NPU. They further utilize instruction sets and programming languages (e.g., CPU: NEON, SVE2 and AVX; GPU: OpenCL, Vulkan and CUDA; NPU: QNN, HIAI and CoreML) to implement the corresponding low-level operators in the runtime graph. Figure 1 illustrates the general framework of the inference engine: PPL, which has been chosen as the baseline for this work.

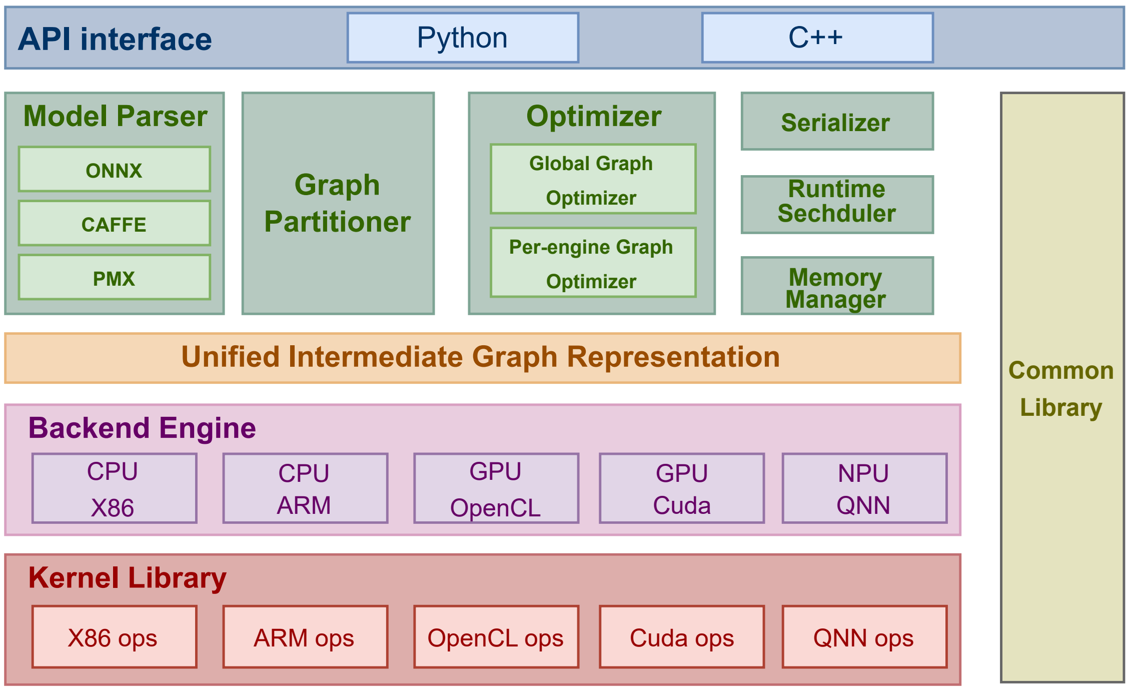

Figure 1. The overall framework of the inference engine referenced by HeteroLLM.

As mentioned in Section 2.2, modern mobile SoCs typically feature multiple heterogeneous processors, meaning that multiple processing units can be simultaneously utilized for AI tasks execution on a single mobile device. Some works have noted this phenomenon, as shown in Table 2. Qualcomm-AI [51] only utilizes NPU's INT calculation for matrix multiplication, which significantly sacrifices the inference accuracy [30,31]. MLLM-NPU [50] is designed for efficient on-device LLM inference using NPU offloading. It optimizes prompt processing, tensor outliers handling and out-of-order block scheduling to significantly enhance performance and energy efficiency. However, it only utilizes INT operations provided by the NPU and heavily depends on model activation sparsity and quantization. Both of these mechanisms can potentially reduce model accuracy. MLC [34] and MNN [35] utilize the CPU and GPU for LLM inference, Onnxruntime [53] leverages the CPU and NPU for computation, but none of these works incorporate heterogeneous parallelism at the tensor granularity. Moreover, all prior works fail to address GPU-NPU parallelism, which represents a more powerful combination for mobile devices.

3. Performance Characteristic

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.

3.1. GPU Characteristics

Mobile GPUs share a similar computing architecture with discrete GPUs for the desktop, including SIMT instructions, on-chip shared memory and streaming multiprocessors or compute units (SM/CU). The difference is that mobile GPUs employ a distinct memory hierarchy characterized by a unified memory address space (UMA) integrated into the system memory. Operations like data transfers between CPU-side and GPU-side memory, which are necessary in discrete GPUs, become redundant in mobile GPUs. However, traditional GPU frameworks such as OpenCL are not designed for these UMA-GPUs, and adhere to the abstraction of a dedicated GPU memory for mobile GPUs.

Characteristic ❶: Linear Performance. Figure 2 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, the total FLOPS stays stable.

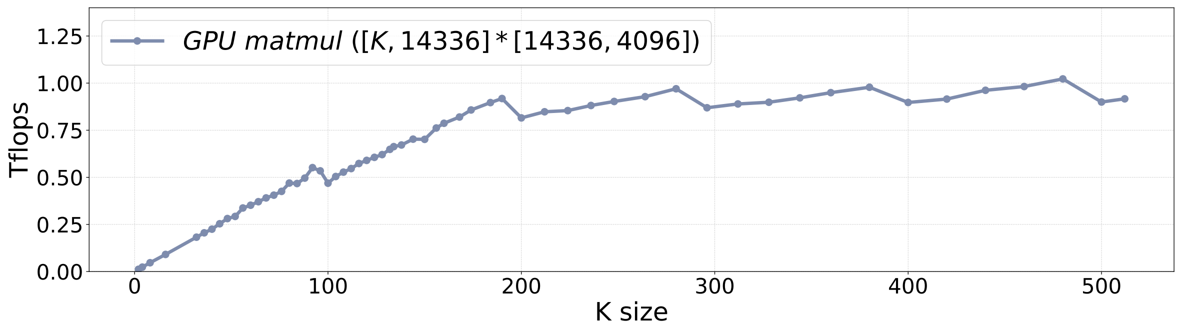

Figure 2. The GPU performance with varying tensor sizes.

Characteristic ❷: 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. Unfortunately, this transfer process incurs a fixed latency, approximately 400 microseconds on our platform, regardless of data size. The second type of overhead is related to kernel submission. As GPUs adopt the asynchronous programming model, subsequent kernels can be queued while the current one is executing, making the submission overhead negligible (about 10 to 20 microseconds).

However, after synchronization, the GPU queue becomes empty, which causes an additional latency of 50 to 100 microseconds due to the overhead of kernel queueing and submission.

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 3 demonstrates a classical 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.

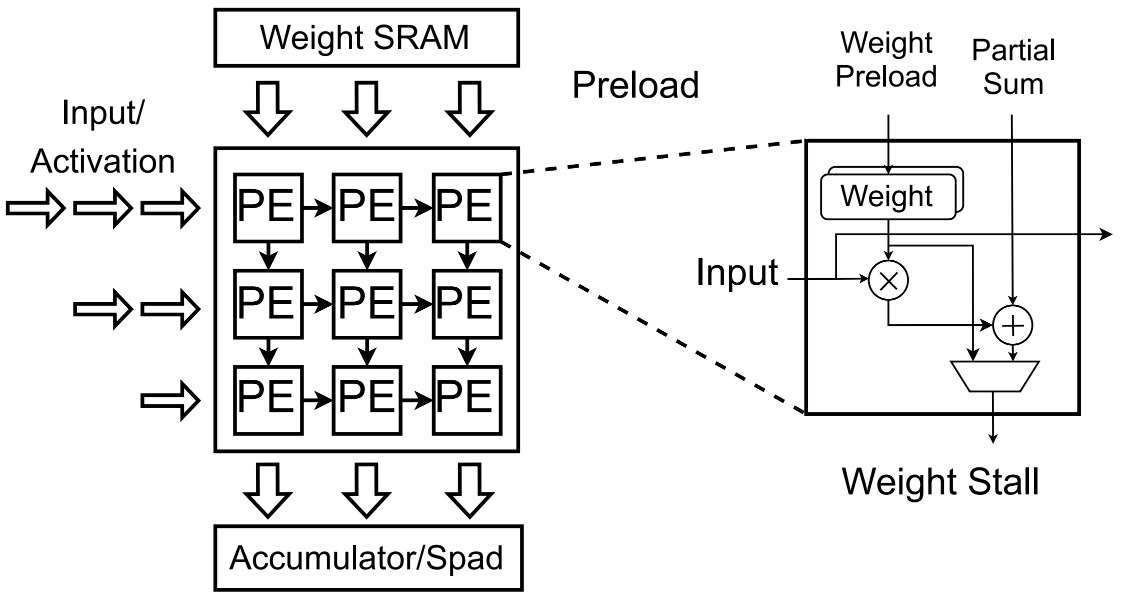

Figure 3. The hardware design for NPU: Systolic array with the weight stall computing paradigm.

Characteristic ❶: Stage Performance. Due to the fixed size of the hardware computing array (e.g., systolic array) within NPUs, the dimensions of the tensor used for the Matmul operator may not align with the size of the hardware computing units, which can lead to inefficient use of computational resources. As shown in Figure 4, this misalignment results in a phenomenon referred to as *stage performance* across different tensor sizes. For instance, considering an NPU with a matrix computation unit utilizing 32×32 systolic arrays, any computing tensor with dimensions smaller than 32 will exhibit the same computational latency, leading to significant performance degradation for certain tensor shapes. To fully utilize the NPU's computational resources, the compiler partitions tensors into tiles that align with the hardware configuration of the matrix computation unit. When tensor dimensions are not divisible by the size of the matrix computation unit, the NPU compiler must introduce internal padding to align with the required computation size. This alignment results in a stage performance effect during NPU calculations.

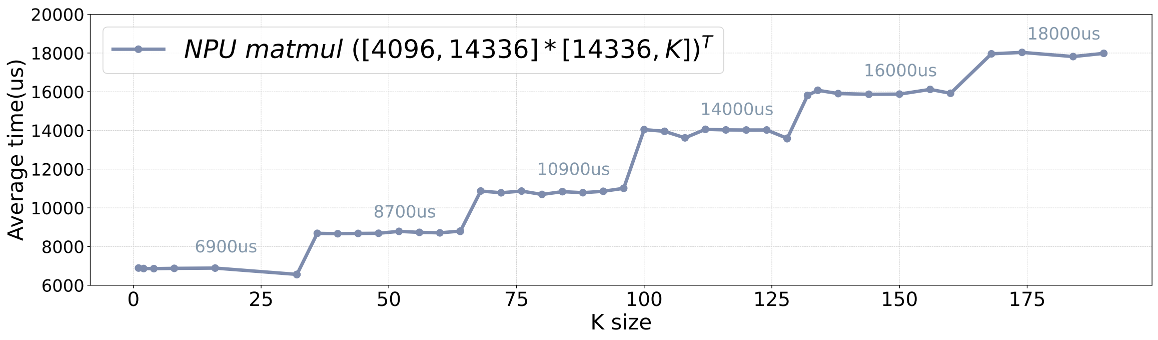

Figure 4. The stage performance of NPUs. The execution time of the Matmul operator across different tensor sizes.

Characteristic ❷: Order-sensitive Performance. In addition to stage performance, NPUs also exhibit order-sensitive computation behavior. Consider two tensors with 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 5 presents a specific example where the matrix multiplication operation of $[14336, 4096] \times [4096, K]$ achieves $6\times$ performance improvement compared to $[K, 4096] \times [4096, 14336]$.

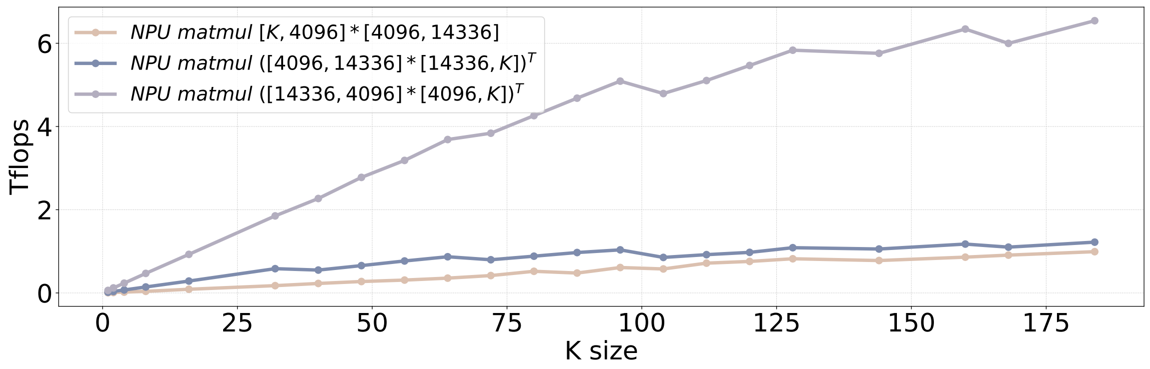

Figure 5. The order-sensitive and shape-sensitive performance of NPUs. The performance of the NPU is significantly influenced by the order and shape of the tensors.

The primary reason for order-sensitive performance is that NPU leverages the weight stall computing to minimize memory load/store overhead. Ideally, the weight tensor 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, even though $[K, N] \times [N, M]$ and $[M, N] \times [N, K]$ involve the same number of computational operations, the larger size of $[N, M]$ compared to $[N, K]$ results in inferior performance due to the extra memory operations involved. In the worst-case scenario for NPU computation, if the input tensor is $[M, N]$ and the weight tensor is hypothetically [N, infinite], the matrix computation unit cannot take advantage of the weight-stall computing paradigm, and thus, the NPU performance regresses to the GPU level.

Characteristic ❸: Shape-sensitive Performance. In addition to 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 5). This is primarily because the column size of the input tensor is the same as the row size of the weight tensor. A larger column size of input tensor results in a larger weight tensor, undermining the advantages of the weight-stall computation paradigm.

3.3. SoC Memory Bandwidth

Characteristic ❶: Underutilized Memory Bandwidth with Single Processor. Although mobile SoCs employ a unified memory address space for multiple heterogeneous processors, we have observed that no single processor can fully utilize the total memory bandwidth of the SoC under decoding workloads. As shown in Figure 6, in the Qualcomm Snapdragon 8 Gen 3 platform, the maximum available SoC memory bandwidth is approximately 68 GB/s (the black dotted line in the figure). However, using one processor (e.g., CPU, GPU or NPU), can only achieve 40 to 45 GB/s under decoding workloads. When NPU and GPU tasks are executed concurrently, the combined memory bandwidth utilization increases to approximately 60 GB/s, which is very close to the theoretical bandwidth limit. Therefore, NPU-GPU parallelism presents a new opportunity to enhance the decoding phase of LLMs, given that the token generation rate is linearly correlated with the available memory bandwidth.

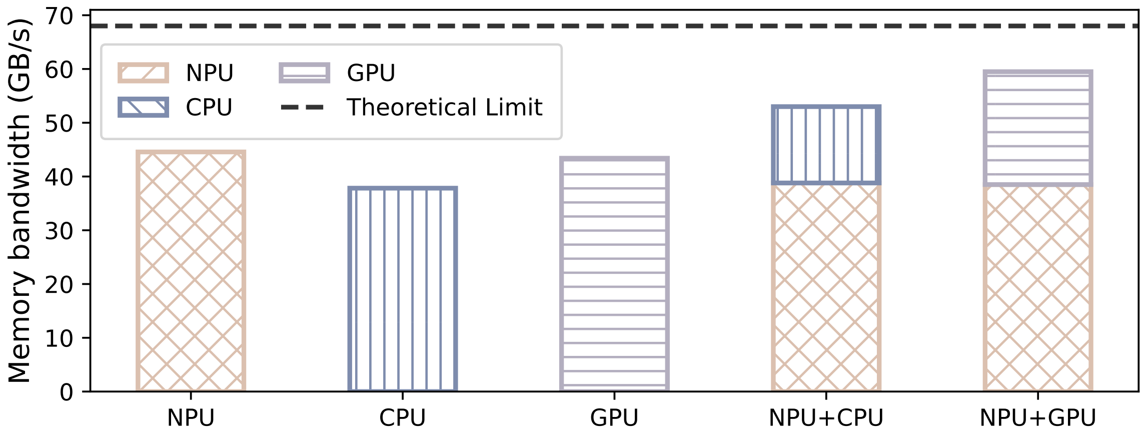

Figure 6. The total memory bandwidth with single and multiple processors. We execute the decoding workloads across different backends and measure the available memory bandwidth in the entire SoC.

4. Design

Given the power constraints and the presence of other applications in mobile systems, we avoid exhausting all available power of heterogeneous processors for LLM tasks. For instance, CPUs are ill-suited as dedicated backends for LLM tasks due to their low energy efficiency and engagement in general-purpose tasks. Consequently, HeteroLLM only utilizes the CPU as a control plane for synchronization, GPU kernel scheduling, and handling non-compute-intensive tasks such as dequantization. As for the others, the NPU outperforms the GPU in most cases but can have significant performance degradation when doing certain calculations. Therefore, our system designates the NPU as the primary computing unit, while leveraging the GPU to enhance the lower bound performance of the NPU in specific cases.

Figure 7 shows the overall execution flow of a typical LLM with HeteroLLM. HeteroLLM employs two methods for GPU-NPU parallelism: **layer-level** heterogeneous execution and **tensor-level** heterogeneous execution. The **layer-level** approach incorporates two key optimizations. First, different operators are assigned to the most suitable backends: for instance, Matmul operators are directed to the NPU backend, whereas RMSNorm/SwiGLU operators are more efficiently handled by the GPU backend. Second, since typical LLM models feature larger size of weight tensors compared to user's input tensors, the input and weight tensors are permuted from $[M, N] \times [N, K] \rightarrow [[K, N] \times [N, M]]^T$, to meet *NPU-❷: order-sensitive performance*. For the **tensor-level** approach, HeteroLLM introduces various tensor partition strategies for different backends (Section 4.1), and designs a solver to determine the optimal partition solution (Section 4.3). Both approaches adopt a novel synchronization technique to reduce synchronization overhead between the NPU and GPU (introduced in Section 4.2).

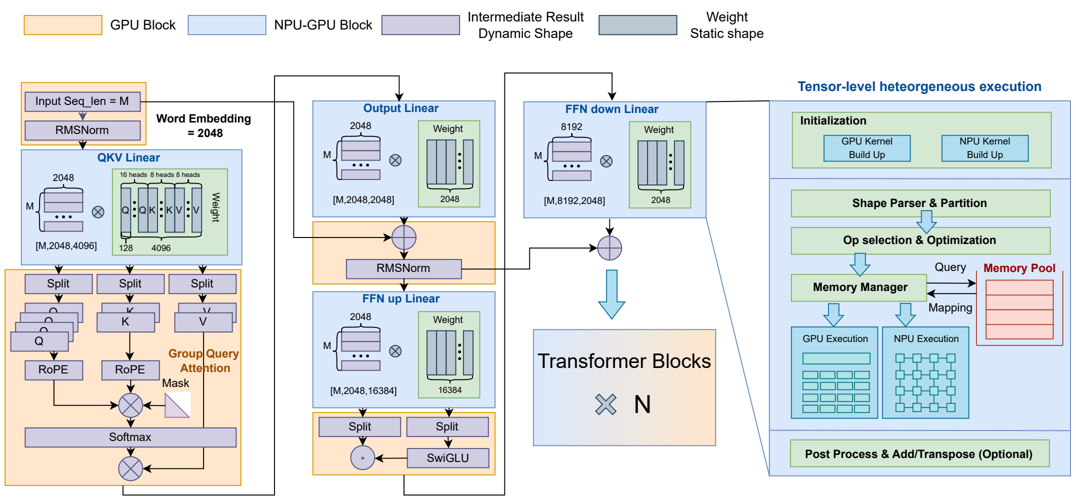

Figure 7. The overall execution flow of a typical LLM with HeteroLLM: Operators within the orange blocks are executed on the GPU backend, whereas operators within the blue blocks can be offloaded to heterogeneous backends. HeteroLLM additionally supports tensor-level heterogeneous execution though different tensor partition strategies.

4.1. Tensor Partition Strategies

HeteroLLM introduces the tensor-level heterogeneous execution with three distinct partition strategies: row-cutting, sequence-length cutting and hybrid-cutting. These strategies address three key deficiencies associated with NPU-only execution: (1) performance degradation for specific tensor shapes, (2) static computational graphs with higher graph generation costs, and (3) underutilization of the SoC memory bandwidth as well as computational power of heterogeneous processors.

4.1.1. Tensor Partition During the Prefill Phase

Row-cutting. In the prefill phase, although the NPU can outperform the GPU by an order of magnitude in ideal scenarios, its performance is significantly influenced by the shape of the input and weight tensors. First, when the sequence length is short, NPU cannot exploit all available computational resources due to *NPU-❶: stage performance*, resulting in a similar performance compared to the GPU. Second, due to the dimensionality reduction matrix inherent in the FFN-down, the column size of this matrix is larger than the row size (after transposition). This configuration is suboptimal for NPU execution even with large sequence lengths, owing to *NPU-❸: shape-sensitive performance*. In such scenarios, the NPU exhibits only 0.5× to 1.5× performance improvement over the GPU, due to the NPU's exceptionally low computational efficiency on this tensor shape. The Matmul on this tensor accounts for nearly half of the total prefill execution time.

In these two cases, we propose a *row-cutting* strategy for GPU and NPU parallelism. As illustrated in Figure 8, row-cutting partitions the first tensor into sub-tensors based on the row dimension, dispatching part of the computational workload from the NPU to the GPU. To ensure that all activations (i.e., inputs) from the last layer are available before the next layer begins execution, HeteroLLM explicitly incorporates synchronization points into the computational graph and manages the splitting and merging of intermediate results across different backends. For an ideal partition, the GPU and NPU will complete their computations simultaneously, thereby reducing end-to-end latency.

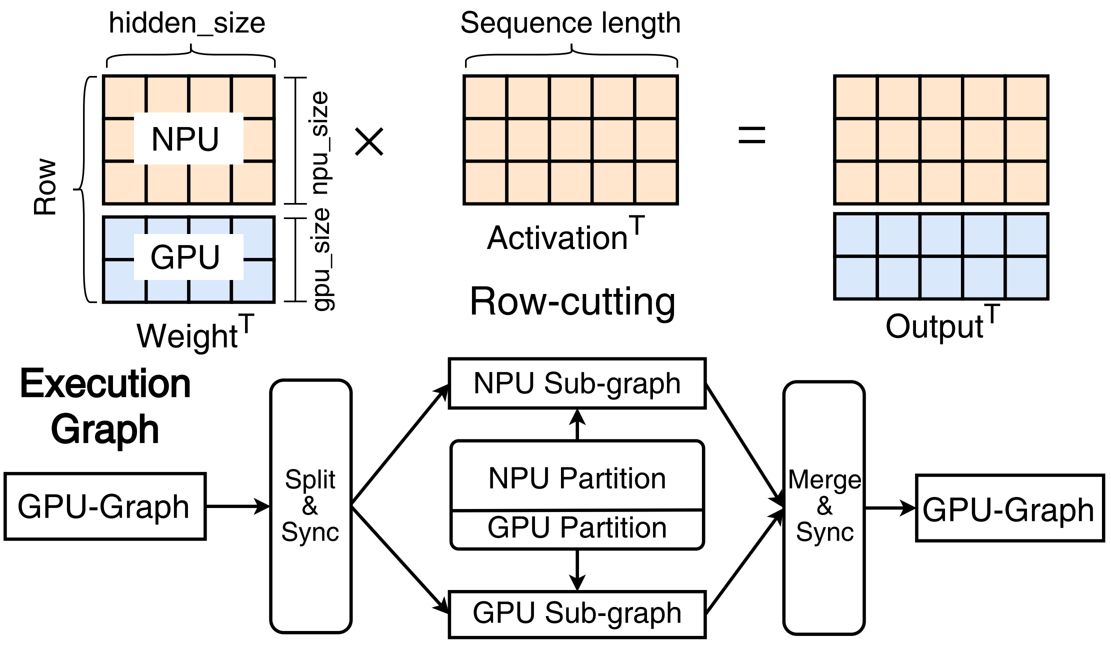

Figure 8. HeteroLLM partitions the weight tensor based on the row dimension, and dispatches computational loads to the different backends.

Sequence-length cutting. In addition to NPU's fluctuating performance, the mobile-side NPUs present another constraint: they only support static graph execution. The shape and size of tensors at runtime need to be ascertained during the kernel initialization phase. This limitation stems from the dataflow graph compilation [61–63], a method widely adopted by current mobile NPUs [33,64]. Furthermore, as shown in Figure 9, the cost of kernel optimization for the NPU is highly dependent on tensor size, as larger tensors expand the search space for optimization [65–67]. In contrast, the GPU framework provides a set of kernel implementations, each of which is adaptable to a variety of tensor shapes. This facilitates the dynamic-shape kernel execution at runtime.

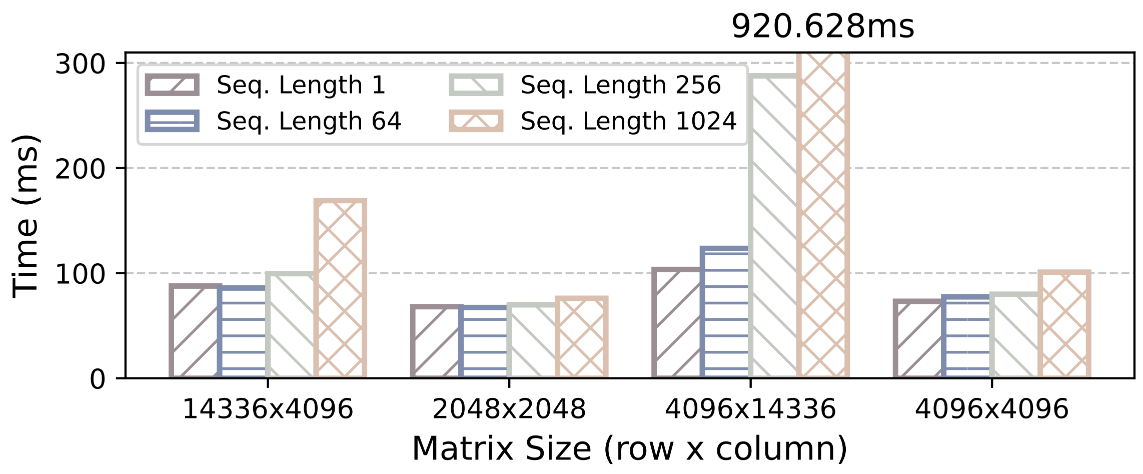

Figure 9. The NPU graph generation time for a single operator across various tensor shapes.

In order to support dynamic input shapes for the NPU engine, a standard approach is to choose a set of predefined tensor shapes, such as 128, 256 and 512, and then pad the input data to conform to these shapes. For instance, if the sequence length of input tokens is 300, the inference engine pads it to an aligned size of 512 to avoid the overhead associated with NPU graph generation for a new tensor size. However, this padding introduces additional computational overhead. To address this, we propose a *sequence-length cutting* strategy to support dynamic tensor shapes while minimizing redundant padding overhead. As shown in Figure 10, HeteroLLM offloads computation tasks involving dynamic-shape tensors to the GPU (according to *GPU-➊: linear performance*), while retaining fixed-size tensor computations on the NPU. For example, with an input sequence length of 300, we partition the tensor into two segments: 44 and 256. Since 256 is a standard tensor shape, its computation graph is pre-generated, allowing it to be processed by the NPU. The segment with the size of 44 (not a pre-defined shape), is executed by the GPU backend concurrently with the NPU. Ideally, NPU execution overlaps with GPU execution, thus eliminating additional overhead.

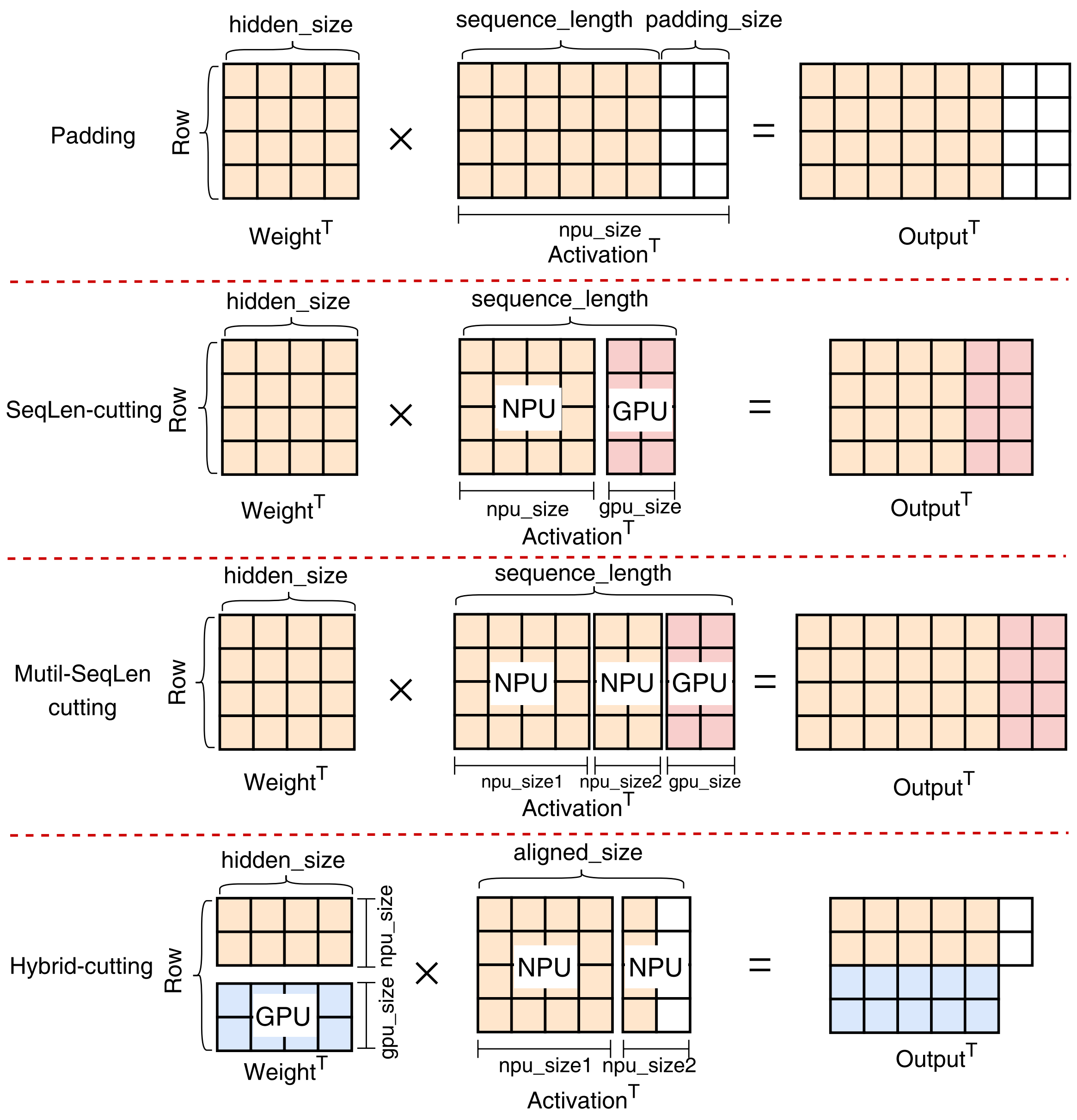

Figure 10. Due to the static computation graph of the NPU, HeteroLLM employs sequence-length cutting to support the dynamic sequence lengths required by LLMs.

Multi-sequence-length cutting and Hybrid-cutting. As the GPU's performance is generally weaker than the NPU, its computations will become a bottleneck when the sequence length surpasses a certain threshold. To mitigate this, we further partition the input tensor along the sequence length dimension into multiple sub-tensors, each with predefined shapes, and one additional sub-tensor with an arbitrary shape (*Multi-sequence-length cutting*). All sub-tensors with predefined shapes are executed sequentially on the NPU. For instance, if the input tensor's sequence length is 600, it can be divided into sub-tensors with sizes of 512, 32 and 56. 512 and 32 are the pre-defined tensor shapes, which can be executed sequentially on the NPU, while the sub-tensor with a dynamic size of 56 is offloaded to the GPU. In addition to multi-sequence-length cutting, HeteroLLM can also employ a hybrid approach, which combines row-cutting and sequence-length cutting (*hybrid-cutting*). In this configuration, HeteroLLM continues to use padding for NPU computation while offloading a portion of the computational load to the GPU backend based on the row dimension. Through these elaborate tensor partition approaches, HeteroLLM can overlap the execution time between the GPU and NPU, and further select the optimal partition strategies according to the different sequence lengths.

4.1.2. Tensor Partition During the Decoding Phase

In the decoding phase, the primary bottleneck shifts from computation to memory bandwidth. Parallel execution on both the GPU and NPU can leverage the whole memory bandwidth in SoC, which is larger than the memory bandwidth of a single processor (*Memory-❶*). During the decoding phase, the sequence length of the input token is fixed—typically one for standard decoding and n for speculative decoding [68]. We can pre-generate the NPU graph using the designated decoding tensor shape and employ a **row-cutting** strategy for tensor partition. Unlike the row-cutting strategy used in the prefill phase, which aims to balance computational load between the GPU and NPU, the approach in the decoding phase focuses on maximizing SoC memory bandwidth as well as minimizing potential memory contention. To achieve optimal partition, we employ a partition solver (introduced in Section 4.3) to determine the best partition ratio between the NPU and GPU.

4.2. Fast Synchronization

While GPU-NPU parallelism can decrease execution time for certain operators, it may introduce new overhead due to GPU and NPU synchronization (*GPU-❷: high-cost synchronization*). This synchronization overhead becomes more pronounced during the decoding phase, where the execution time for a single operation (e.g., Matmul) is reduced to hundreds of microseconds, while a synchronization operation incurs an overhead at least 400 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 HeteroLLM 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 the different 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.

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 11. In the prefill phase, the NPU exhibits superior computational capability, making it NPU-dominant. HeteroLLM 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.

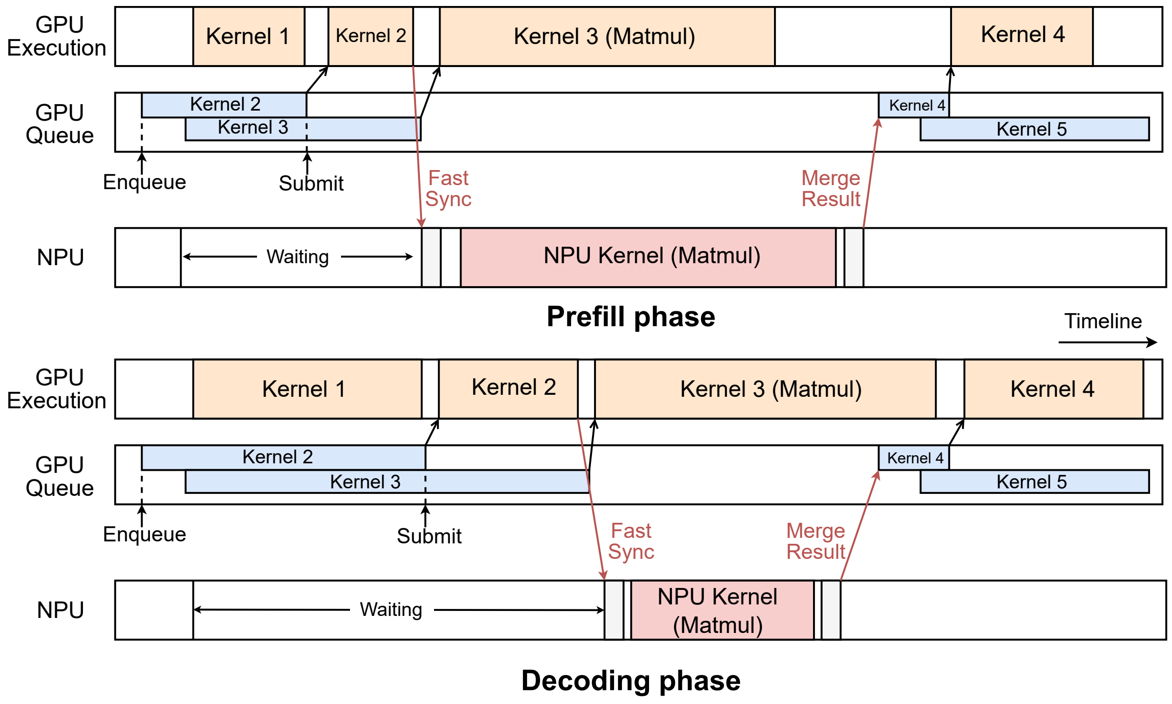

Figure 11. Fast synchronization during the prefill and decoding phases.

Conversely, in the decoding phase, the GPU outperforms the NPU because GPU kernel implementations obtain more stable and efficient memory bandwidth, making this phase GPU-dominant. Here, we overlap NPU execution with the GPU execution. Once the NPU task is completed, we promptly submit the next GPU kernel to the GPU queue. The inherent queue ordering guarantees synchronization for GPU kernels, eliminating any additional GPU submission overhead during the decoding phase.

4.3. Putting It All Together

Figure 12 illustrates the overall architecture of HeteroLLM. Given a large language model, our tensor partition solver first identifies the various tensor shapes utilized by the model across several predefined sequence lengths. It then cooperates with the performance profiler to determine the partition strategy of each tensor, and then partitions each tensor with the most suitable ratio. Finally, it generates the computation graph for different backends, conducting the execution of inference engine at runtime.

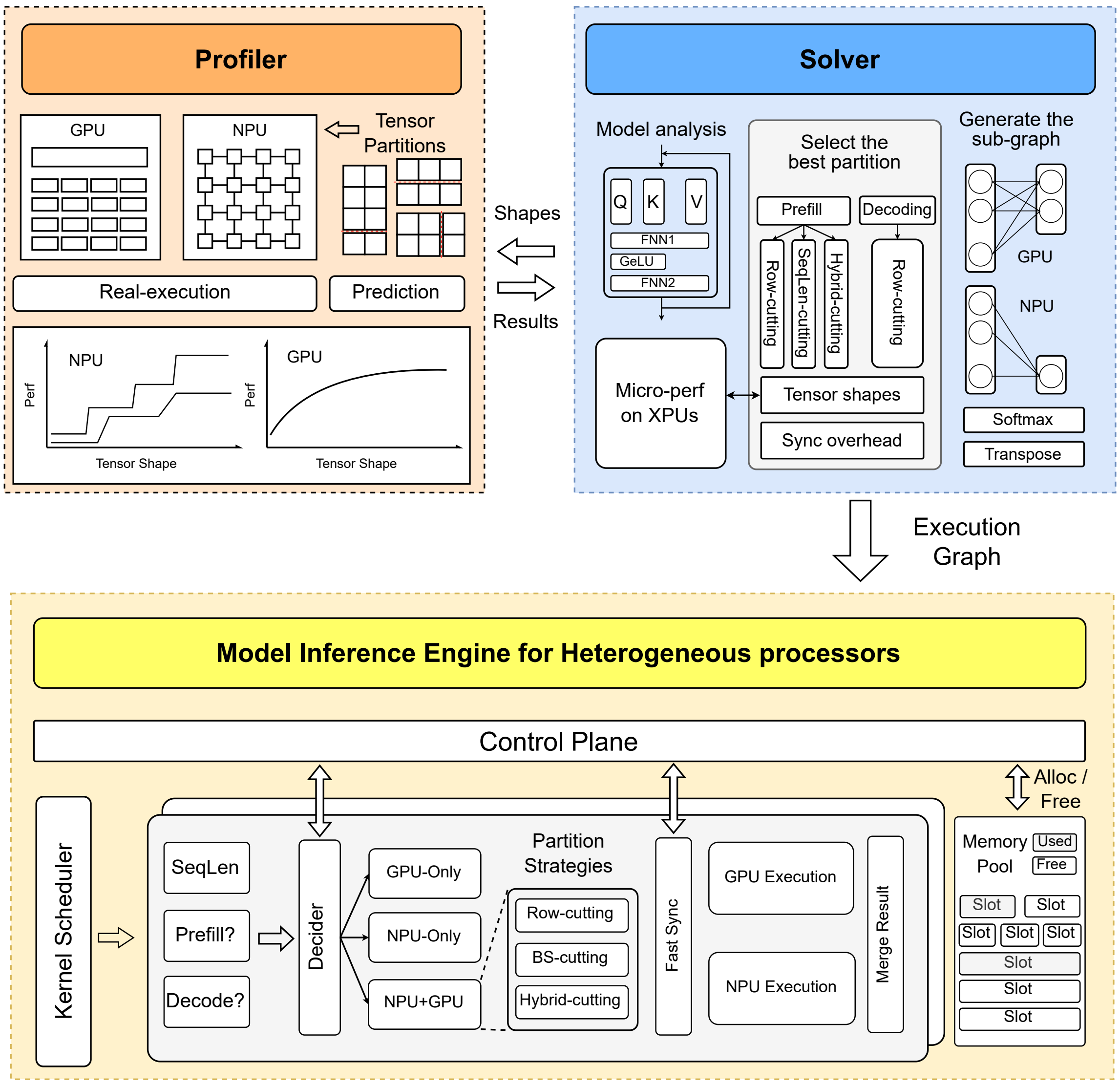

Figure 12. The overall architecture of HeteroLLM contains three components: Profiler, Solver and Inference Engine.

Performance Profiler. To determine the optimal partition solution, the solver works in conjunction with a performance profiler tailored for heterogeneous processors. Our profiler operates in two modes: real-execution and prediction. In the real-execution mode, the profiler executes the target operator with various tensor shapes on actual hardware, gathering precise performance metrics for both GPUs and NPUs. Although this mode is time-consuming, it can be conducted offline. Moreover, the NPU's stage performance characteristic facilitates effective pruning of the tensor partition search space, with constraints requiring row partitions to be aligned to 256 and sequence length partitions to 32, thereby reducing the number of candidate partitions. In addition to the real-execution mode, we provide a prediction mode. Due to the inherent fluctuation in hardware performance, minor inaccuracies in performance results across different backends are tolerable for our solver. Using traditional machine learning techniques, such as decision tree regression, we can predict NPU performance across different tensor shapes. Conversely, given that GPU performance is more stable and less dependent on tensor shapes, we easily estimate GPU execution time in compute-intensive scenarios using a fixed TFLOPS rate.

Tensor Partition Solver. After obtaining the hardware performance results across various tensor shapes, the solver must determine the optimal partition strategy, taking into account GPU-only, NPU-only and GPU-NPU parallelism. The objective function to be optimized by the solver is illustrated in the following equation.

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

In the context of GPU-NPU parallelism, the solver traverses all possible partition solutions for the GPU and NPU, and retrieves performance results from the profiler. Commonly, the execution times for the GPU and NPU cannot be perfectly overlapped, so the solver uses the maximum execution time of these two backends as the actual computation time. Beyond computation time, the solver also accounts for synchronization overhead, including kernel submission and activation transfer between GPU and NPU memory. The cost associated with kernel submission is further influenced by whether the execution is GPU-dominant or NPU-dominant. For certain tensor sizes where GPU-NPU parallelism does not yield any performance benefits, the solver opts not to partition the tensor, instead selecting the optimal backend for execution.

Inference Engine. During execution, a control plane decider determines whether a kernel is executed on the NPU backend, the GPU backend, or using NPU-GPU parallelism, based on the inference phases and sequence length. When two adjacent kernels are allocated to different backends, our HeteroLLM 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 HeteroLLM engine also 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 have implemented a prototype of HeteroLLM based on PPL [32], a SOTA mobile-side inference engine that supports both CPU and GPU. HeteroLLM extends PPL by incorporating NPU support (using QNN-NPU library) and implements both layer-level and tensor-level heterogeneous execution across GPU and NPU. As for the model quantization, HeteroLLM employs the W4A16 (weight-only) quantization[1] [69–73] which balances the model accuracy (FLOAT computation) and storage overhead (INT4 for weight storage). We evaluate the performance of HeteroLLM on one of the most powerful mobile SoCs: Qualcomm 8 Gen 3. Even when compared to inference engines [26,29,51] that exploit model/activation sparsity or rely on INT-only NPU calculations, which can potentially compromise the model accuracy, HeteroLLM demonstrates comparable performance through more efficient NPU utilization and GPU-NPU parallelism, without sacrificing any accuracy. Moreover, we believe that our techniques could also enhance performance for sparse inference, which remains an orthogonal aspect to our work.

In our evaluation, we mainly compare with inference engines using dense computation without sacrificing the inference accuracy, such as llama.cpp (CPU), MLC (GPU) and MNN (GPU). HeteroLLM achieves significant performance improvements through layer-level heterogeneous execution (Hetero-layer) during the prefill phase, with enhancements of 25.1×, 7.27× and 3.18× compared to above inference engines respectively. Additionally, HeteroLLM gains a further 40% improvement via tensor-level heterogeneous execution (Hetero-tensor). During the decoding phase, HeteroLLM achieves a performance increase of up to 23.4% by fully utilizing the SoC's memory bandwidth.

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

5.2. Prefill Performance

We first evaluate the prefill performance of HeteroLLM. As mobile NPU only supports static graph execution, the evaluation is conducted from two perspectives: regular sequence lengths that fit in pre-defined graphs and arbitrary sequence lengths that do not align with the NPU's static graph.

5.2.1. Aligned Sequence Length

We compare prefill performance of HeteroLLM with MNN-OpenCL, llama.cpp, MLC and PPL-OpenCL across sequence length of 64, 256, 1024. As shown in Figure 13, when running Llama-8B model with W4A16 quantization under sequence length of 256, Hetero-layer achieves 5.85×, 24.9×, 5.64×, 2.99× speedup compared to above frameworks. For sequence lengths 64 and 1024, Hetero-layer can speed up 3.67-14.36× and 3.17-25.12× respectively. These performance improvements result from utilizing the NPU as a substitute for the GPU in executing computationally demanding operators (e.g., Matmul), given that a well-optimized NPU, with a more suitable tensor order, significantly outperforms the GPU in most scenarios.

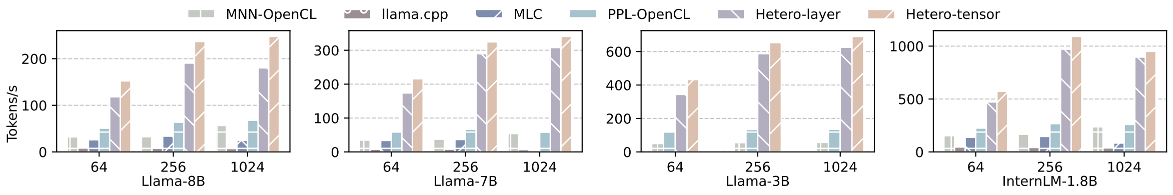

Figure 13. Prefill speed of different models under different prompt lengths.

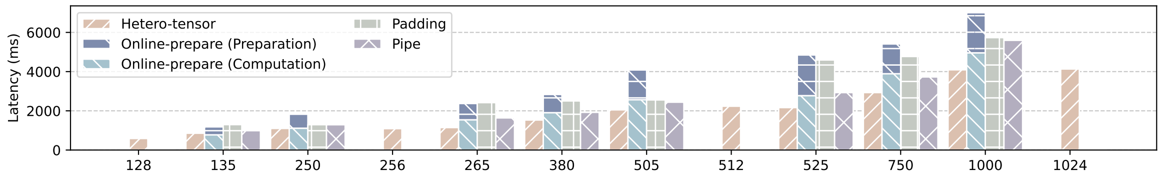

Figure 14. Prefill Latency of different methods when running Llama-8B with misaligned sequence length.

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). Under sequence length of 1024, Hetero-tensor achieves 34.5×, 9.99× and 4.36× performance improvement over llama.cpp, MLC and MNN-OpenCL respectively. The prefill speed of Hetero-tensor reaches 247.9 tokens per second on Llama-8B and is up to 1092 tokens per second on InternLM-1.8B. This is because Hetero-tensor takes the performance characteristics of both GPU and NPU into consideration, which allows the GPU to compensate for the NPU's performance degradation with certain tensor shapes (e.g., FFN-down), thereby unleashing the potential of heterogeneous execution for GPU and NPU. Moreover, Hetero-tensor also utilizes the fast synchronization mechanism to overcome the synchronization overhead between GPU and NPU, which we will evaluate in Section 5.4. For other models, Hetero-layer and Hetero-tensor also show significant performance improvements. On Llama-7B, Hetero-tensor is averagely 6.05× faster than MNN-OpenCL, and 6.63× faster than MLC. Similarly, on the Llama-3B model, Hetero-tensor achieves an average speed-up of 10.2× over MNN-OpenCL and 3.51× over PPL-OpenCL.

Compared with other inference engines [26,29] that just leverage the NPU for sparse computation (INT), HeteroLLM exploits the dense computational capabilities (FLOAT) of the NPU. Thanks to the involvement of GPU, HeteroLLM even achieves 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 MLLM-npu [26] attains only 564 tokens/s under the same model size (relies on the sparse activation).

5.2.2. Misaligned Sequence Length

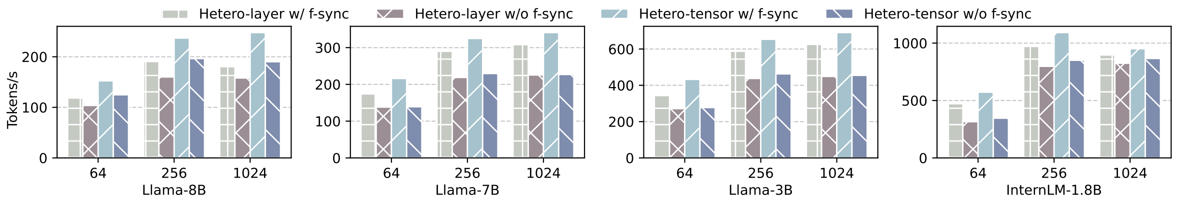

Figure 15. Prefill speed of Hetero-layer and Hetero-tensor with and without fast synchronization under sequence lengths of 64, 256 and 1024.

As current mobile NPUs only support static computation graphs, and it is impossible to prepare a graph for every sequence length in advance. An intuitive way is to create new graphs for every request with different sequence lengths in the runtime. This method is designated as "Online-prepare". Another approach is to preload several graphs in standard sizes (e.g., 128, 256, 512) and pad misaligned input to the closest standard size. Moreover, misaligned input can also be segmented into multiple parts using a multi-sequence-length cutting approach (described in Section 4.1.1). The portion exceeding the standard size, referred to as the *margin*, is processed with a smaller standard-sized graph and executed sequentially on the NPU. These two methods are referred to as "Padding" and "Pipe" respectively.

MLLM-NPU addresses the challenge of static computation graphs on NPU by splitting the input sequence into fixed-size chunks, a method referred to as "Chunked Prefill". 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 [26]. In contrast, Pipe (i.e., multi-sequence-length cutting without GPU support) chooses different standard sizes dynamically and has less overhead in graph loading. We adopt Pipe as the best solution for NPU-only prefill.

Figure 14 presents the performance of Hetero-tensor under misaligned sequence lengths running Llama-8B and compares it with the above three methods. The standard sizes are set to be powers of 2 between 32 and 1024. We record both the graph preparation time and the computation time of Online-prepare. The results show that Hetero-tensor outperforms other methods under every sequence length and achieves 2.24×, 2.21× and 1.35× speedup compared to Online-prepare, Padding and Pipe under misaligned sequence length of 525. Online-prepare has the longest latency in most cases, because graph preparation time increases with sequence length and is related to the number of graphs (typically 4 graphs). Under sequence length 135, preparation time is 408.4 ms and accounts for 34.6% of the total latency. This overhead increases to 2050 ms as the sequence length extends to 1000. The latency of Padding increases in a stepwise manner rather than linearly. It introduces extra latency and causes a waste of computation resources. When the misaligned sequence length slightly exceeds a standard size, the latency of Padding is on average 1.91× longer than that of Hetero-tensor. Pipe compensates for the overhead of Padding as it fits the *margin* in a smaller graph and adds less padding. Compared to Pipe, Hetero-tensor is still capable of reducing the latency by 13.2%-30.1%.

The performance improvement of Hetero-tensor is attributed to its ability to parallelize computations across heterogeneous accelerators, and its flexibility in adopting various tensor partitioning strategies. For every matrix multiplication with misaligned sizes, Hetero-tensor decides the partition strategy according to the computational power of NPU and GPU. When the ratio of the margin to the standard size closely aligns with the ratio of computational power between the GPU and NPU, the computation is partitioned by sequence length between the GPU and NPU. If the *margin* is too small to fully utilize the computational power of GPU, Hetero-tensor applies hybrid-cutting to partition the multiplication in both row and sequence length.

5.3. Decoding Performance

Figure 16 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. On InternLM-1.8B, Hetero-tensor achieves 1.94× speedup compared to MNN-OpenCL and 2.62× speedup compared to MLC. As for Hetero-layer, since NPU computation is typically slower than GPU for small sequence length, it always chooses the GPU in decoding layers and performs similarly to PPL-OpenCL.

Hetero-tensor is the only framework that utilizes both GPU and NPU in decoding phase. When NPU and GPU are running concurrently, the memory bandwidth increases from 43.3 GB/s (only GPU) to 59.1 GB/s. Hetero-tensor uses row-cutting strategy to partition the computation and maximizes SoC memory bandwidth. As a result, Hetero-tensor is 23.4% faster than PPL-OpenCL on Llama-8B, 8.52% faster on Llama-3B and 13.38% faster on InternLM-1.8B.

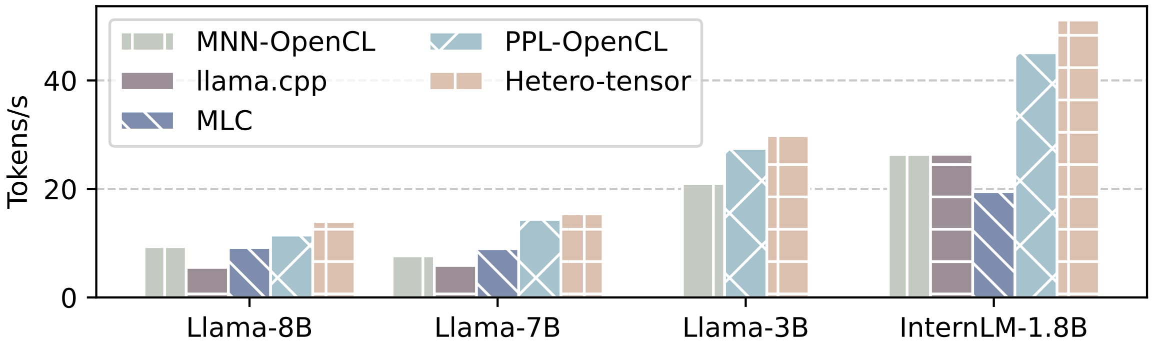

Figure 16. Decoding rate of MNN-OpenCL, llama.cpp, MLC, PPL-OpenCL and Hetero-tensor on Llama-8B, Llama-7B, Llama-3B and InternLM-1.8B. The prompt sequence length is 256.

5.4. Effect of Fast Synchronization

Prefill Performance: Figure 15 presents the prefill performance of Hetero-layer and Hetero-tensor with and without fast synchronization on various models, under sequence lengths of 64, 256 and 1024. On Llama-8B, the use of fast synchronization improves the performance of Hetero-layer and Hetero-tensor by 15.8% and 24.3% on average. Under a sequence length of 256, the prefill speed of Hetero-tensor increases from 196.44 tokens/s to 236.92 tokens/s. For Llama-7B and InternLM-1.8B, they achieve performance improvements of 49.0% and 34.5% with Hetero-tensor and 31.7% and 26.7% with Hetero-layer, respectively. Hetero-tensor is more susceptible to the synchronization cost, which may disrupt the 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.

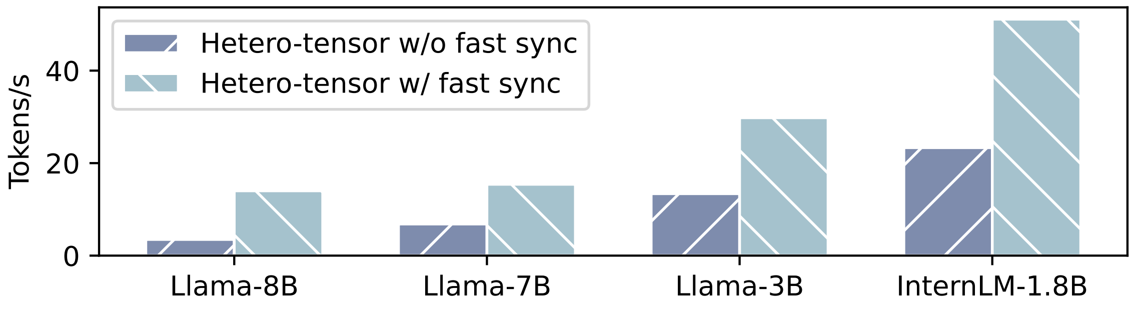

Figure 17. Decoding rate of Hetero-tensor running Llama-8B, Llama-7B, Llama-3B and InternLM-1.8B with and without fast synchronization. The prompt sequence length is 256.

5.5. GPU Performance Interference

To evaluate the impact of HeteroLLM on system-level image rendering as well as interference with other GPU applications, we conduct experiments by running PPL-OpenCL, 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 their default values.

As shown in Figure 18, when running concurrently with the game, the prefill speed of Hetero-layer and Hetero-tensor decreases by 9.57% and 7.26% respectively, while Hetero-tensor (w/ game) is still 15.3% faster even compared to Hetero-layer (w/o game). The frame per second (FPS) of the game is not affected by the execution of Hetero-layer and Hetero-tensor, keeping steady at 60 FPS, and all the operations in the game are still smooth. In contrast, the FPS of the game significantly degrades (dropping to zero), when run concurrently with PPL-OpenCL. This occurs because GPU kernels initiated by the PPL runtime fully occupy the GPU submission queue, preventing the game's rendering tasks from completing within their designated time frame. In contrast, Hetero-layer and Hetero-tensor allocate only a small portion of the computation workload to GPU, leaving sufficient capacity for GPU to handle the game's rendering tasks in a timely manner.

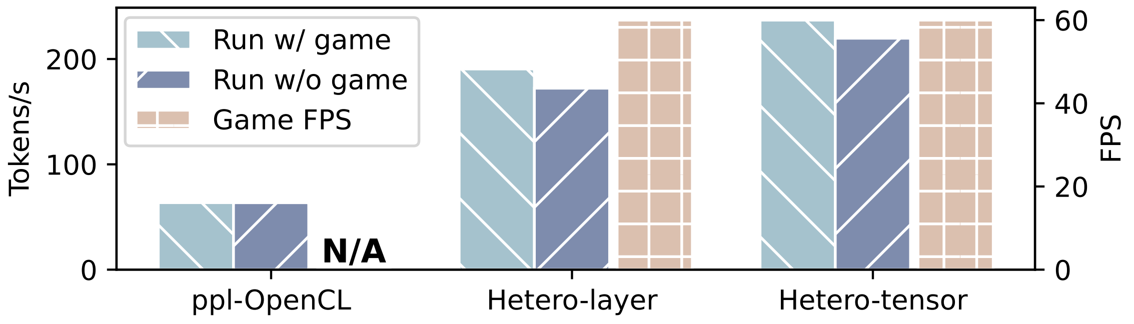

Figure 18. Prefill speed of PPL-OpenCL, Hetero-layer and Hetero-tensor when running with and without game and frame per second (FPS) of the game. The model is Llama-8B and the sequence length is 256.

5.6. Energy Consumption

Figure 19 illustrates the power and energy consumption of PPL-OpenCL, Hetero-layer and Hetero-tensor during the prefill phase of Llama-8B with a sequence length of 256. Hetero-layer demonstrates the lowest power consumption at 2.23W, primarily due to its reliance on the NPU for most computations. The power consumption of Hetero-tensor is only 23.2% higher than that of Hetero-layer and is reduced by 36.7% compared to PPL-OpenCL. The total energy consumption during the prefill phase is determined by the product of power and execution time. Hetero-tensor achieves a prefill speed that is 24.4% faster than Hetero-layer and 2.72× faster than PPL-OpenCL. As a result, Hetero-tensor consumes only 3.3% more energy than Hetero-layer, while providing an energy efficiency that is 5.87× better than PPL-OpenCL.

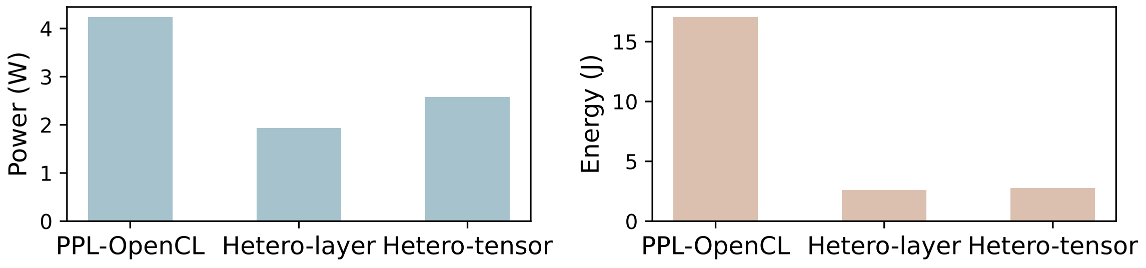

Figure 19. Power and energy consumption of PPL-OpenCL, Hetero-layer and Hetero-tensor when running Llama-8B prefill phase with a sequence length of 256.

6. Discussion

Model Quantization. HeteroLLM 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 [26,29,51] require quantization of both activations and weights from FLOAT to INT, which may compromise inference accuracy.

Shared Memory Between GPUs and NPUs. Current mobile SoCs (e.g., Apple M/A series, Qualcomm Snapdragon series) support a unified address space for CPU, GPU and NPU. In our implementation, we have successfully established shared memory between the CPU and GPU using OpenCL, and between the CPU and NPU using QNN APIs. Additionally, by employing the 'CL_MEM_USE_HOST_PTR' flag, we can also map the NPU's shared memory to the GPU, thereby establishing shared memory between the GPU and NPU. We recommend that mobile SoC vendors provide native APIs to allocate unified memory for all heterogeneous processors.

7. Conclusions

This paper introduces HeteroLLM, the fastest LLM inference engine optimized for heterogeneous processors in modern mobile SoCs. We provide an in-depth analysis of the GPU and NPU hardware architecture, highlighting the NPU's tensor shape-sensitive performance characteristics. Unfortunately, certain layers in contemporary LLMs, such as FFN-down, are processed less efficiently by NPUs, creating significant bottlenecks. HeteroLLM mitigates these inefficiencies by leveraging GPUs to complement NPU computational limitations. It also addresses challenges like the underutilization of SoC memory bandwidth and static graph constraints. Furthermore, this paper offers new insights into designing more efficient edge AI accelerators and heterogeneous SoCs.

References

1. OpenAI. Introducing ChatGPT. https://openai.com/index/chatgpt/, 2024. Referenced January 2024.
2. McIntosh, T.R.; Susnjak, T.; Liu, T.; Watters, P.; Halgamuge, M.N. 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**.
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. GLM, T.; Zeng, A.; Xu, B.; Wang, B.; Zhang, C.; Yin, D.; Rojas, D.; Feng, G.; Zhao, H.; Lai, H.; et al. ChatGLM: A Family of Large Language Models from GLM-130B to GLM-4 All Tools, 2024, [arXiv:id='cs.CL' full_name='Computation and Language' is_active=True alt_name='cmp-lg' in_archive='cs' is_general=False description='Covers natural language processing. Roughly includes material in ACM Subject Class I.2.7. Note that work on artificial languages (programming languages, logics, formal systems) that does not explicitly address natural-language issues broadly construed (natural-language processing, computational linguistics, speech, text retrieval, etc.) is not appropriate for this area.'/2406.12793].

5. Hong, W.; Wang, W.; Lv, Q.; Xu, J.; Yu, W.; Ji, J.; Wang, Y.; Wang, Z.; Dong, Y.; Ding, M.; et al. Cogagent: A visual language model for gui agents. In Proceedings of the Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2024, pp. 14281–14290.
6. You, K.; Zhang, H.; Schoop, E.; Weers, F.; Swearngin, A.; Nichols, J.; Yang, Y.; Gan, Z. Ferret-ui: Grounded mobile ui understanding with multimodal llms. In Proceedings of the European Conference on Computer Vision. Springer, 2025, pp. 240–255.
7. Wang, J.; Xu, H.; Jia, H.; Zhang, X.; Yan, M.; Shen, W.; Zhang, J.; Huang, F.; Sang, J. Mobile-Agent-v2: Mobile Device Operation Assistant with Effective Navigation via Multi-Agent Collaboration. *arXiv preprint arXiv:2406.01014* **2024**.
8. Wang, J.; Xu, H.; Ye, J.; Yan, M.; Shen, W.; Zhang, J.; Huang, F.; Sang, J. Mobile-Agent: Autonomous Multi-Modal Mobile Device Agent with Visual Perception. *arXiv preprint arXiv:2401.16158* **2024**.
9. Yang, Z.; Teng, J.; Zheng, W.; Ding, M.; Huang, S.; Xu, J.; Yang, Y.; Hong, W.; Zhang, X.; Feng, G.; et al. CogVideoX: Text-to-Video Diffusion Models with An Expert Transformer. *arXiv preprint arXiv:2408.06072* **2024**.
10. Wang, P.; Bai, S.; Tan, S.; Wang, S.; Fan, Z.; Bai, J.; Chen, K.; Liu, X.; Wang, J.; Ge, W.; et al. Qwen2-VL: Enhancing Vision-Language Model's Perception of the World at Any Resolution. *arXiv preprint arXiv:2409.12191* **2024**.
11. Bai, J.; Bai, S.; Yang, S.; Wang, S.; Tan, S.; Wang, P.; Lin, J.; Zhou, C.; Zhou, J. Qwen-VL: A Versatile Vision-Language Model for Understanding, Localization, Text Reading, and Beyond. *arXiv preprint arXiv:2308.12966* **2023**.
12. Hong, W.; Ding, M.; Zheng, W.; Liu, X.; Tang, J. CogVideo: Large-scale Pretraining for Text-to-Video Generation via Transformers. *arXiv preprint arXiv:2205.15868* **2022**.
13. Zhang, P.; Dong, X.; Zang, Y.; Cao, Y.; Qian, R.; Chen, L.; Guo, Q.; Duan, H.; Wang, B.; Ouyang, L.; et al. Internlm-xcomposer-2.5: A versatile large vision language model supporting long-contextual input and output. *arXiv preprint arXiv:2407.03320* **2024**.
14. Xue, Y.; Liu, Y.; Nai, L.; Huang, J. V10: Hardware-Assisted NPU Multi-tenancy for Improved Resource Utilization and Fairness. In Proceedings of the Proceedings of the 50th Annual International Symposium on Computer Architecture, New York, NY, USA, 2023; ISCA '23. https://doi.org/10.1145/3579371.3589059.
15. Xue, Y.; Liu, Y.; Huang, J. System Virtualization for Neural Processing Units. In Proceedings of the Proceedings of the 19th Workshop on Hot Topics in Operating Systems, New York, NY, USA, 2023; HOTOS '23, p. 80–86. https://doi.org/10.1145/3593856.3595912.
16. Kwon, H.; Lai, L.; Pellauer, M.; Krishna, T.; Chen, Y.H.; Chandra, V. Heterogeneous dataflow accelerators for multi-DNN workloads. In Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2021, pp. 71–83.
17. Odema, M.; Chen, L.; Kwon, H.; Al Faruque, M.A. SCAR: Scheduling Multi-Model AI Workloads on Heterogeneous Multi-Chiplet Module Accelerators. In Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO), 2024, pp. 565–579. https://doi.org/10.1109/MICRO61859.2024.00049.
18. Jouppi, N.P.; Young, C.; Patil, N.; Patterson, D.; Agrawal, G.; Bajwa, R.; Bates, S.; Bhatia, S.; Boden, N.; Borchers, A.; et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In Proceedings of the Proceedings of the 44th Annual International Symposium on Computer Architecture, New York, NY, USA, 2017; ISCA '17, pp. 1–12. https://doi.org/10.1145/3079856.3080246.
19. Hsu, K.C.; Tseng, H.W. Accelerating applications using edge tensor processing units. In Proceedings of the Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New York, NY, USA, 2021; SC '21. https://doi.org/10.1145/3458817.3476177.
20. Xue, Y.; Liu, Y.; Nai, L.; Huang, J. Hardware-Assisted Virtualization of Neural Processing Units for Cloud Platforms. *arXiv preprint arXiv:2408.04104* **2024**.
21. George, B.; Omer, O.J.; Choudhury, Z.; V, A.; Subramoney, S. A Unified Programmable Edge Matrix Processor for Deep Neural Networks and Matrix Algebra. *ACM Trans. Embed. Comput. Syst.* **2022**, *21*. https://doi.org/10.1145/3524453.
22. Jiang, Y.; Zhu, Y.; Lan, C.; Yi, B.; Cui, Y.; Guo, C. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), 2020, pp. 463–479.
23. Hsu, K.C.; Tseng, H.W. Simultaneous and Heterogenous Multithreading. In Proceedings of the Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, 2023, pp. 137–152.

24. Hsu, K.C.; Tseng, H.W. SHMT: Exploiting Simultaneous and Heterogeneous Parallelism in Accelerator-Rich Architectures. *IEEE Micro* **2024**.
25. Qualcomm. Snapdragon 8Gen3 - Mobile Platform ignites endless possibilities. https://www.qualcomm.com/products/mobile/snapdragon/smartphones, 2023-2024. Referenced December 2024.
26. Xu, D.; Zhang, H.; Yang, L.; Liu, R.; Huang, G.; Xu, M.; Liu, X. Empowering 1000 tokens/second on-device llm prefilling with mllm-npu. *arXiv preprint arXiv:2407.05858* **2024**.
27. Xu, D.; Xu, M.; Wang, Q.; Wang, S.; Ma, Y.; Huang, K.; Huang, G.; Jin, X.; Liu, X. Mandheling: Mixed-precision on-device dnn training with dsp offloading. In Proceedings of the Proceedings of the 28th Annual International Conference on Mobile Computing And Networking, 2022, pp. 214–227.
28. Gerogiannis, G.; Aananthakrishnan, S.; Torrellas, J.; Hur, I. HotTiles: Accelerating SpMM with Heterogeneous Accelerator Architectures. In Proceedings of the 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2024, pp. 1012–1028. https://doi.org/10.1109/HPCA57654.2024.00081.
29. Xue, Z.; Song, Y.; Mi, Z.; Chen, L.; Xia, Y.; Chen, H. PowerInfer-2: Fast Large Language Model Inference on a Smartphone. *arXiv preprint arXiv:2406.06282* **2024**.
30. Kumar, T.; Ankner, Z.; Spector, B.F.; Bordelon, B.; Muennighoff, N.; Paul, M.; Pehlevan, C.; Ré, C.; Raghunathan, A. Scaling laws for precision. *arXiv preprint arXiv:2411.04330* **2024**.
31. Xiao, C.; Cai, J.; Zhao, W.; Zeng, G.; Han, X.; Liu, Z.; Sun, M. Densing Law of LLMs, 2024, [arXiv:cs.AI/2412.04315].
32. Sensetime. PPL - High-performance deep-learning inference engine for efficient AI inferencing. https://github.com/OpenPPL/ppl.nn, 2018. Referenced December 2024.
33. Qualcomm. QNN. https://www.qualcomm.com/developer/software/qualcomm-ai-engine-direct-sdk, 2024. Referenced December 2024.
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. Lv, C.; Niu, C.; Gu, R.; Jiang, X.; Wang, Z.; Liu, B.; Wu, Z.; Yao, Q.; Huang, C.; Huang, P.; et al. Walle: An End-to-End, General-Purpose, and Large-Scale Production System for Device-Cloud Collaborative Machine Learning. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), Carlsbad, CA, 2022; pp. 249–265.
36. Nanoreview. Online. https://nanoreview.net/en/soc-list/rating, 2023-2024. Referenced December 2024.
37. Liu, Y.; Li, H.; Cheng, Y.; Ray, S.; Huang, Y.; Zhang, Q.; Du, K.; Yao, J.; Lu, S.; Ananthanarayanan, G.; et al. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In Proceedings of the Proceedings of the ACM SIGCOMM 2024 Conference, New York, NY, USA, 2024; ACM SIGCOMM '24, p. 38–56. https://doi.org/10.1145/3651890.3672274.
38. Lee, W.; Lee, J.; Seo, J.; Sim, J. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), Santa Clara, CA, 2024; pp. 155–172.
39. Ainslie, J.; Lee-Thorp, J.; de Jong, M.; Zemlyanskiy, Y.; Lebrón, F.; Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245* **2023**.
40. Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C.H.; Gonzalez, J.; Zhang, H.; Stoica, I. Efficient memory management for large language model serving with pagedattention. In Proceedings of the Proceedings of the 29th Symposium on Operating Systems Principles, 2023, pp. 611–626.
41. Yu, G.I.; Jeong, J.S.; Kim, G.W.; Kim, S.; Chun, B.G. Orca: A distributed serving system for {Transformer-Based} generative models. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), 2022, pp. 521–538.
42. Fu, Y.; Xue, L.; Huang, Y.; Brabete, A.O.; Ustiugov, D.; Patel, Y.; Mai, L. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), Santa Clara, CA, 2024; pp. 135–153.
43. Li, Z.; Zheng, L.; Zhong, Y.; Liu, V.; Sheng, Y.; Jin, X.; Huang, Y.; Chen, Z.; Zhang, H.; Gonzalez, J.E.; et al. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), Boston, MA, 2023; pp. 663–679.
44. Zhang, C.; Yu, M.; Wang, W.; Yan, F. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19), Renton, WA, 2019; pp. 1049–1062.

45. Zhuang, D.; Zheng, Z.; Xia, H.; Qiu, X.; Bai, J.; Lin, W.; Song, S.L. MonoNN: Enabling a New Monolithic Optimization Space for Neural Network Inference Tasks on Modern GPU-Centric Architectures. In Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), Santa Clara, CA, 2024; pp. 989–1005.
46. Zhong, Y.; Liu, S.; Chen, J.; Hu, J.; Zhu, Y.; Liu, X.; Jin, X.; Zhang, H. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), Santa Clara, CA, 2024; pp. 193–210.
47. Shubha, S.S.; Shen, H. AdaInf: Data Drift Adaptive Scheduling for Accurate and SLO-guaranteed Multiple-Model Inference Serving at Edge Servers. In Proceedings of the Proceedings of the ACM SIGCOMM 2023 Conference, 2023, pp. 473–485.
48. Patel, P.; Choukse, E.; Zhang, C.; Shah, A.; Goiri, Í.; Maleki, S.; Bianchini, R. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In Proceedings of the 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), 2024, pp. 118–132. https://doi.org/10.1109/ISCA59077.2024.00019.
49. Agrawal, A.; Kedia, N.; Panwar, A.; Mohan, J.; Kwatra, N.; Gulavani, B.; Tumanov, A.; Ramjee, R. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), Santa Clara, CA, 2024; pp. 117–134.
50. Yi, R.; Li, X.; Qiu, Q.; Lu, Z.; Zhang, H.; Xu, D.; Yang, L.; Xie, W.; Wang, C.; Xu, M. mllm: fast and lightweight multimodal LLM inference engine for mobile and edge devices, 2023.
51. 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.
52. 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.
53. Microsoft. Accelerated Edge Machine Learning. https://onnxruntime.ai/, 2020. Referenced April 2023.
54. Apple. A18. https://nanoreview.net/en/soc/apple-a18, 2024. Referenced December 2024.
55. Huawei. Kirin-9000. https://www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-9000, 2020. Referenced December 2024.
56. Tencent. NCNN - High-performance neural network inference computing framework optimized for mobile platforms. https://github.com/Tencent/ncnn, 2017. Referenced December 2024.
57. Intel. OpenVino - Open-source toolkit for optimizing and deploying deep learning models from cloud to edge. https://docs.openvino.ai/2024/index.html, 2018. Referenced December 2024.
58. Google. TensorFlow Lite - Google's high-performance runtime for on-device AI. https://tensorflow.google.cn/lite, 2017. Referenced December 2024.
59. Liu, Y.; Wang, Y.; Yu, R.; Li, M.; Sharma, V.; Wang, Y. Optimizing CNN Model Inference on CPUs. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19), Renton, WA, 2019; pp. 1025–1040.
60. Iyer, V.; Lee, S.; Lee, S.; Kim, J.J.; Kim, H.; Shin, Y. Automated Backend Allocation for Multi-Model, On-Device AI Inference. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* **2023**, *7*, 1–33.
61. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. TensorFlow: A System for Large-Scale Machine Learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, 2016; pp. 265–283.
62. Team, T.T.D.; Al-Rfou, R.; Alain, G.; Almahairi, A.; Angermueller, C.; Bahdanau, D.; Ballas, N.; Bastien, F.; Bayer, J.; Belikov, A.; et al. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688* **2016**.
63. Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, 2018; pp. 578–594.
64. Huawei. HUAWEI HiAI Engine - About the Service. https://developer.huawei.com/consumer/en/doc/hiai-References/ir-overview-0000001052569365, 2024. Referenced December 2024.
65. Zhu, H.; Wu, R.; Diao, Y.; Ke, S.; Li, H.; Zhang, C.; Xue, J.; Ma, L.; Xia, Y.; Cui, W.; et al. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), Carlsbad, CA, 2022; pp. 233–248.

66. Zheng, L.; Jia, C.; Sun, M.; Wu, Z.; Yu, C.H.; Haj-Ali, A.; Wang, Y.; Yang, J.; Zhuo, D.; Sen, K.; et al. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 2020, pp. 863–879.
67. Ragan-Kelley, J.; Barnes, C.; Adams, A.; Paris, S.; Durand, F.; Amarasinghe, S. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* **2013**, *48*, 519–530.
68. Miao, X.; Oliaro, G.; Zhang, Z.; Cheng, X.; Wang, Z.; Zhang, Z.; Wong, R.Y.Y.; Zhu, A.; Yang, L.; Shi, X.; et al. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In Proceedings of the Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, 2024, pp. 932–949.
69. Frantar, E.; Ashkboos, S.; Hoefler, T.; Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323* **2022**.
70. Lin, J.; Tang, J.; Tang, H.; Yang, S.; Chen, W.M.; Wang, W.C.; Xiao, G.; Dang, X.; Gan, C.; Han, S. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. *Proceedings of Machine Learning and Systems* **2024**, *6*, 87–100.
71. Li, J.; Xu, J.; Huang, S.; Chen, Y.; Li, W.; Liu, J.; Lian, Y.; Pan, J.; Ding, L.; Zhou, H.; et al. Large language model inference acceleration: A comprehensive hardware perspective. *arXiv preprint arXiv:2410.04466* **2024**.
72. Cheng, W.; Cai, Y.; Lv, K.; Shen, H. Teq: Trainable equivalent transformation for quantization of llms. *arXiv preprint arXiv:2310.10944* **2023**.
73. Park, G.; Park, B.; Kim, M.; Lee, S.; Kim, J.; Kwon, B.; Kwon, S.J.; Kim, B.; Lee, Y.; Lee, D. Lut-gemm: Quantized matrix multiplication based on luts for efficient inference in large-scale generative language models. *arXiv preprint arXiv:2206.09557* **2022**.