Agent.xpu: Efficient Scheduling of Agentic LLM Workloads on Heterogeneous SoC

Anonymous Author(s)

Abstract

The rise of personal LLM agents introduces workloads with heterogeneous temporal demands—reactive requests require responsiveness, while proactive tasks benefit from throughput-oriented scheduling. On-device deployment on heterogeneous SoCs (with CPU, iGPU, and NPU) promises low-latency, private execution, yet faces unique challenges: mismatches between LLM dynamism and accelerator rigidity, shared-memory contention, and deficient runtime abstractions. We present Agent.xpu, a workload-aware serving system that co-designs scheduling and execution with hetero-SoC characteristics. Agent.xpu introduces a heterogeneous execution graph (HEG) for elastic kernel mapping, disaggregates prefill and decode pipelines across NPU and iGPU, and implements fine-grained kernel-level preemption and slack-aware piggybacking. Our prototype on Intel Core Ultra SoCs shows that Agent.xpu improves proactive throughput by up to 2.4× and reduces reactive latency by up to 97% compared to state-of-the-art on-device engines. These results demonstrate that careful orchestration of heterogeneous accelerators can unlock fluid, concurrent, and personalized LLM agent experiences fully on-device.

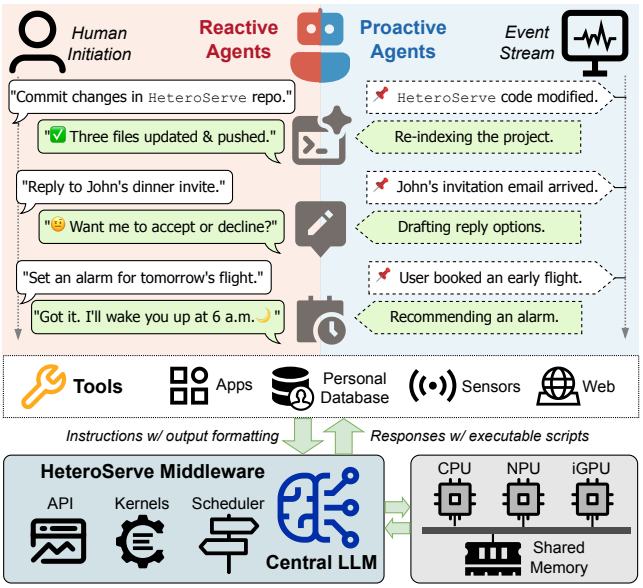

Figure 1. Three-Layer Personal LLM Agent System. Agent.xpu with LLM backbone bridges the agent application layer and hetero-SoC hardware layer, orchestrating both real-time *reactive* queries and best-effort *proactive* tasks.

1 Introduction

LLMs have catalyzed a paradigm shift in intelligent personal assistants [12], enabling systems to autonomously reason, plan, and interact with external tools through agentic frameworks [37, 63, 69]. For example, given a user instruction *"make a daily exercise plan based on my fitness last year"*, the LLM of a mobile agent [35, 62, 64, 74] will retrieve relevant health data and generate personalized suggestions. With rising privacy concerns and latency requirements, local LLM deployment on mobile or desktop devices has gained traction [67, 71, 73]. Agentic applications further underscore on-device LLM since they frequently manipulate personal data and demand prompt response [37]. Beyond that, agents escalate LLM calls compared to traditional chatbots [19, 75] and their wide adoption can overwhelm current cloud-based LLM services. Therefore, local deployment is efficient and secure for agentic LLM workloads.

As the foundation of on-device AI, modern system-on-chip (SoCs) tend to integrate heterogeneous accelerators, including CPU, integrated GPU (iGPU), and neural processing units (NPU). The growing demand for on-device LLM at AI PCs or cellphones has spurred intense competition among hardware vendors, with mainstream products including Intel Core Ultra [27], AMD Ryzen AI [53], Apple Silicon [10], and Qualcomm Snapdragon [30]. Software ecosystems complement SoC infrastructure, exemplified by Apple Intelligence [29] and Microsoft Copilot [45]. Meanwhile, emerging lightweight LLMs (e.g., Llama-3.2-3B [43], Phi-4-mini-3.8B [44]) alleviate inference cost on resource-constrained SoC platforms. Acting as controllers instead of knowledge bases, they can outperform GPT-4 [2, 40] in specific agentic tasks after finetuning [18].

However, the gap between real-world personal assistant and on-device LLM is still widening. On one hand, existing on-device inference optimizations [5, 66, 68] focus on isolated inference tasks, overlooking the dynamic and concurrent nature of agentic LLM calls. On the other hand, recent works on agentic workflow optimization [39, 41, 60] target multi-tenant LLM serving with task semantics, regardless of the distinct workloads of personal LLM agents on personal devices. Before introducing our framework, we identify the typical patterns of agentic workloads, together with the gap between personal agentic applications and heterogeneous SoC (hetero-SoC) systems.

[*]Internal draft; do not redistribute.

Agentic LLM Workloads. As shown in Fig. 1, we summarize two patterns of workload from typical personal agents [12]. ① ***Proactive workload***, where daemon-like agents listen to predetermined event signals and act on them accordingly without human intervention [13, 34, 40]. Proactive agents are ambient, digesting event streams in the background without hard deadlines. ② ***Reactive workload***, on the contrary, is triggered by user conversation and expects timely response. Taking LLM-powered coding as an example, proactive agents monitor code changes, silently performing project parsing, cache construction, or code completion, while reactive agents respond to user prompts on demand, providing explanations, suggestions, or fixes based on the current context. As the central coordinator, the LLM parses natural language queries and crafts outputs that integrate its internal knowledge with tool-interaction commands.

Agent-SoC Gaps. The efficient execution of these mixed workloads is challenged by several fundamental gaps (§2.4), which we confirm and quantify in our hetero-SoC analysis (§3). First, there is a mismatch between the dynamic nature of LLMs and the hardware rigidity of accelerators. NPUs excel at static, pre-compiled computation graphs but struggle with the variable sequence lengths inherent to LLM inference, while more flexible iGPUs suffer from lower energy efficiency and distraction from graphics tasks. Second, constrained memory resources and bandwidth contention on shared-memory SoCs create a critical bottleneck that degrades performance, especially when latency-sensitive and throughput-oriented tasks run concurrently. Third, current hetero-SoC runtimes provide deficient abstractions for agentic workloads, lacking native support for the fine-grained preemption, priority scheduling, and dynamic batching necessary to efficiently co-locate reactive and proactive tasks.

Agent.xpu Framework. To bridge these gaps, we propose Agent.xpu to efficiently schedule agentic LLM workloads on hetero-SoCs. It is also the first on-device LLM engine for concurrent inference serving. As shown in Fig. 1, Agent.xpu is designed from the ground up to understand and orchestrate the distinct proactive and reactive tasks, mapping them intelligently onto the underlying hardware to balance latency, throughput, and energy efficiency. It addresses the identified gaps through a combination of offline, profiling-driven compilation and online, adaptive scheduling. We implement HeteroServe on Intel Core Ultra SoCs [27] using OpenVINO [47] APIs, with a lightweight C++ serving engine. Evaluation with Llama-3B/8B models under diverse benchmarks shows that HeteroServe improves proactive throughput by up to 2.4× and reduces reactive latency by up to 97%, while lowering iGPU utilization by 37.1% and energy per token by 26.8%, compared with industrial on-device serving frameworks.

- We systematically analyze the unique characteristics of agentic LLM workloads and perform an in-depth empirical study (§3) that quantifies operator-accelerator affinity, memory contention, batching effects, and proactive-reactive interference on modern hetero-SoC.
- We propose heterogeneous execution graph (HEG) (§4.2), a hetero-centric compute abstraction for elastic XPU mapping. It enables a principled, heterogeneous disaggregation of prefill and decode stages across the NPU and iGPU to leverage accelerator strengths and mitigate interference.
- We design an online scheduler (§4) that combines fine-grained, kernel-level preemption for reactive responsiveness with a slack-aware piggybacking mechanism for proactive work conserving. Its built-in XPU coordinator fulfills adaptive kernel dispatch to avoid bandwidth contention, reduce pipeline bubbles, and maximize system throughput.
- We present a full-system implementation of Agent.xpu (§5) and evaluate it on a commercial SoC with popular on-device LLMs (§6), Our hetero-driven approach significantly outperforms existing on-device LLM engines in single- and mixed-workload scenarios.

2 Background and Challenges

We first provide the preliminaries, including an overview of personal LLM agents (§2.1), the basics of LLM inference (§2.2), and the landscape of hardware/software for on-device LLM (§2.3). Then we describe the gaps between hetero-SoC and agent serving (§2.4), which motivate Agent.xpu design.

2.1 Personal LLM Agents

In this work, we focus on *personal* LLM agents deeply coupled with personal data, personal devices, and personal applications [37]. Analogous to the kernel in a traditional OS, the foundational LLM serves as the core execution engine of a personal LLM agent system, handling diverse queries issued by both reactive and proactive agents, each with distinct throughput and latency requirements. Reactive agents are instantiated in direct response to explicit human requests (such as question answering, event scheduling, process automation, etc.), where responsiveness is paramount. In contrast, proactive agents operate in a "human-in-the-loop" [46] manner, initiating actions autonomously based on contextual cues from the user's environment, yet seeking human feedback or approval before execution. This duality introduces heterogeneous temporal characteristics into the workload: reactive requests demand low-latency turnaround, while proactive tasks may tolerate higher latency but benefit from opportunistic batching and background scheduling. LLM agent orchestration frameworks, such as LangChain [8], LlamaIndex [9], and AutoGen [61], are equipped to customize both proactive and reactive agentic workflows.

2.2 LLM Inference Primer

LLM inference is typically based on decoder-only Transformers, comprising a ***prefill*** stage, which encodes the prompt and generates the first token, and an auto-regressive ***decode*** stage to produce subsequent tokens one by one. Intermediate states (known as ***KV cache*** [51]) are stored and updated after each step. The end-to-end latency of LLM inference consists of ***time to first token (TTFT)***, i.e., the prefill phase, and ***time per output token (TPOT)*** multiplied by the number of decoded tokens after the first. In both prefill and decode, each Transformer layer constitutes three major blocks: ***1) QKV projection***, ***2) multi-head attention (MHA)***, and ***3) feed-forward network (FFN)***. QKV projection and FFN operates independently on each *token*, while attention attends over the entire *sequence*. This distinction underlines differences in data dependencies and parallelism between the two types of operations.

Cloud-based LLM inference is dedicated to serve user queries at high throughput while meeting serving-level objectives (SLOs). Established techniques include kernel optimization [11, 65], continuous batching [33, 72] with chunked prefill [3], prefill-decode disaggregation [49, 78], KV cache reuse [1, 58, 77] or defragmentation [33], and improved tensor or pipeline parallelism [38, 56]. For resource-constrained conditions where GPU memory is limited, existing solutions propose CPU memory/disk offloading by leveraging weight locality [57] or I/O-aware scheduling [55].

2.3 On-Device LLM

Heterogeneous SoC. Heterogeneous SoCs have become a cornerstone for efficient LLM deployment on personal devices, spanning mobile, laptop, and edge platforms. As illustrated in Fig. 12, these SoCs exhibit a unified memory architecture distinct from heterogeneous systems with discrete accelerators: the CPU, iGPU, and NPU share the same physical system memory, eliminating costly host–device data transfers. The iGPU adopts a SIMT execution model akin to discrete GPUs, while the NPU is purpose-built for specific tensor operations, offering similar levels of parallelism but with significantly higher energy efficiency. By integrating these components on a single die, heterogeneous SoCs reduce latency and power overheads, while enabling fine-grained, workload-specific optimizations.

On-Device Inference Engines. Driven by increasing demands for privacy, responsiveness, and power efficiency, a range of industrial on-device LLM inference engines have emerged, including Llama.cpp [21], OpenVINO [47], ONNX Runtime [16], IPEX-LLM [7], MNN [42], QNN [17], Core ML [14], LiteRT [15], and MLLM [66, 70]. These frameworks facilitate out-of-the-box LLM deployment on vendor-specific CPUs, GPUs, or NPUs. Table 1 summarizes mainstream frameworks in comparison to Agent.xpu. Agent.xpu implements W8A16 quantization on both NPUs and GPUs to balance inference efficiency and accuracy precision, and supports NPU–GPU hybrid inference via disaggregated prefill–decode execution. In contrast, Llama.cpp [21] enables layer-wise CPU–GPU pipelining without temporal overlapping. Llama.cpp and OpenVINO [47] supports server-client-based LLM serving with continuous batching, which is inefficient on a single accelerator (§3.2). Except for Agent.xpu, none support preemptive scheduling, which is the prerequisite for serving requests with differing priorities. Recent research has explored GPU–NPU (or CPU–NPU) co-execution via activation outlier isolation [66, 68], as well as tensor parallelism through activation or weight partitioning [5]. However, these techniques primarily target reducing the end-to-end latency of monolithic inferences, whereas future on-device agentic applications demand broader support for concurrency, state dependency, and real-time interactivity.

Table 1. Comparison of On-Device LLM Engines.

Framework	GPU	NPU	Hetero. Execution	Model Serving	Preempt. Support
Llama.cpp [21]	W4A16	/	CPU-GPU	Cont. Batching	/
ONNX Runtime [16]	FP16/FP32	INT8	/	N/A (Offline)	/
IPEX-LLM [7]	W8A16	INT4	/	N/A (Offline)	/
MNN [42]	W4A16	/	/	N/A (Offline)	/
MLLM [66, 70]	/	INT4	/	N/A (Offline)	/
Qualcomm AI [52]	W4A16	INT4/8	/	N/A (Offline)	/
OpenVINO [47]	W8A16	INT4	/	Cont. Batching	/
Agent.xpu	W8A16	W8A16	NPU-GPU	Hetero. PD-Disaggregation	Sub-100 ms Preemption

The listed GPU/NPU quantization methods represent the most common configurations for each framework; CPU is omitted as typically supported by default. Both W8A16 and INT8 store weights in INT8, but use FP16 and INT activation/arithmetic, respectively; similarly for W4A16 and INT4.

2.4 Gaps between Hetero-SoC and Agent Serving

Mismatch between LLM dynamism and hardware rigidity. Modern LLMs process arbitrarily sized user inputs via dynamic-shape operators along the sequence dimension, whereas mainstream NPUs are optimized for static-shape NN operations. NPUs depend on costly compilation of fixed computational graphs to pre-allocate resources and optimize dataflows (e.g., tiling onto fixed-size MAC arrays), making cold-start compilation at runtime infeasible. Programmable iGPUs can execute dynamic kernels but at lower energy efficiency and with contention against graphics tasks (e.g., UI composition, video playback), rendering full-iGPU execution impractical on personal devices. As analyzed in §3.1, NPU–iGPU collaboration is the most viable approach, and Agent.xpu further overcomes its workload-partitioning and interference challenges.

Constrained memory and bandwidth contention. Personal SoCs face tight memory capacity and bandwidth limits.

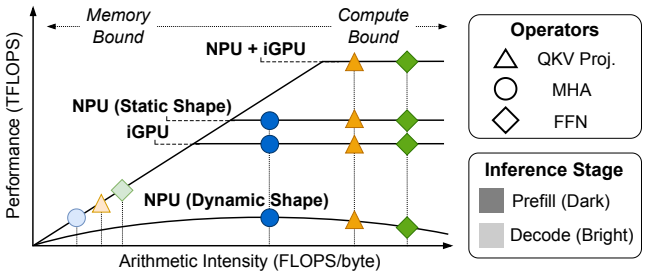

Figure 2. Schematic Roofline Illustration of LLM Ops.

NPUs and iGPUs have small on-chip SRAM and depend heavily on shared, bandwidth-limited DRAM, constraining kernel size, context length, and batch dimensions critical to serving throughput. Concurrent DDR access by NPU and iGPU causes *contention* that slows parallel requests, as illustrated in our contention analysis in §3.1. For mixed agentic workloads, memory-bound proactive tasks in the decode stage can saturate DDR bandwidth, delaying latency-sensitive reactive requests on any accelerator. Agent.xpu leverages memory-efficient, bandwidth-aware scheduling to alleviate memory pressure and maintain system responsiveness.

Deficient runtime abstractions for agentic workloads. Hetero-SoC runtimes, designed for stateless offline workloads, lack key abstractions for agentic LLM serving. First, they often omit dynamic batching across requests, reducing hardware utilization. Second, they lack mechanisms for fine-grained preemption and priority scheduling, preventing isolation of high-priority tasks from background inferences. Third, with minimal hardware-level control, software schedulers have limited visibility into accelerator states (e.g., NPU buffer occupancy), making dynamic load balancing across accelerators difficult. Our task-level analysis (§3.2) motivates Agent.xpu runtime extensions to fully unlock hetero-SoC potential for personal agents.

3 Hetero-SoC Analysis and Opportunities

We conduct a comprehensive hetero-SoC analysis to guide the design of Agent.xpu. Operator-level analysis (§3.1) characterizes the compute and memory demands of representative LLM operators (ops), while task-level analysis (§3.2) examines runtime behavior in end-to-end agentic LLM calls. Profiling experiments are conducted on an Intel Core Ultra processor [27] with DDR5 DRAM.

3.1 Operator-Level Analysis

Operator-Accelerator Affinity. To illustrate the affinity between LLM ops and hetero-SoC accelerators, we construct a schematic roofline model (Fig. 2) derived from our profiling results of Llama [43] 3B/8B models[1]. Static-shaped NPU kernels can be pre-compiled, whereas dynamic-shaped kernels require costly JIT compilation. For dynamic NPU prefill kernels, we amortize this cost by dividing compilation time by the number of LLM layers, as kernels are reused across layers. However, the long compilation latency makes on-demand compilation at runtime impractical. Static-shaped kernels, by contrast, offer superior energy efficiency (TFLOPS/Watt) and comparable throughput against iGPUs. ***Opportunity:*** Precompile chunked NPU kernels for token-wise prefill ops, while offloading dynamic prefill and all decode ops to iGPU, which better handles growing dimensions and avoids NPU underutilization. At runtime, warm up NPU MHA kernels for queued requests on-the-fly to hide compilation latency and opportunistically shift prefill from iGPU.

[1]Simplifications include: arithmetic intensity of each operator varies with prompt length (here is a moderate 512 tokens); roofline curves of different ops may actually diverge and are merged here; relative NPU/iGPU peak performance are vendor-dependent; memory bandwidth slope for each curve differs by factors such as L2 or scratchpad memory size; and NPU+iGPU parallelism can slightly boost bandwidth when memory-bound (Fig. 3, [5]).

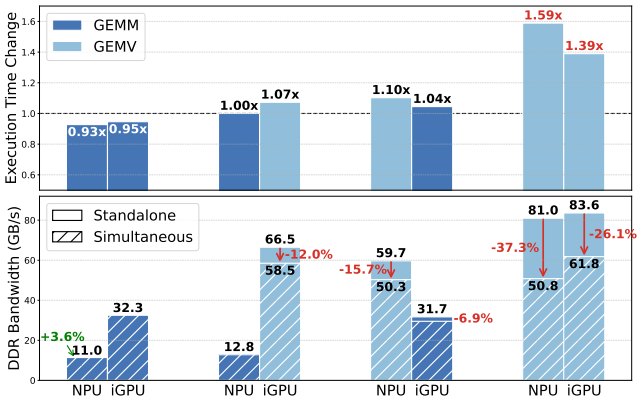

Figure 3. Memory Contention Analysis. Changes of execution time (upper) and DDR bandwidth (lower) from standalone NPU/iGPU kernel running to simultaneous co-execution. Memory-bound GEMV kernels are more sensitive to NPU/iGPU parallelism than compute-bound GEMM.

Memory Access Pattern and Contention. Discrete GPUs with dedicated VRAM exhibit *bulk transfer and decoupled compute*, moving data between host DRAM and VRAM before or after execution. In contrast, we observe that NPUs and iGPUs with limited SRAM and DMA capacity adopt *streaming access and coupled compute*, consuming data progressively during execution, which underscores NPU-iGPU contention management under heavy DDR traffic. As shown in Fig. 3, we measure latency and bandwidth changes between separate and simultaneous GEMM/GEMV executions with Intel VTune [31]. GEMM and GEMV dominate prefill and decode stages, respectively. Co-executing memory-bound GEMV degrades both latency and bandwidth, whereas compute-bound GEMM are largely unaffected. ***Opportunity:*** Contention can be largely mitigated through meticulous dispatching of prefill and decode kernels to NPU/iGPU (§3.2), complemented by adaptive kernel reordering or deferral (§4.3).

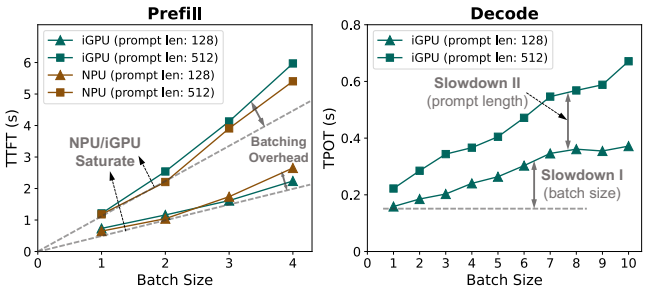

Figure 4. Individual Task Latency in Batching. Distinctive batching effects of prefill and decode on Llama-3B model.

3.2 Task-Level Analysis

Batching Effects on Hetero-SoC. Batching intuitively improves throughput, while the individual latency is more sensitive to batch size on resource-constrained SoCs. As shown in Fig. 4, we analyze prefill and decode batching across varying batch sizes, prompt lengths, and accelerators[2]. Prefill latency scales nearly linearly with batch size, saturating the NPU or iGPU without performance gain. Decode latency rises gradually with larger batches or longer prompts, since each task's MHA operates on its own KV cache with unchanged arithmetic intensity, limiting batching efficiency. This effect is amplified by constrained memory bandwidth and quadratic MHA complexity in sequence length. Furthermore, the latency gap between prefill and decode reflects decode degradation when batched with prefill. ***Opportunity:*** To trade-off latency and throughput in batching, we can disaggregate prefill and decode across NPU and iGPU (§4.3), and employ adaptive decode batching tailored to task priorities (§4.5).

Proactive-Reactive Interference. Agentic LLM serving interleaves proactive tasks with latency-critical reactive tasks, creating conflicting throughput–latency demands. Fig. 5 compares four co-scheduling schemes. Single-accelerator approaches, including (a) instant preemption without context saving, (b) time-sharing via multi-stream or virtualization, and (c) continuous batching [33, 72], all suffer inefficiencies: (a) recomputation and idle time, (b) slowdowns and duplicated intermediate buffers, and (c) prefill–decode interference, which lengthens decode and is exacerbated on resource-constrained SoCs with longer prefill times. By co-locating heterogeneous stages on the same accelerator with coupled resource allocation, these methods cannot simultaneously satisfy the objectives of agentic workloads. Our hetero-SoC scheme (d) partitions prefill to the NPU and decode to the iGPU, with the iGPU also handling reactive prefill and dynamic prefill kernels. In this way, we achieve lowest reactive task latency and highest overall throughput. ***Opportunity:*** Achieve both responsiveness and system throughput via efficient preemption for reactive tasks, plus recomputation-free resumption (4.4) and slack-aware piggybacking (§4.5) guided by batching and contention profiles for proactive tasks.

[2]NPU decode is omitted since per-iteration kernel compilation is infeasible under growing sequence length and varying batch size.

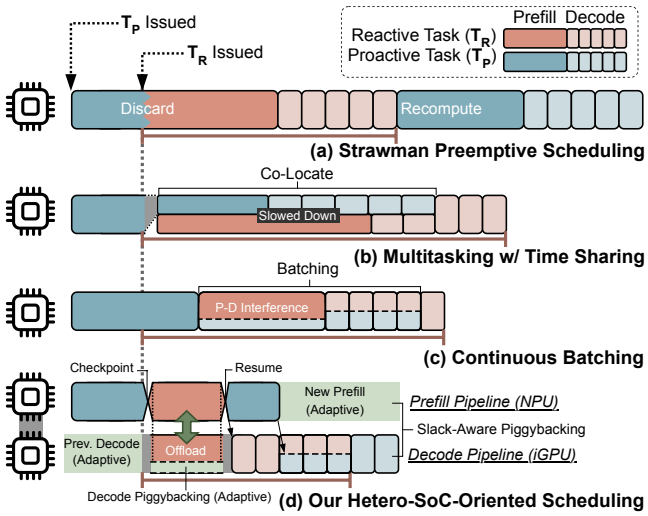

Figure 5. Proactive-Reactive Co-Scheduling. (a)(b)(c) target single accelerator, while (d) uses NPU and iGPU primarily for prefill an decode, respectively.

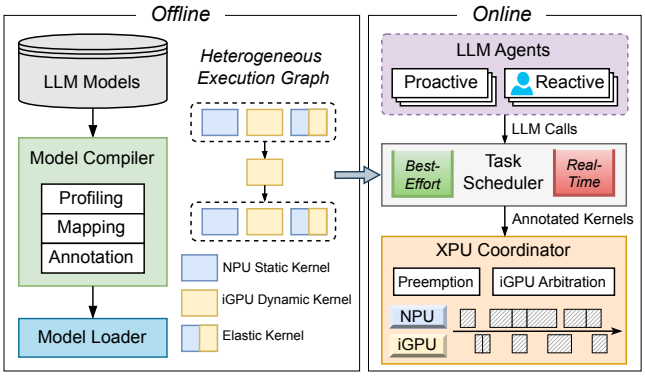

Figure 6. Agent.xpu System Design.

4 Agent.xpu Design

4.1 Contextualization and System Overview

Role of Agent.xpu. Agent.xpu serves concurrent LLM calls from on-device personal agents, rather than handling standalone LLM inferences. Operating in a non-clairvoyant manner, Agent.xpu does not rely on knowledge of the agentic workflow or task arrival times. It is informed only of task priorities (proactive or reactive) at the time of issuance. Agent.xpu is designed with the following primary ***objectives***: 1) prioritizing the reduction of end-to-end latency for LLM requests originating from reactive agents to enhance user

experience, 2) increasing the overall throughput for background LLM calls from proactive agents, and 3) optimizing compute and memory resource utilization in hetero-SoC to achieve improved performance and energy efficiency.

Workload Characteristics. On resource-limited hetero-SoCs, we anticipate a typical LLM request rate from personal agents of 1 to 30 requests per minute. Calls from proactive and reactive agents are assumed to be independently distributed. LLM inferences are separate from other agentic sub-tasks, such as human interaction or tool engagement, which are assumed to be primarily CPU- or I/O-bound, avoiding NPU and iGPU resource competence. Within Agent.xpu, iGPU usage is intentionally limited to ensure graphics availability and power efficiency.

System Design. As depicted in Fig. 6, the system comprises offline and online components. 1) ***Offline***: it maintains model weights and a heterogeneous execution graph (HEG; §4.2) with precompiled NPU kernels and JIT iGPU kernels. Each HEG node carries profiling-guided annotations (latency and bandwidth hints, as a function of batch/sequence). An elastic-kernel abstraction enables late binding of each operator to NPU or iGPU at dispatch time. 2) ***Online***: Agent.xpu dynamically schedules agentic LLM calls across the SoC via the following modules:

① *Request Manager.* This module interfaces with the agent frontend to asynchronously admit LLM calls. It maintains a global request table, where each entry records the request UUID, lifecycle, KV cache allocation, prompt tokens, and inference progress (phase, layer, current kernel, generated tokens). This fine-grained tracking enables task checkpoint and resume without recomputation. The manager handles lightweight admission and moves requests across task queues.

② *Dual Task Queues.* For both prefill and decode, Agent.xpu maintains a real-time queue (reactive) and a best-effort queue (proactive). The queues feed the corresponding event loops, enabling efficient scheduling and immediate preemption.

③ *Prefill/Decode Event Loops.* Two dedicated loops busy-poll their queues for low-latency dispatch. They decompose tasks into NPU, iGPU, or elastic kernels and submit them to the XPU coordinator. Compute-bound prefill is run singly for each request, while decode loop adopts in-flight request batching (§4.3). The prefill loop supports sub-100 ms preemption for reactive requests at kernel boundaries (§4.4), while the decode loop dynamically adjusts batch size based on request priority and latency budgets (§4.5).

④ *XPU Coordinator.* This module orchestrates concurrent execution across NPU, iGPU, and related CPU activities (e.g., NPU kernel compilation). It tracks per-accelerator occupancy, memory residency, and contention signals, binding elastic kernels to accelerators and processing kernel-level preemption based on HEG annotations, task priority, and current load. The coordinator also arbitrates simultaneous iGPU requests under a prefill-prioritized policy (§4.3: ③).

⑤ *Recurrent Activation Double Buffer.* Agent.xpu maintain single-layer activation buffer, which can be reused recurrently through layers. Prefill and decode buffers are sized by max context length and maximum batch size, respectively, with the same hidden dimension. Agent.xpu adopts reactive-proactive double buffer to enable copy-free context switching for preemption (§4.4).

⑥ *Memory Manager.* To fully utilize on-device memory, Agent.xpu employs a background garbage collector that reclaims KV caches and on-demand NPU kernels once completed. We assume moderate request density typical of personal agents without memory overflow. Should out-of-memory condition arises, application-directed tiering is prefered over blind paging: selectively offloading cold KV cache or weight shards to flash storage. Such offloading policies are orthogonal to our core design and can be seamlessly integrated [55].

4.2 Heterogeneous Execution Graph

Graph Representation. Agent.xpu models LLM execution as a heterogeneous execution graph (HEG): a parametric, accelerator-aware multigraph where nodes denote operator variants and edges capture dataflow, memory liveness, and accelerator transfers. The HEG consists of a one-shot prefill DAG and a recurrent decode micro-DAG invoked per token. Token-wise QKV projection and FFN can be chunked during prefill or batched during decode with shared weights, while sequence-wise MHA operates per request over individual KV cache.

Operator Mapping. To map operators onto specific accelerators while enabling runtime elasticity, we balance computational efficiency, data locality, and resource utilization:

- *Computation Affinity.* Ops are placed based on roofline characteristics (§3.1) and micro-architectural fit. For prefill, QKV projection and FFN use chunked NPU kernels or dynamic iGPU kernels; chunk sizes are tuned to saturate the NPU. Prefill MHA defaults to iGPU, though on-demand NPU warm-up can be employed. During decode, all kernels run on iGPU due to dynamic batch sizes, growing sequence lengths, and request heterogeneity. The CPU is reserved for control flow, toking sampling, and kernel compilation.
- *Runtime Elasticity.* Prefill ops along the sequence dimension can be partitioned into chunked and dynamic parts. The coordinator adaptively chooses NPU or iGPU for each, depending on request priority, accelerator load, and bandwidth pressure. On-demand NPU kernels, if compiled in time, may also replace iGPU execution for dynamic ops.
- *Memory Awareness.* To minimize costly DDR transfers, we fuse consecutive ops into three core kernels, exploiting NPU/iGPU SRAM with data locality. Unlike prior approaches [5, 66] that split linear and nonlinear ops across accelerators, our fused kernels exploit modern NPUs' nonlinear units to avoid cross-accelerator transfer. For decode, we similarly fuse ops into three iGPU kernels, and further

optimize by collapsing an entire decode layer into a single iGPU kernel for single-batch decode iteration.

Predictive Kernel Annotation. Combining our roofline profiling (§3.1) and analytical model (detailed in §B), we can predict kernel latency and bandwidth utilization as a function of sequence length or batch size, given accelerator choice. LLM ops are idempotent with fixed FLOPs, and we find that kernel execution times on NPU/iGPU are stable across invocations. This allows the scheduler to estimate TTFT for individual tasks and TPOT for batched decode, enabling adaptive scheduling via kernel reordering or deferral.

4.3 Disaggregated Prefill/Decode Pipelines

Agent.xpu decouples prefill and decode into specialized scheduling pipelines, exploiting the asymmetry between one-shot prompt ingestion and iterative token generation. Prefill primarily leverages the NPU for compute-intensive token-wise ops, while decode resides on the iGPU to support dynamic batching and sequence growth.

① **Prefill Pipeline.** For reactive requests, we minimize TTFT by partitioning token-wise ops across NPU and iGPU with elastic tensor parallelism (details in ④). For *proactive requests*, token-wise ops run mainly on NPU, while dynamic ops (e.g., MHA) and residual fragments in chunked kernels run on iGPU. Because a single request already saturates either accelerator, prefill avoids batching and processes requests serially. This pipeline also serves as the substrate for the preemption mechanism of reactive requests (§4.4).

② **Decode Pipeline.** The decode pipeline adopts continuous batching decoupled from prefill, scheduling requests at iteration granularity. Reactive and proactive requests can be co-batched under adaptive strategies (§4.5). Executing all decode kernels on the iGPU naturally accommodates dynamic sequence lengths and fluctuating batch composition.

③ **Pipeline Coordination and iGPU Arbitration.** On shared memory SoCs, prefill and decode share the in-place KV cache without costly cross-accelerator transfers. However, both stages may contend for the iGPU due to common dynamic operators, necessitating arbitration mechanisms. Agent.xpu adopts a *prefill-first* arbitration policy: iGPU kernels from prefill always take precedence over decode, regardless of request type. This design is motivated by: 1) decode is memory-bound and generally longer than prefill, and 2) prefill requires only a small fraction of iGPU kernels, with the bulk of computation offloaded to the NPU. Prioritizing prefill ensures that short bursts of iGPU work complete promptly, avoiding long stalls that would otherwise block the entire prefill pipeline and inflate TTFT. Decode jobs can then efficiently utilize the wide gaps between prefill bursts.

④ **Elastic NPU-iGPU Tensor Parallelism.** To reduce TTFT for reactive requests while mitigating interference with ongoing decode, Agent.xpu elastically partitions reactive prefill kernels across NPU and iGPU at runtime. Decisions are made *layer-wise* by the XPU coordinator:

1. *Decode idle:* Prefill fully exploits both NPU and iGPU, with chunked operators split so that the two accelerators complete near-simultaneously, eliminating pipeline bubbles.
2. *Decode busy:* Despite uncertain decode completion, the coordinator conservatively reduces iGPU usage by assigning fewer chunked kernels, preserving iGPU slot for decode. Prefill iGPU kernels are deferred to complete no earlier than their parallel NPU counterparts, increasing the chance that decode finishes mid-layer with contention alleviated.
3. *Fallback:* If decode becomes idle again, prefill reverts to full NPU+iGPU parallel execution.

This elastic scheme balances reactive TTFT and decode throughput (TPOT), exploiting profiling-guided kernel annotations to align execution across heterogeneous accelerators.

⑤ **On-the-fly NPU Kernel Warm-up.** At runtime, Agent.xpu opportunistically prepares dynamic NPU kernels (e.g., MHA) once a request is queued or preempted, thereby hiding compilation latency and reducing iGPU prefill load. Since prompt length is known at enqueue time, the CPU can start compiling static NPU kernels immediately; if compilation completes before prefill begins, the request switches to pure-NPU prefill, mitigating iGPU interference with decode. Compiled kernels are reclaimed once unused or expired. To eliminate potential contention introduced by NPU-iGPU co-execution, when NPU prefill kernels become memory-bound (e.g., MHA with short prompt length) and overlap with memory-intensive iGPU decode, the coordinator prioritizes reactive tasks: if both pipelines share the same priority (all-reactive or all-proactive), they proceed concurrently; otherwise, work with lower priority is deferred until the higher-priority side completes for reactive latency preservation.

4.4 Fine-Grained Preemption of Reactive Tasks

Fast preemption of reactive tasks is critical for reducing TTFT. However, commodity NPUs and iGPUs on hetero-SoCs lack native preemption support. Prior work [24] proposed reset-based preemption on discrete GPUs, which kills all queued kernels and later restores context, avoiding long waits but incurring additional overhead. In contrast, our setting benefits from shared memory and the double activation buffer design, which makes context switching lightweight. Combined with fine-grained, short-lived prefill kernels and moderate reactive request frequency, we adopt wait-based preemption at kernel or layer boundaries, instead of reset-based termination that is costly on resource-constrained SoCs.

① **Copy-Free Context Switching.** An inference context comprises the KV cache, progress metadata (stage, current layer, finished kernels, generated tokens), and intermediate activations. The shared-memory architecture eliminates the need to swap KV cache when a reactive request arrives. Our

double activation buffer separates proactive and reactive activations of prefill, enabling low-latency preemption and instant restoration without data copying.

② **Kernel-Level Prefill Preemption.** During prefill, we employ wait-then-execute preemption at kernel boundaries. Each kernel is registered with a lightweight completion callback. Upon finishing, the callback efficiently probes the reactive queue; if non-empty, it signals the prefill event loop to immediately dequeue and decompose the reactive job into kernels, thereby preempting the current proactive task. Reactive tasks do not preempt each other and run sequentially like proactive ones. Because token-wise ops are chunked, kernel execution time is tightly bounded. For an 8B model, chunked kernels (size 256) complete within 38 ms on NPU/iGPU, while MHA with sequence length 2048 completes within 97 ms. This ensures sub-100 ms preemption latency—nearly imperceptible to users with negligible TTFT impact.

③ **Iteration-Level Decode Preemption.** For decode, reactive requests wait until the current iteration completes, then immediately join the next batch. Proactive jobs may be evicted under adaptive batching policies (§4.5) to preserve reactive latency. Since iteration-level preemption is bounded by TPOT and the only per-request context is the localized KV cache and generated tokens, it avoids expensive swapping while still guaranteeing fast response.

4.5 Slack-Aware Piggybacking of Proactive Tasks

To preserve reactive responsiveness while preventing starvation of proactive requests, Agent.xpu employs *slack-aware piggybacking*, where proactive work opportunistically fills pipeline slack in prefill and decode with negligible impact on reactive task latency.

① **Slack Identification.** The scheduler detects two forms of slack during reactive–proactive coexistence: 1) *Compute slack*, arising from idle accelerator resources. Proactive prefill can overlap with the decode pipeline whenever no reactive prefill is pending. 2) *Bandwidth slack*, during memory-bound decode stage. Proactive requests can be co-batched with reactive ones under elaborate batching rules that bound the additional latency seen by reactive tasks, as discussed below.

② **Adaptive Decode Batching.** Profiling reveals that per-task decode latency increases with both batch size and request sequence length. Agent.xpu employs a *reactive-first* batching strategy, admitting proactive tasks only when their marginal impact on reactive latency is accepted. Each decode iteration begins by consolidating previously surviving requests with new arrivals, categorizing them into reactive and proactive groups. In homogeneous batches that are entirely reactive or proactive, all requests are included. For heterogeneous batches exceeding the predetermined maximum size B_{rf} (set to 3 in this work), proactive requests are selectively evicted to meet size constraints, prioritizing the 1) removal of longer bottleneck sequences, and 2) newly arrived tasks, to reduce TPOT and minimize system churn.

③ **Starvation Prevention.** While reactive requests receive priority, the scheduler implements aging mechanisms to prevent indefinite postponement of proactive requests. Proactive requests that exceed a threshold age are promoted to prevent starvation. They obtain two privileges without suspending reactive execution: 1) the scheduler reallocates iGPU to manage the overdue proactive prefill, while retaining reactive prefill (except MHA) solely on NPU, and 2) after prefill, the aged request can immediately join the decode batch, bypassing the batch size cap for a reactive-first batch.

5 Implementation

Agent.xpu is implemented as an on-device LLM serving system for personal agents, exposing a RESTful API frontend (2.3K lines of Python code) for prioritized query submission in a server-client manner. The serving engine is written in 6K lines of C++ with custom abstractions for tensors, compute graphs, and model loading, eschewing third-party dependencies (e.g., PyTorch [48], GGML [20]). This lightweight design allows Agent.xpu to natively embed the scheduling mechanisms introduced in §4, while keeping kernel interfaces modular to accommodate diverse SoC platforms.

Our prototype targets Intel Core Ultra SoCs [27], implementing NPU and iGPU kernels using the low-level APIs of OpenVINO 2025.2 [47]. Request- and kernel-level scheduling is realized through a two-tier asynchronous interface: 1) *Inter-op parallelism* leverages prefill and decode event loops based on `std::thread`, coordinated via thread-safe task queues with mutexes, condition variables, and atomics; (2) *Intra-op parallelism* exploits elastic NPU-iGPU tensor partitioning, orchestrated by the XPU coordinator using hardware-specific coroutines (`start_async/wait` in OpenVINO). The offline HEG and online scheduler jointly enable seamless adaptation to different LLM models, agent workflows, and SoC targets, ensuring high-performance execution with minimal software overhead.

6 Evaluation

We evaluate Agent.xpu on proactive-only and mixed agentic workloads using Llama-3B/8B models. Overall, Agent.xpu improves proactive throughput by up to 2.4× over iGPU baseline (§6.2), while reducing reactive latency by up to 97% under load (§6.3). Detailed breakdowns further attribute these gains to efficient heterogeneous co-scheduling (§6.4).

6.1 Experimental Setup

Hetero-SoC Testbed and Models. We deploy Agent.xpu on an ASUS NUC 14 Pro+ mini-PC equipped with an Intel Core Ultra 5 125H processor [27] and 64GB DDR5 DRAM, running Ubuntu 24.04. The processor integrates an Intel Arc iGPU and Intel AI Boost NPU. For evaluation, we use

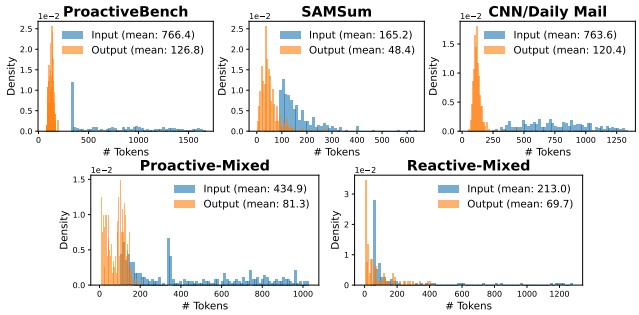

Figure 7. Input/output length distributions of three proactive datasets and two proactive- or reactive-mixed datasets.

Llama-3.1-8B-Instruct and Llama-3.2-3B-Instruct [43], which cover the most common on-device LLM sizes. All models are quantized with W8A16 round-to-nearest quantization, incurring negligible accuracy loss [28].

Agentic Workloads. To approximate realistic personal agent behavior, we construct both proactive and reactive workloads: *Proactive:* 1) ProactiveBench [40] with real user events such as keyboard, clipboard, and browser activity; 2) SAMSum [22], modeling group chat reply drafting; and 3) CNN/-DailyMail [26], for news summarization. *Reactive:* 1) LMSys-chat-1M [76], covering diverse one-on-one conversations; 2) MTRAG [32], a multi-turn retrieval-augmented generation benchmark; and 3) Berkeley Function Call Leaderboard [50], which produces structured API calls.

We evaluate two regimes: 1) *Proactive-only:* Each proactive benchmark is run individually. 2) *Mixed scheduling:* Proactive and reactive requests are co-scheduled. To synthesize mixed workloads, we uniformly sample from the three proactive and three reactive benchmarks to form proactive-mixed and reactive-mixed datasets. As shown in Fig. 7, the input/output length distribution of each dataset differs substantially, which improves evaluation comprehensiveness. The reactive workloads follow a long-tail distribution, stressing latency guarantees under unpredictable request lengths.

Since the original datasets lack timestamps, we synthesize the arrival times using Poisson process with request rates varying between 1–30 requests per minute (req/min), matching observed densities for personal agents. Proactive and reactive arrivals are generated independently. For each dataset, we evaluate the baselines and Agent.xpu with 15-minute traces, corresponding to increasing request rates.

Baselines. Only a few on-device frameworks support LLM serving with a server-client interface. We compare with state-of-the-art industrial engines with such support and optimizations for the underlying hetero-SoC platform:

- Llama.cpp [21], a versatile on-device inference engine optimized for multi-core CPUs. We use its serving mode with continuous batching.
- OpenVINO [47], Intel's deployment stack for CPUs, iGPUs, and NPUs. Since Agent.xpu also builds on OpenVINO's low-level APIs, single-batch inference performance is almost aligned. For serving, OpenVINO enables continuous batching on iGPUs, but NPU execution remains serial due to limited batching support. Moreover, NPU decoding is restricted to INT4, which is slower than iGPU's W8A16 kernels given padding and masking overheads. To ensure fairness, we re-implement the NPU baseline with serial execution but use iGPU kernels for decode and MHA.

Metrics. We measure both performance and efficiency under varying proactive and reactive request densities. The metrics include: 1) *Normalized Latency*, calculated as the average request end-to-end latency divided by combined input and output lengths, gauging throughput under high request rates. We also measure P90 latency of reactive requests to highlight user-facing responsiveness; 2) *iGPU Utilization*, derived by measuring iGPU utilization during prefill and decode phases, summed across the timeline of each stage; 3) *Energy per Token*, measured as the energy consumed (J/token), normalized by the processed token count.

6.2 End-to-End Performance

Proactive-Only Workloads. We first examine proactive-only workloads, where throughput is the primary metric. In this regime, Agent.xpu assigns uniform priority without preemption or adaptive batching. Although designed for mixed workloads, Agent.xpu already outperforms single-accelerator baselines in throughput (Fig. 8). OpenVINO (iGPU) ranks second, benefitting from throughput-oriented continuous batching. CPU excels in decode but is bottlenecked by slow prefill, while NPU is limited by serial execution. Agent.xpu shares low-level kernels with OpenVINO with similar single-batch inference speed, but it achieves lower iGPU utilization (§ 6.4) by leveraging NPU-iGPU co-execution.

Performance gains are amplified at higher request rates. On SAMSum, where prompts and outputs are relatively short, all systems sustain higher rates before saturation. With Llama-3B, Agent.xpu achieves 2.0×–2.4× throughput over OpenVINO (iGPU), and delivers ~20% more requests at 1.0 s/-token latency on ProactiveBench and CNN/DailyMail. Gains are even larger with Llama-8B, as heavier load amplifies the benefit of heterogeneous scheduling. Compared with Llama.cpp (CPU), Agent.xpu achieves up to 3.9× throughput. These results highlight both the scalability and efficiency of our approach.

Reactive-Proactive Mixed Workloads. We next evaluate mixed workloads, where throughput and reactive latency must be balanced. Baselines lack request prioritization, so they treat proactive and reactive inputs equally. In contrast, Agent.xpu enforces reactive-first scheduling with adaptive batching. Figure 9 reports mean and P90 latencies to capture both responsiveness and tail behavior.

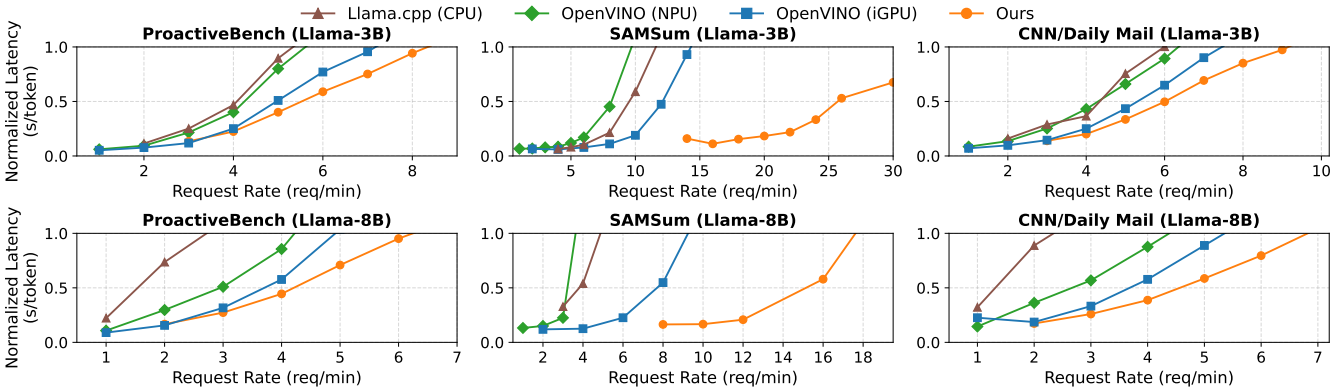

Figure 8. Proactive-Only Workloads. End-to-end results with Llama-3B/8B models on three proactive datasets.

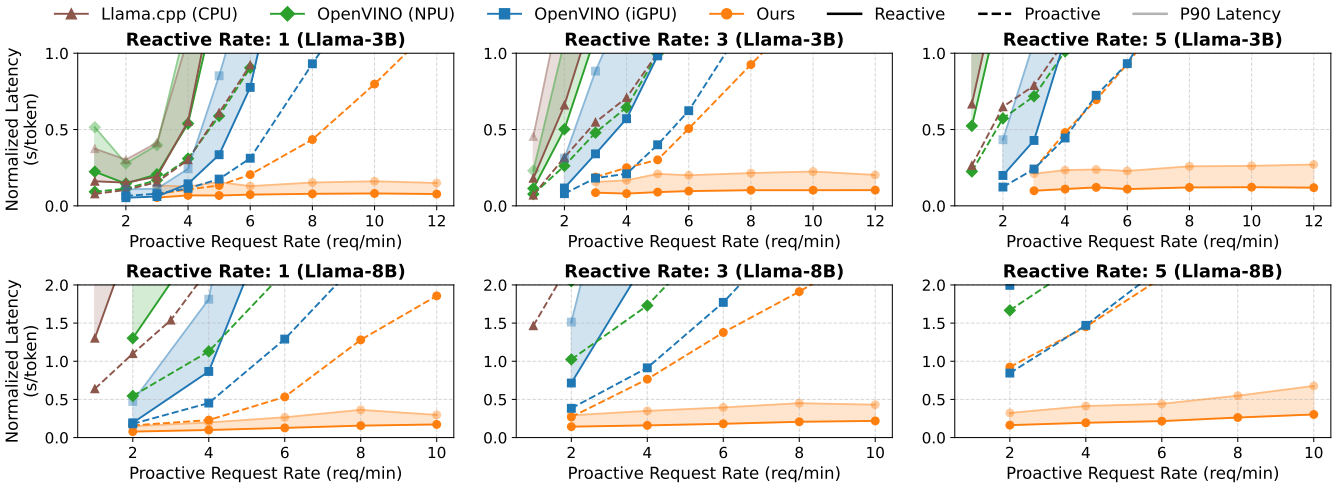

Figure 9. Reactive-Proactive Mixed Workloads. End-to-end results with Llama-3B/8B models on mixed datasets with varying reactive/proactive rate combinations. P90 latencies of reactive requests are displayed alongside mean latencies.

Agent.xpu consistently achieves much lower reactive latencies while sustaining higher proactive throughput. For Llama-3B at 6 proactive req/min, Agent.xpu reduces mean reactive latency by 91.61%, 93.84%, and 96.01% at 1, 3, and 5 reactive req/min, respectively, compared to OpenVINO (iGPU). With Llama-8B, the reductions are 96.23%, 96.01%, and 96.70%. Tail (P90) latency reductions are even larger, underscoring improved user experience. On the proactive side, mean latency improvements reach 34.4%–0.3% for Llama-3B and 58.7%–6.1% for Llama-8B.

Reactive latency in Agent.xpu grows slowly with increasing proactive rates—by only 19% (3B) and 49% (8B) from 4 to 10 proactive req/min—thanks to adaptive batching that caps batch sizes for reactive-first scheduling. Moreover, the gap between P90 and mean latency remains narrow, showing stable performance without outliers. In contrast, baselines degrade quickly under higher load: latencies inflate, tails widen, and responsiveness collapses. In baselines, reactive latencies consistently exceed proactive ones, as independent arrival patterns increase queuing delays under concurrency, and the absence of priority scheduling exacerbates delays for reactive requests. These results confirm the effectiveness of our co-scheduling policies.

6.3 Latency Breakdown

To explain the end-to-end gains, we decompose latencies into pending time, prefill latency, and decode latency (Fig. 10). This serves as both breakdown and ablation.

Baselines suffer long prefill pending times: CPUs are stalled by slow prefill, and NPUs by serial scheduling. By contrast, Agent.xpu keeps reactive prefill pending time to 0.048s on average, enabled by kernel-level sub-100ms preemption. Decode pending appears only in Agent.xpu due to our decoupled scheduling, and grows with proactive load until saturated by the maximum reactive-first batch size.

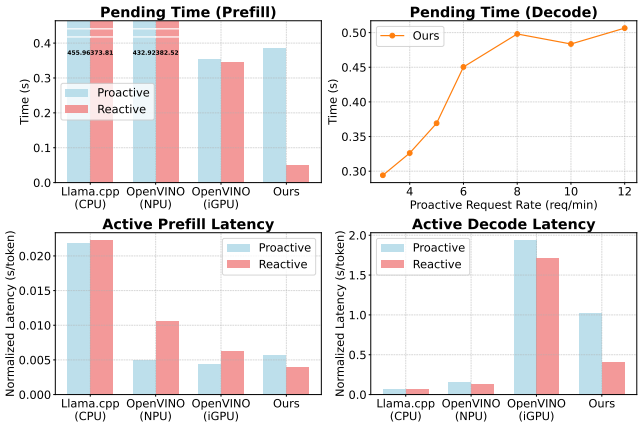

Figure 10. Breakdown of Agentic Serving Latencies. Most measured under mixed workloads (3 reactive and 6 proactive req/min), except the decode pending time with varying proactive rates on Agent.xpu, which is zero on baselines with serial scheduling or continuous batching.

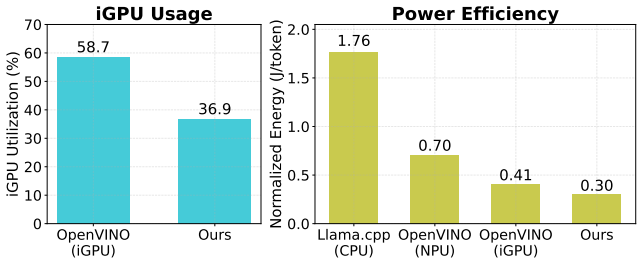

Figure 11. Overhead Analysis. iGPU usage and power efficiency under the same mixed workloads as Fig. 10.

Prefill speeds of Agent.xpu and OpenVINO are aligned, but Agent.xpu accelerates reactive prefill via NPU-iGPU parallelism. Decode shows different trends: CPU benefits from multithreading and large caches, while NPU reflects fast single-batch iGPU decoding. iGPU decode latencies increase due to co-batching with long prefills, but Agent.xpu reduces reactive decode interference through reactive-first batching and fine-grained iGPU arbitration. This explains why reactive requests consistently outperform proactive ones under our design.

6.4 Overhead Analysis

iGPU Utilization. We measure iGPU utilization across inference stages on the same mini PC running Windows 11, using the system resource monitor. Prefill saturates the iGPU (100% utilization), while decode averages 46%. The higher-than-expected decode utilization (relative to FLOPS) arises from memory traffic and synchronization overheads. We account only for active prefill and decode time, excluding duplicated computation from batching. As shown in Fig. 11, Agent.xpu reduces overall iGPU utilization by 37.1%, enabling lighter iGPU load for concurrent user graphics and improved energy efficiency.

Power Efficiency. Using Intel VTune [31], we measure that NPU power remains stable around 10W for chunked prompt length. iGPU power grows from 25W (decode) to 31W (prefill), with minimal sensitivity to batch size ($\leqslant$32). CPU workloads show 12W during single-threaded NPU kernel compilation and up to 58W under multi-threaded inference. Fig. 11 compares per-token energy: Llama.cpp (CPU) is the least efficient, followed by OpenVINO (NPU) with deficient single-batch iGPU decode. Agent.xpu achieves 26.8% lower energy consumption than OpenVINO (iGPU) by offloading prefill to the NPU and employing efficient batching.

7 Related Work

Stateful LLM Serving. Recent systems address stateful and agentic LLM serving with caching or workflow optimizations. InferCept [1] intercepts intermediate states to avoid recomputation, while Parrot [39] and Ayo [60] expose application-level dataflows or primitive graphs for coordinated execution. Autellix [41] and SGLang [77] generalize LLM agents into program-like structures with specialized runtimes. These systems target multi-tenant cloud settings with abundant resources, whereas our focus is on single-device agents operating under tight SoC constraints.

On-Device LLM Inference. On-device inference efforts have explored heterogeneous accelerator use and architectural specialization. HeteroLLM [5] and `llm.npu` [66] exploit NPU–GPU (or CPU) parallelism, while PowerInfer-2 [68] adopts neuron-cluster decomposition for polymorphic execution on smartphones. These works primarily optimize monolithic inference latency relying on accuracy-reducing INT4 quantization, and lack support for stateful workloads. Orthogonal innovations in memory architecture, such as PIM-based co-design [25, 36], flexible DRAM mapping [54], and compute-enabled flash [59], require specialized hardware and limit deployability on personal devices.

Preemptive Scheduling of DNN Workloads. Preemption mechanisms improve responsiveness under concurrent deep learning workloads. PipeSwitch [4] pipelines GPU context switching, PREMA [6] predicts phases for NPU preemption, and REEF [24] achieves microsecond-scale preemption via GPU reset. Pantheon [23] enables fine-grained preemption through sliced execution and early exits. While effective for small-scale DNN tasks on high-end GPUs or custom NPUs, these techniques do not address the unique challenges of heterogeneous SoCs running stateful LLM workloads.

8 Conclusion

The rise of personal LLM agents demands efficient on-device execution, yet the interplay of proactive and reactive workloads remains challenging for heterogeneous SoCs. Agent.xpu

addresses this by co-designing workload-aware scheduling with hardware heterogeneity: its HEG abstraction supports elastic kernel mapping and prefill–decode disaggregation, while the online scheduler balances reactive latency and proactive throughput via fine-grained preemption, piggybacking, and contention-aware dispatch. Our work paves the way for future edge platforms that must serve concurrent, stateful LLM tasks under tight resource constraints.

References

[1] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. 2024. InferCept: Efficient Intercept Support for Augmented Large Language Model Inference. In *International Conference on Machine Learning*. JMLR.org, Vienna, Austria, 81–95.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 technical report. arXiv preprint arXiv:2303.08774.

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

[4] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. {PipeSwitch}: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 499–514.

[5] Le Chen, Dahu Feng, Erhu Feng, Rong Zhao, Yingrui Wang, Yubin Xia, Haibo Chen, and Pinjie Xu. 2025. HeteroLLM: Accelerating Large Language Model Inference on Mobile SoCs platform with Heterogeneous AI Accelerators. arXiv preprint arXiv:2501.14794.

[6] Yujeong Choi and Minsoo Rhu. 2020. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 220–233.

[7] IPEX-LLM contributors. 2024. IPEX-LLM Toolkit. https://github.com/intel/ipex-llm

[8] LangChain contributors. 2023. LangChain: A composable framework to build with LLMs. https://www.langchain.com

[9] LlamaIndex contributors. 2023. LlamaIndex: Build AI Knowledge Assistants over your enterprise data. https://www.llamaindex.ai

[10] Wikipedia contributors. 2025. Apple silicon. https://en.wikipedia.org/w/index.php?title=Apple_silicon&oldid=1282328599

[11] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.

[12] Allan de Barcelos Silva, Marcio Miguel Gomes, Cristiano André Da Costa, Rodrigo da Rosa Righi, Jorge Luis Victoria Barbosa, Gustavo Pessin, Geert De Doncker, and Gustavo Federizzi. 2020. Intelligent personal assistants: A systematic literature review. *Expert Systems with Applications* 147 (2020), 113193.

[13] Yang Deng, Lizi Liao, Zhonghua Zheng, Grace Hui Yang, and Tat-Seng Chua. 2024. Towards human-centered proactive conversational agents. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, Washington DC USA, 807–818.

[14] Core ML developers. 2023. Core ML - Integrate machine learning models into your app. https://developer.apple.com/documentation/coreml/

[15] LiteRT developers. 2025. Google LiteRT Overview. https://ai.google.dev/edge/litert

[16] ONNX Runtime developers. 2021. ONNX Runtime. https://onnxruntime.ai/.

[17] QNN developers. 2025. Qualcomm AI Engine Direct (QNN). https://docs.qualcomm.com/bundle/publicresource/topics/80-63442-50/overview.html

[18] Lutfi Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2024. TinyAgent: Function Calling at the Edge. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. ACL, Miami,

Florida, USA, 80–88.

[19] Shubham Gandhi, Manasi Patwardhan, Lovekesh Vig, and Gautam Shroff. 2024. BudgetMLAgent: A cost-effective LLM multi-agent system for automating machine learning tasks. In *Proceedings of the 4th International Conference on AI-ML Systems*. ACM, Baton Rouge, LA, USA, 1–9.

[20] Georgi Gerganov. 2023. GGML tensor library for machine learning. https://github.com/ggml-org/ggml

[21] Georgi Gerganov. 2023. llama.cpp - Inference of Meta's LLaMA model (and others) in pure C/C++. https://github.com/ggml-org/llama.cpp

[22] Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. 2019. SAMSum Corpus: A Human-annotated Dialogue Dataset for Abstractive Summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*. Association for Computational Linguistics, Hong Kong, China, 70–79. doi:10.18653/v1/d19-5409

[23] Lixiang Han, Zimu Zhou, and Zhenjiang Li. 2024. Pantheon: Preemptible multi-dnn inference on mobile edge gpus. In *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*. 465–478.

[24] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 539–558.

[25] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. 2024. Neupims: Npu-pim heterogeneous acceleration for batched llm inferencing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 722–737.

[26] Karl Moritz Hermann, Tomáš Kočiský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching Machines to Read and Comprehend. arXiv:1506.03340 [cs.CL] https://arxiv.org/abs/1506.03340

[27] Michelle Johnston Holthaus. 2023. Intel Core Ultra Processors. AI Everywhere 2023. https://newsroom.intel.com/client-computing/core-ultra-client-computing-news-1

[28] Wei Huang, Xingyu Zheng, Xudong Ma, Haotong Qin, Chengtao Lv, Hong Chen, Jie Luo, Xiaojuan Qi, Xianglong Liu, and Michele Magno. 2024. An empirical study of LLaMA3 quantization: from LLMs to MLLMs. *Visual Intelligence* 2, 1 (Dec. 2024), 36:1–36:13. doi:10.1007/s44267-024-00070-x

[29] Apple Inc. 2025. Apple Intelligence. https://www.apple.com/apple-intelligence

[30] Qualcomm Inc. 2024. Qualcomm Snapdragon. https://www.qualcomm.com/snapdragon

[31] Intel. 2025. Intel® VTune™ Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.i6xhgk

[32] Yannis Katsis, Sara Rosenthal, Kshitij Fadnis, Chulaka Gunasekara, Young-Suk Lee, Lucian Popa, Vraj Shah, Huaiyu Zhu, Danish Contractor, and Marina Danilevsky. 2025. MTRAG: A Multi-Turn Conversational Benchmark for Evaluating Retrieval-Augmented Generation Systems. arXiv:2501.03468 [cs.CL] https://arxiv.org/abs/2501.03468

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

[34] LangChain. 2025. Introducing ambient agents. https://blog.langchain.dev/introducing-ambient-agents/

[35] Sunjae Lee, Junyoung Choi, Jungjae Lee, Munim Hasan Wasi, Hojun Choi, Steve Ko, Sangeun Oh, and Insik Shin. 2024. MobileGPT: Augmenting LLM with human-like app memory for mobile task automation. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. ACM, Washington D.C., DC, USA, 1119–1133.

[36] Cong Li, Yihan Yin, Xintong Wu, Jingchen Zhu, Zhutianya Gao, Dimin Niu, Qiang Wu, Xin Si, Yuan Xie, Chen Zhang, et al. 2025. H2-LLM: Hardware-Dataflow Co-Exploration for Heterogeneous Hybrid-Bonding-based Low-Batch LLM Inference. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 194–210.

[37] Yuanchun Li, Hao Wen, Weijun Wang, Xiangyu Li, Yizhen Yuan, Guohong Liu, Jiacheng Liu, Wenxing Xu, Xiang Wang, Yi Sun, et al. 2024. Personal LLM agents: Insights and survey about the capability, efficiency and security. arXiv:2401.05459 [cs.HC]

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

[39] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient Serving of LLM-based Applications with Semantic Variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, USA, 929–945.

[40] Yaxi Lu, Shenzhi Yang, Cheng Qian, Guirong Chen, Qinyu Luo, Yesai Wu, Huadong Wang, Xin Cong, Zhong Zhang, Yankai Lin, et al. 2024. Proactive Agent: Shifting LLM Agents from Reactive Responses to Active Assistance. arXiv:arXiv:2410.12361 [cs.AI]

[41] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E Gonzalez, et al. 2025. Autellix: An Efficient Serving Engine for LLM Agents as General Programs. arXiv:2502.13965 [cs.LG]

[42] Chengfei Lv, Chaoyue Niu, Renjie Gu, Xiaotang Jiang, Zhaode Wang, Bin Liu, Ziqi Wu, Qiulin Yao, Congyu Huang, Panos Huang, et al. 2022. Walle: An End-to-End, General-Purpose, and Large-Scale Production System for Device-Cloud Collaborative Machine Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 249–265.

[43] Meta. 2024. Llama 3.2. https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/

[44] Microsoft. 2024. Phi-4-mini-instruct. https://github.com/marketplace/models/azureml/Phi-4-mini-instruct

[45] Microsoft. 2025. Microsoft Copilot for PC. https://www.microsoft.com/en-us/microsoft-copilot

[46] Eduardo Mosqueira-Rey, Elena Hernández-Pereira, David Alonso-Ríos, José Bobes-Bascarán, and Ángel Fernández-Leal. 2023. Human-in-the-loop machine learning: a state of the art. *Artificial Intelligence Review* 56, 4 (2023), 3005–3054.

[47] OpenVINO. 2018. Open-source toolkit for deploying performant AI solutions in the cloud, on-prem, and on the edge alike. https://docs.openvino.ai/2025/index.html

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems* 32 (2019).

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

[50] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large Language Model Connected with Massive APIs. arXiv:2305.15334 [cs.CL]

[51] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings*

of Machine Learning and Systems 5 (2023), 606–624.

[52] Qualcomm. 2024. Deploy Llama-v3.2-3B-Instruct on Snapdragon 8 Elite Mobile. https://aihub.qualcomm.com/mobile/models/llama_v3_2_3b_instruct

[53] Alejandro Rico, Satyaprakash Pareek, Javier Cabezas, David Clarke, Baris Ozgul, Francisco Barat, Yao Fu, Stephan Münz, Dylan Stuart, Patrick Schlangen, Pedro Duarte, Sneha Date, Indrani Paul, Jian Weng, Sonal Santan, Vinod Kathail, Ashish Sirasao, and Juanjo Noguera. 2024. AMD XDNA NPU in Ryzen AI Processors. *IEEE Micro* 44, 6 (2024), 73–82.

[54] Seong Hoon Seo, Junghoon Kim, Donghyun Lee, Seonah Yoo, Seokwon Moon, Yeonhong Park, and Jae W Lee. 2025. FACIL: Flexible DRAM Address Mapping for SoC-PIM Cooperative On-device LLM Inference. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Las Vegas, NV, USA, 1720–1733.

[55] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-throughput generative inference of large language models with a single GPU. In *International Conference on Machine Learning*. PMLR, Honolulu, HI, USA, 31094–31116.

[56] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training multi-billion parameter language models using model parallelism. arXiv:1909.08053 [cs.CL]

[57] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. PowerInfer: Fast large language model serving with a consumer-grade GPU. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. ACM, Austin, TX, USA, 590–606.

[58] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. 2025. Preble: Efficient Distributed Prompt Scheduling for LLM Serving. In *The Thirteenth International Conference on Learning Representations*. openreview.net, Singapore, 26 pages.

[59] Weiyi Sun, Mingyu Gao, Zhaoshi Li, Aoyang Zhang, Iris Ying Chou, Jianfeng Zhu, Shaojun Wei, and Leibo Liu. 2025. Lincoln: Real-Time 50˜100B LLM Inference on Consumer Devices with LPDDR-Interfaced, Compute-Enabled Flash Memory. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Las Vegas, NV, USA, 1734–1750.

[60] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. 2025. Towards End-to-End Optimization of LLM-based Applications with Ayo. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, Rotterdam, Netherlands, 1302–1316.

[61] Chi Wang, Gagan Bansal, Eric Zhu, Beibin Li, Jian Li, Xiaoyun Zhang, Ahmed Awadallah, Ryen White, Doug Burger, Robin Moeur, Victor Dibia, Adam Fourney, Pial Choudhury, Saleema Amershi, Ricky Loynd, Hamed Khanpour, and Ece Kamar. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework. Technical Report.

[62] Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2025. Mobile-Agent-v2: Mobile Device Operation Assistant with Effective Navigation via Multi-Agent Collaboration. *Advances in Neural Information Processing Systems* 37 (2025), 2686–2710.

[63] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 186345.

[64] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. AutoDroid: LLM-powered task automation in Android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. ACM, Washington D.C., DC, USA, 543–557.

[65] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*. PMLR, JMLR.org, Honolulu, HI, USA, 38087–38099.

[66] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. 2025. Fast On-device LLM Inference with NPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ACM, Rotterdam, Netherlands, 445–462.

[67] Jiajun Xu, Zhiyuan Li, Wei Chen, Qun Wang, Xin Gao, Qi Cai, and Ziyuan Ling. 2024. On-device language models: A comprehensive review. arXiv:2409.00088 [cs.CL]

[68] Zhenliang Xue, Yixin Song, Zeyu Mi, Xinrui Zheng, Yubin Xia, and Haibo Chen. 2024. PowerInfer-2: Fast large language model inference on a smartphone. arXiv:2406.06282 [cs.LG]

[69] Ke Yang, Jiateng Liu, John Wu, Chaoqi Yang, Yi R Fung, Sha Li, Zixuan Huang, Xu Cao, Xingyao Wang, Yiquan Wang, et al. 2024. If LLM is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents. arXiv:2401.00812 [cs.CL]

[70] Rongjie Yi, Xiang Li, Zhenyan Lu, Hao Zhang, Daliang Xu, Liming Yang, Weikai Xie, Chenghua Wang, Xuanzhe Liu, and Mengwei Xu. 2023. mllm: fast and lightweight multimodal LLM inference engine for mobile and edge devices. https://github.com/UbiquitousLearning/mllm

[71] Wangsong Yin, Mengwei Xu, Yuanchun Li, and Xuanzhe Liu. 2024. LLM as a system service on mobile devices. arXiv:2403.11805 [cs.OS]

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

[73] Jinliang Yuan, Chen Yang, Dongqi Cai, Shihe Wang, Xin Yuan, Zeling Zhang, Xiang Li, Dingge Zhang, Hanzi Mei, Xianqing Jia, et al. 2024. Mobile foundation model as firmware. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. ACM, Washington D.C., DC, USA, 279–295.

[74] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2025. AppAgent: Multimodal Agents as Smartphone Users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama, Japan, Article 70, 20 pages.

[75] Jieyu Zhang, Ranjay Krishna, Ahmed Hassan Awadallah, and Chi Wang. 2024. EcoAssistant: Using LLM Assistants More Affordably and Accurately. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*. openreview.net, Vienna, Austria, 22 pages.

[76] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric. P Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2023. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. arXiv:2309.11998 [cs.CL]

[77] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. SGLang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems* 37 (2024), 62557–62583.

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

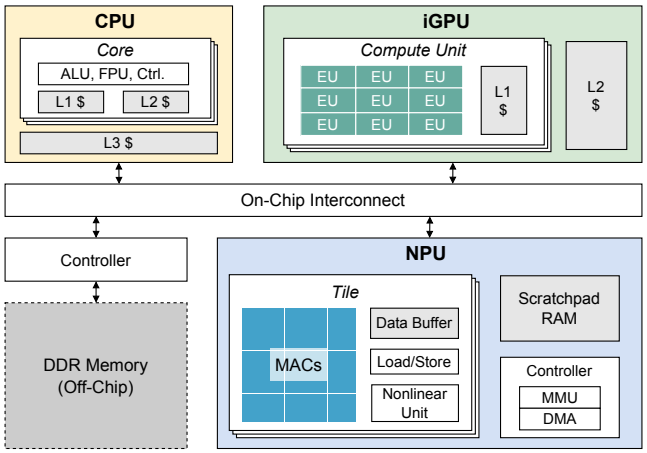

Figure 12. Shared-Memory Hetero-SoC. iGPU builds upon thread-level execution unit (EU), while NPU adopts multiply-accumulate (MAC) array for efficient tensor operations.

A Heterogeneous SoC

As shown in Fig. 12, a heterogeneous System on Chip (SoC) is a highly integrated microchip that combines multiple types of processors or computational units, each specialized for distinct tasks, onto a single chip. This architecture is designed to enhance performance and energy efficiency by executing a wide range of computational workloads more effectively than homogeneous systems.

One of the key components in a heterogeneous SoC is the integrated Graphics Processing Unit (iGPU). Unlike traditional CPUs, which are optimized for single-threaded performance, iGPUs excel in parallel processing. They are built upon a sophisticated array of thread-level Execution Units (EUs), which allow them to manage thousands of concurrent threads. This design makes iGPUs exceptionally well-suited for tasks that require high throughput, such as graphics rendering, image processing, and data parallel workloads. Each EU in the iGPU is capable of executing multiple operations simultaneously, providing the necessary computational power for rendering complex graphics and executing parallel tasks efficiently.

Complementing the iGPU's capabilities is the Neural Processing Unit (NPU), which is tailored specifically for accelerating artificial intelligence (AI) and machine learning (ML) workloads. The NPU harnesses a multiply-accumulate (MAC) array, which is the cornerstone of its design for performing tensor operations, a core component of deep learning models. By leveraging vast arrays of MAC units, the NPU can perform matrix multiplications—the most computationally intensive aspect of neural network operations—at remarkable speed and efficiency. This is crucial for tasks such as real-time inference and training of neural networks, wherein large volumes of data are multiply-accumulated to predict outcomes or adjust model parameters.

The integration of iGPUs and NPUs within a heterogeneous SoC not only ensures that each processing unit can automatically handle specialized tasks in parallel, but it also promotes a symbiotic relationship where both units complement each other. While the iGPU might tackle graphical computations and data-parallel tasks, the NPU can offload AI-centric processes, ensuring optimal resource allocation and improved overall system performance. This architecture is pivotal in modern devices, from smartphones to embedded systems, enabling a seamless and fluid user experience even with demanding applications.

B Analytical Modeling of Kernel Metrics

We model a hetero-SoC accelerator kernel by its total floating-point work W [FLOPs], total data movement Q [Bytes], and arithmetic intensity $I \triangleq W/Q$ [FLOPs/Byte]. Let the target accelerator provide a peak compute throughput P_{pk} [FLOP/s] and a peak off-chip memory bandwidth B_{pk} [B/s]. The classical roofline bound on achievable performance is

$$P(I) \;\le\; \min\big(P_{\text{pk}},\, B_{\text{pk}}\, I\big). \tag{1}$$

To capture non-idealities, we calibrate effective ceilings using two empirical anchor measurements that bracket the operating regimes: (i) a most memory-bound case at intensity I_{m} with measured bandwidth utilization $u_{\text{m}} \in (0, 1]$ (achieved bandwidth divided by B_{pk}), and (ii) a most compute-bound case at intensity I_{c} with measured bandwidth utilization $u_{\text{c}} \in (0, 1]$. From these we define

$$B_{\text{eff}} \triangleq u_{\text{m}}\, B_{\text{pk}}, \tag{2}$$

$$P_{\text{eff}} \triangleq \min\big(P_{\text{pk}},\, u_{\text{c}}\, B_{\text{pk}}\, I_{\text{c}}\big), \tag{3}$$

which are the calibrated memory and compute ceilings, respectively. Equation (2) reflects that in the memory-bound anchor the achieved bandwidth is $u_{\text{m}}B_{\text{pk}}$. Equation (3) follows because, in the compute-bound regime, achieved bandwidth equals P/I; hence $u_{\text{c}} = \frac{(P/I_{\text{c}})}{B_{\text{pk}}}$ implies $P_{\text{eff}} \approx u_{\text{c}}B_{\text{pk}}I_{\text{c}}$, clipped by P_{pk}.

Define the knee (transition) intensity

$$I_{\star} \;\triangleq\; \frac{P_{\text{eff}}}{B_{\text{eff}}} \;=\; \frac{P_{\text{eff}}}{u_{\text{m}}B_{\text{pk}}} \;=\; \frac{u_{\text{c}}}{u_{\text{m}}}\, I_{\text{c}} \quad \text{if } P_{\text{eff}} = u_{\text{c}}B_{\text{pk}}I_{\text{c}}. \tag{4}$$

With these calibrated ceilings, the realized performance, bandwidth, utilization, and latency as functions of arithmetic intensity I are:

Achieved performance and bandwidth.

$$P(I) = \min\big(P_{\text{eff}},\, B_{\text{eff}}\, I\big), \tag{5}$$

$$b(I) = \min\big(B_{\text{eff}},\, P_{\text{eff}}/I\big). \tag{6}$$

Equivalently, using only the two anchor utilizations $(u_{\text{m}}, u_{\text{c}})$ and I_{c}, the achieved bandwidth utilization has a closed-form

expression:

$$u(I) \triangleq \frac{b(I)}{B_{\mathrm{pk}}} = \min\Big(u_{\mathrm{m}},\, u_{\mathrm{c}}\,\frac{I_{\mathrm{c}}}{I}\Big). \tag{7}$$

Then $b(I) = B_{\mathrm{pk}}\,u(I)$ and $P(I) = I\,b(I) = B_{\mathrm{pk}}\,u(I)\,I$.

Latency. Let $T(I)$ denote the runtime (latency). Under the roofline assumption that runtime is governed by the slower of compute or memory,

$$T(I) = \max\Big(\frac{W}{P_{\mathrm{eff}}},\, \frac{Q}{B_{\mathrm{eff}}}\Big) = \frac{W}{P(I)} = \frac{Q}{b(I)}. \tag{8}$$

Using (7), a convenient unified expression is

$$T(I) = \frac{Q}{B_{\mathrm{pk}}\,u(I)} = \frac{W}{B_{\mathrm{pk}}\,u(I)\,I}. \tag{9}$$

Piecewise forms. Explicitly,

$$T(I) = \begin{cases} \dfrac{Q}{u_{\mathrm{m}}B_{\mathrm{pk}}}, & I \le I_\star \quad \text{(memory-bound)},\\[2mm] \dfrac{W}{P_{\mathrm{eff}}}, & I \ge I_\star \quad \text{(compute-bound)}, \end{cases} \tag{10}$$

$$u(I) = \begin{cases} u_{\mathrm{m}}, & I \le I_\star,\\[2mm] u_{\mathrm{c}}\,\dfrac{I_{\mathrm{c}}}{I}, & I \ge I_\star. \end{cases} \tag{11}$$

Remarks and assumptions. (i) This model assumes a single dominant off-chip memory roof. If multiple memory levels matter, take B_{eff} as the active ceiling for the kernel at the given I. (ii) The compute-bound anchor $(I_{\mathrm{c}}, u_{\mathrm{c}})$ should be sufficiently high intensity so that $P(I_{\mathrm{c}}) \approx P_{\mathrm{eff}}$; otherwise P_{eff} inferred by (3) will be conservative. (iii) The clipping in (3) enforces $P_{\mathrm{eff}} \le P_{\mathrm{pk}}$. (iv) Equations (7)–(11) provide closed-form latency and bandwidth utilization predictions at arbitrary I, requiring only the two measured utilizations $(u_{\mathrm{m}}, u_{\mathrm{c}})$, their associated intensity I_{c}, and the problem size through (W, Q) or $I = W/Q$.