HELM: Compiler-Guided Heterogeneous Execution for Large Language Models on Memory-Constrained Systems

by

Shashwat Pandey

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2026 by the
Graduate Supervisory Committee

Aviral Shrivastava, Chair
Aman Arora
Vidya Chhabria

ARIZONA STATE UNIVERSITY

May 2026

ABSTRACT

Large language model (LLM) execution on commodity hardware containing a single graphics processing unit (GPU) remains challenging because model weights and the key-value (KV) cache often exceed available video random access memory (VRAM). Existing inference backends either require all weights to fit in GPU memory or stream layer weights across the Peripheral Component Interconnect Express (PCIe) bus during execution, incurring repeated data transfers that severely limit decode throughput.

This thesis presents HELM, a compiler and runtime for heterogeneous LLM inference that partitions model layers across the central processing unit (CPU) and GPU at compile time. HELM uses hardware-calibrated roofline modeling to search over all feasible layer-to-device assignments and selects the partition that minimizes per-token decode latency. The plan compiles into static per-device execution graphs. This eliminates per-token weight movement and reduces cross-device traffic to a single activation transfer per decode step. HELM further includes a paged KV-cache manager that offloads inactive cache pages to CPU memory, along with asynchronous streaming attention, extending the usable context window well beyond GPU VRAM capacity.

HELM is compared against vLLM, Hugging Face Accelerate, and DeepSpeed ZeRO-Inference on models ranging from 4B to 32B parameters. In memory-constrained systems where existing backends fail with out-of-memory errors, HELM sustains inference by partitioning layers across devices, achieving up to 5.1× higher decode throughput than the strongest feasible baseline while enabling models to use far more context than GPU VRAM alone permits, reaching the full native window where host RAM allows. These results demonstrate that co-designing compile-time layer placement with runtime memory management across CPU and GPU is a viable path toward LLM inference on consumer hardware without multi-GPU or cloud infrastructure.

DEDICATION

To my family and friends, for their unwavering support and encouragement throughout this journey.

ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to my thesis supervisor, Dr. Aviral Shrivastava, whose unwavering guidance and support have been instrumental throughout my research journey. Your expertise and encouragement have inspired me to push the boundaries of my work. My sincere appreciation extends to my thesis committee members, Dr. Aman Arora and Dr. Vidya Chhabria, whose insightful feedback and constructive criticism have significantly enhanced the quality of this thesis. Your contributions have been vital in enriching my research and deepening my analysis. I also wish to thank our research group at the MPS Lab for their constant support throughout the development of this work. Thank you all for your dedication and support; I am truly grateful for your mentorship.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER Page

Chapter 1

INTRODUCTION

Large language models (LLMs) have become an integral part of human life, ranging from basic to complex tasks, including web search, code generation, scientific reasoning, and interactive assistants. Driven by privacy and security concerns, as well as the demand for offline, on-device availability, both the tech industry and end users are pushing towards local LLM execution through tools such as LM Studio LM-Studio (2024) and Ollama Ollama (2024), alongside frameworks like OpenClaw Openclaw (2024) for building personalized agents that interface with locally hosted models. Larger models consistently deliver better performance, and for complex reasoning and problem-solving, at least a 7B model is required Zheng *et al.* (2024); Srivastava *et al.* (2025). However, bigger LLMs' memory requirements are fundamentally incompatible with consumer hardware. For example, a half-precision (fp16) 7 billion parameter (7B) model requires $\approx$16 GB of GPU memory; a 32B model requires $\approx$64 GB. The overwhelming majority of consumer and prosumer GPUs provide only 8–24 GB of VRAM.

This matters in practice, where local development and cost-constrained inference rely on commodity hardware. Enabling efficient inference in these settings expands accessibility and reduces dependence on specialized multi-GPU infrastructure. Existing systems address this problem in ways that leave substantial performance on the table. GPU-only engines like Kwon *et al.* (2023) require the entire model to fit in VRAM, making them unusable for models larger than the available GPU memory.

Hugging Face Accelerate Gugger *et al.* (2022) supports partial CPU offloading via its `device_map=auto` feature, which uses a greedy strategy that first fills the GPU and spills the remaining layers to the CPU without any performance considerations. DeepSpeed ZeRO-Inference Rajbhandari *et al.* (2020) streams model weights from CPU to GPU one layer at a time, incurring $O(L)$ PCIe round-trips (where L represents the number of transformer layers offloaded to CPU), regardless of how little computing each transfer actually enables. Figure 1.1 illustrates these failure modes and contrasts them with HELM's approach.

The goal of this thesis is to build a system that determines the optimal CPU-GPU partition for a given model and hardware configuration during compilation, thereby minimizing inference latency. The system also provides a runtime that executes the model while overlapping computation and communication without per-token weight movement.

This thesis presents **HELM** (Heterogeneous Execution for Large Models), a compiler and runtime system that treats single-node CPU-GPU inference as a compile-time optimization problem. HELM profiles the target hardware, builds a roofline cost model that evaluates every feasible layer-to-device assignment to select the partition that minimizes latency. The compiler lowers this partition into two static execution graphs, one per device, where model weights are placed once and never transferred during token generation. At runtime, only a single hidden-state activation tensor crosses the PCIe boundary per token, replacing the $O(L)$ weight transfers of layer-streaming approaches. HELM also provides a paged KV-cache manager that evicts cold pages to CPU RAM and streams them back on demand, decoupling context length from GPU memory capacity. Figure 1.1 shows the proposed HELM architec-

ture.

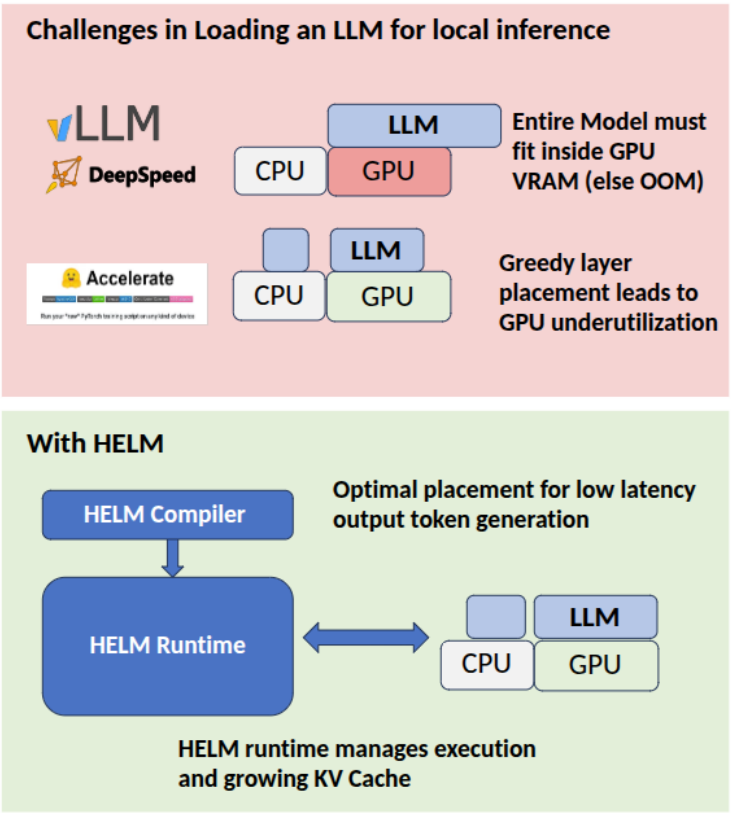

Figure 1.1: (a) vLLM fails on sub-16 GB GPUs; Accelerate's greedy layer assignment is sub-optimal and uses slow per-hook dispatch; DeepSpeed incurs $O(L)$ PCIe round-trips per generated token. (b) HELM's roofline cost model selects the optimal layer split k, compiles static per-device subgraphs (only *activations* cross PCIe once per step), and pages KV to CPU RAM for context lengths beyond GPU VRAM capacity.

1.1 Contributions

This thesis makes the following contributions:

1. **Compiler-driven heterogeneous LLM inference.** HELM formulates CPU-GPU LLM inference as a compile-time optimization problem and introduces a compiler pipeline that generates static per-device execution graphs, eliminating per-token weight transfers (Chapter 4).

2. **Hardware-calibrated roofline partitioning.** HELM develops a transformer block-level cost model that incorporates measured FLOPS, memory bandwidth, and PCIe transfer costs, and uses it to exhaustively search for the globally optimal CPU-GPU partition in constant time (Chapter 4).

3. **Paged KV-cache with streaming attention.** HELM provides a KV-cache manager that decouples context length from GPU memory via paging and integrates with a streaming online-softmax attention kernel, enabling models to use far more context than GPU VRAM alone permits, reaching the full native window where host RAM allows (Chapter 4).

4. **AVX2+F16C general matrix-vector product (GEMV) kernel for CPU-resident layers.** HELM implements a just-in-time (JIT) compiled C++ kernel that replaces PyTorch's unoptimized fp16 linear path on CPU, achieving near-peak memory-bandwidth utilization for the decode GEMV (Chapter 4).

HELM demonstrates up to **5.1**× higher decode throughput than prior CPU-offload systems and enables models to use far more context than GPU VRAM alone permits, reaching the full native window where host RAM allows, while sustaining inference for models that GPU-only systems cannot load at all (Chapter 6).

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background on transformer model structure, inference regimes, and the KV cache. Chapter 3 surveys related work on memory-constrained LLM inference. Chapter 4 presents the HELM system design, covering the compiler pipeline and runtime KV cache

manager. Chapter 5 describes the experimental setup and methodology. Chapter 6 presents the results and evaluation. Chapter 7 concludes and discusses directions for future work.

Chapter 2

BACKGROUND

2.1 Transformer Model Structure

The Original Transformer. The transformer architecture was introduced by Vaswani et al. Vaswani *et al.* (2017) for sequence-to-sequence tasks such as machine translation. The original design is an *encoder–decoder*: the encoder reads the full input sequence and produces a sequence of context vectors; the decoder generates the output token by token, attending to both its own previous outputs (self-attention) and the encoder's context vectors (cross-attention). The key innovation replacing recurrent networks is *multi-head attention*, which computes, for each position, a weighted sum of all other positions' values using learned query, key, and value projections. Stacking this mechanism with a position-wise feed-forward sublayer and residual connections proved highly effective across a wide range of sequence tasks.

Decoder-Only Models and GPT. For language modeling—predicting the next token given all previous tokens—the encoder is unnecessary. Radford et al. Radford *et al.* (2019) showed that a stack of *decoder-only* transformer blocks, trained autoregressively on large text corpora, learns powerful general-purpose representations. Each block applies *causal* self-attention, masking future positions so that token t can only attend to tokens $1, \ldots, t$. This decoder-only design became the foundation for the GPT series and virtually all subsequent large language models, including LLaMA, Mistral, and Qwen3.

Modern LLM Block Structure and Notation. A modern large language model is a stack of L identical transformer blocks, bookended by a token embedding layer at the input and a linear projection to vocabulary logits at the output. Let the input to one block be $X \in \mathbb{R}^{S \times H}$, where S is the number of tokens being processed and H is the hidden dimension.

Each block contains two sublayers. The *attention sublayer* has four weight matrices: $W_Q \in \mathbb{R}^{H \times d}$, $W_K, W_V \in \mathbb{R}^{H \times d_{kv}}$, and $W_O \in \mathbb{R}^{d \times H}$, where $d = H/n_h$ is the per-head dimension and n_h is the number of query heads. Most recent open-weight models use *grouped-query attention* (GQA) Ainslie *et al.* (2023), where $n_{kv} < n_h$ key/value heads are shared across groups of query heads, so $d_{kv} = n_{kv} \cdot d_h < d$, reducing both parameter count and the per-token KV storage cost. The *feed-forward sublayer* uses a *SwiGLU* variant with three weight matrices $W_{\text{gate}}, W_{\text{up}} \in \mathbb{R}^{H \times d_{ff}}$ and $W_{\text{down}} \in \mathbb{R}^{d_{ff} \times H}$, which expand the hidden dimension to an intermediate width d_{ff} and project back. Each sublayer is preceded by RMSNorm and wrapped in a residual connection. In Qwen3-8B (H=4096, d_{ff}=12288), the FFN accounts for roughly 78% of each block's weight bytes.

This block structure makes layer-level partitioning natural: each block is a self-contained unit with a well-defined input and output of the same shape ($S \times H$), so any contiguous group of blocks can be placed on one device and the hidden-state tensor passed to the next device at the boundary.

2.2 Prefill and Decode

2.2.1 Prefill

During *prefill*, all S input tokens are available at once, so the model processes them as a batch. The three projection matrices are applied simultaneously to the full input:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V, \qquad Q, K, V \in \mathbb{R}^{S\times d}. \tag{2.1}$$

The attention scores are then computed over all S positions at once:

$$A = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) V \in \mathbb{R}^{S\times d}. \tag{2.2}$$

The output projection W_O maps this back to $\mathbb{R}^{S\times H}$, completing the attention sublayer. The FFN sublayer then applies its three matrices to all S positions independently. This repeats across all L blocks. At the end, a final linear projection produces output logits for each position; only the logit at position S is sampled to obtain the first generated token.

Each weight projection in Eq. (2.1) is a matrix multiplication of shape $(S, H) \times (H, H)$, requiring $2SH^2$ floating-point operations while loading $2H^2$ bytes of weight data (half precision). The *arithmetic intensity*—FLOPs per byte of memory traffic—is:

$$\text{AI}_{\text{prefill}} = \frac{2SH^2}{2H^2} = S \text{ FLOPs/byte}. \tag{2.3}$$

For a prompt of $S = 512$ tokens, $\text{AI}_{\text{prefill}} = 512$. Every GPU used in this thesis has a ridge point well below 512, so the arithmetic units are fully saturated and prefill is *compute-bound*.

2.2.2 Decode and Why the KV Cache Is Required

After prefill, the model generates one token per step. Let $\mathbf{h}_t \in \mathbb{R}^{1\times H}$ be the hidden state of the newly generated token at step t. The projections now produce single vectors rather than matrices:

$$\mathbf{q}_t = \mathbf{h}_t W_Q, \quad \mathbf{k}_t = \mathbf{h}_t W_K, \quad \mathbf{v}_t = \mathbf{h}_t W_V. \tag{2.4}$$

However, the attention output for token t must attend over *all* previous positions. Writing the accumulated keys and values as $K_{1:t}$ and $V_{1:t}$:

$$\mathbf{a}_t = \text{softmax}\left(\frac{\mathbf{q}_t\, K_{1:t}^\top}{\sqrt{d}}\right) V_{1:t}. \tag{2.5}$$

Without caching, computing $K_{1:t}$ and $V_{1:t}$ at step t would require re-projecting all t previous hidden states through W_K and W_V, a cost that grows as $O(t)$ per step and $O(T^2)$ over a generation of T tokens. The solution is to store $\mathbf{k}_i$ and $\mathbf{v}_i$ as they are computed and simply append $\mathbf{k}_t, \mathbf{v}_t$ at each new step. This persistent storage is the *KV cache* (Section 2.3).

Each weight projection in Eq. (2.4) is now a matrix-vector product of shape $(1, H) \times (H, H)$, requiring $2H^2$ FLOPs while still loading all $2H^2$ weight bytes. The arithmetic intensity drops to:

$$\text{AI}_{\text{decode}} = \frac{2H^2}{2H^2} = 1 \text{ FLOP/byte}, \tag{2.6}$$

regardless of sequence length or model size. This arithmetic intensity of 1 FLOP/byte falls far below the ridge point of every device used in this thesis, so decode is *memory-bandwidth-bound*: the only way to run faster is to stream weight bytes faster, not to add more arithmetic units.

This has a direct implication for HELM. Since decode throughput on any device is proportional to its memory bandwidth rather than its peak FLOPs, a CPU holding k layers contributes to generation at a rate proportional to its DRAM bandwidth. It is slower than a GPU, but not idle. Placing a portion of the model on the CPU and streaming its weights in parallel with the GPU's layers allows the CPU to contribute real token throughput rather than sitting unused as dead weight storage.

2.3 KV Cache

To avoid recomputing attention over all previous tokens at every decode step, the key and value vectors produced by each token are stored in a *KV cache*. Each new token appends one key vector and one value vector per layer per KV head. For a model with L layers, n_{kv} KV heads, and head dimension d_h stored in half precision (2 bytes per element), the cache grows by

$$2 \times L \times n_{kv} \times d_h \times 2 \quad \text{bytes per token.} \tag{2.7}$$

For Qwen3-8B (L=36, n_{kv}=8, d_h=128), this works out to roughly 144 KB per token across all layers. At a context length of 4,096 tokens, the total KV cache reaches approximately 576 MB.

The KV cache creates a second source of memory pressure that is independent of model weights. Even when the weights fit in VRAM, the cache can exhaust the remaining memory at moderate context lengths. On consumer GPUs with 8 to 24 GB of VRAM, where model weights already consume most of the budget, the residual memory for KV entries can be as little as a few hundred megabytes, capping the usable context at a few hundred tokens. This is the problem HELM's paged KV cache manager addresses.

Chapter 3

RELATED WORK

The challenge of deploying large language models on memory-constrained hardware has motivated a diverse body of work. Prior systems differ primarily along two dimensions: what they offload—model weights or the KV cache—and the role assigned to the CPU, ranging from a passive weight store to an active compute participant.

3.1 Weight Offloading Approaches for Large Models

When model weights exceed GPU VRAM, the simplest strategy is to partition weight tensors across the memory hierarchy and stream each partition to the GPU just before it is required. llama.cpp Gerganov (2023) quantizes weights to 2–8 bits and permanently assigns the first N transformer blocks to VRAM while the remainder reside in CPU RAM, where they are executed by the CPU's Single Instruction, Multiple Data (SIMD) units. The number of GPU-resident layers is a static parameter that must be tuned manually by the user. HuggingFace Accelerate Gugger *et al.* (2022) implements `device_map=auto`, which assigns layers to devices in order, filling GPU VRAM before spilling to CPU. CPU-assigned layers are moved to the GPU just before their forward pass and freed immediately after via dispatch hooks, introducing Python-level synchronization at each layer boundary.

vLLM Kwon *et al.* (2023) introduced PagedAttention, a memory management scheme that treats the KV cache as non-contiguous pages rather than pre-allocated contiguous blocks, eliminating fragmentation and enabling continuous batching across

concurrent requests. vLLM additionally supports partial weight offloading to CPU via the `cpu-offload-gb` argument, which streams the offloaded weight subset from CPU to GPU on each forward pass.

DeepSpeed ZeRO-Inference Rajbhandari *et al.* (2020) offloads all model weights to CPU RAM, freeing GPU memory entirely for large batch sizes and input sizes. During the forward pass, each layer's weights are streamed to the GPU via direct memory access (DMA) transfer, computed at full GPU throughput, then immediately evicted with the next layer pre-fetched asynchronously to overlap transfer and execution. However, this incurs $O(L)$ PCIe round-trips per token generation where L is the number of transformer layers, making single-request latency communication-bound unless batches are large enough to amortize the transfer latency—which is generally not the case for local single-user inference.

FlexGen Sheng *et al.* (2023) extends sequential weight offloading to a three-tier hierarchy spanning GPU, CPU, and Non-Volatile Memory Express (NVMe) storage, formulating tensor placement as a linear programme over weights, activations, and KV cache to maximize throughput at batch sizes of $\sim$100–1000.

The shared limitation of these systems is that the CPU is either a serial compute device, as in llama.cpp, or a passive weight store, as in ZeRO-Inference and FlexGen, but never both simultaneously. GPU compute stalls whenever the next weight tile is in transit over PCIe.

3.2 KV Cache Offloading for Long-Context Inference

A complementary memory pressure arises when the model fits in VRAM but the KV cache does not, as occurs with long sequences or large concurrent batches.

LMCache Liu *et al.* (2024) is a KV cache layer that extracts KV entries from inference engines and stores them across a hierarchy of CPU memory, local disk, and remote storage for reuse across queries and engines. It targets workloads with repeated shared prefixes such as multi-turn conversations and retrieval-augmented generation (RAG) pipelines; vLLM integrates LMCache for this purpose, though the focus is on reducing recomputation cost rather than enabling inference on models that exceed VRAM capacity. FastDecode He and Zhai (2024) and NEO Jiang *et al.* (2024) offload KV cache pages and associated attention computation to the CPU, freeing GPU VRAM for additional concurrent requests. LayerKV Xiong *et al.* (2024) performs layer-wise KV offloading with overlapped data transfer to extend effective context length. HeadInfer Luo *et al.* (2025) operates at the finer granularity of individual attention heads, retaining only the heads with the highest importance scores in VRAM. These systems are orthogonal to the weight-offloading problem: they assume the model itself fits in VRAM and address only KV cache pressure.

3.3 CPU-GPU Parallel Execution

Rather than treating the CPU as a passive store or a serial fallback, a more recent line of work uses the CPU and GPU as co-equal compute devices that operate simultaneously on different portions of each forward pass. HeteGen Zhao *et al.* (2024) applies heterogeneous parallelism to the linear projections within each transformer layer, splitting each weight matrix so that a configurable fraction is processed by the CPU while the GPU handles the remainder, with asynchronous overlap designed to hide the PCIe transfer cost. Unlike HeteGen, HELM places whole transformer blocks on a single device and transfers only the activation tensor at one fixed layer boundary,

eliminating per-layer PCIe weight traffic entirely.

TwinPilots Yu *et al.* (2024) categorizes layers as pinned (permanently resident on GPU) or unpinned (resident in CPU RAM) and schedules unpinned layers so that CPU compute time fully hides the DMA transfer latency of the subsequent layer, reporting up to 3.39× throughput improvement over FlexGen. HELM avoids this latency-hiding assumption by keeping all weights stationary on their assigned device; only an 8 KB activation tensor crosses PCIe at the partition boundary, making throughput insensitive to CPU compute speed.

NEO Jiang *et al.* (2024) uses asymmetric pipelining to overlap decoding attention on the CPU with linear operations on the GPU and introduces an adaptive policy that adjusts the CPU-GPU split dynamically as sequence lengths change. HELM instead determines the split at compile time through an exhaustive search over candidate partition points using a roofline cost model, requiring no per-step runtime adaptation.

3.4 Sparsity-Aware Neuron-Level Offloading

PowerInfer Song *et al.* (2023) and Q-Infer Lu *et al.* (2025) exploit power-law activation sparsity in LLM inference: a small subset of neurons, termed *hot*, are activated for the majority of inputs, while *cold* neurons activate infrequently. PowerInfer solves an offline integer linear programme to preload hot neurons onto the GPU and route cold neurons to the CPU, achieving up to 11.69× speedup over llama.cpp by dramatically reducing PCIe traffic. Q-Infer extends this with dynamic asynchronous neuron loading and KV cache-aware scheduling.

The critical limitation of these systems is that they require ReLU-family or specially retrained sparse activation functions. Mainstream dense models, including

LLaMA-3, Qwen3, and Mistral, use SwiGLU activations, which exhibit near-zero structured sparsity, making neuron-level offloading inapplicable without model retraining.

Chapter 4

SYSTEM DESIGN

Existing inference systems treat device placement as a runtime concern. Hugging Face Accelerate uses greedy layer assignment, while DeepSpeed streams layers on demand. Both rely on local decisions, ignoring global interactions among CPU, GPU, and communication costs. Optimal partitioning instead requires jointly minimizing these costs via pre-inference evaluation of all candidate splits.

HELM reframes placement as a *compile-time optimisation problem*. Before any token is generated, HELM: **(1)** traces the model into a cost-annotated intermediate representation; **(2)** measures hardware capabilities via microbenchmarks; **(3)** scores every candidate CPU–GPU partition using a roofline-based cost model; **(4)** compiles the optimal partition into static per-device execution graphs; and **(5)** executes those graphs per output token, managing KV cache growth through paging and streaming attention. Figure 4.1 shows a high-level view of the HELM compiler and runtime.

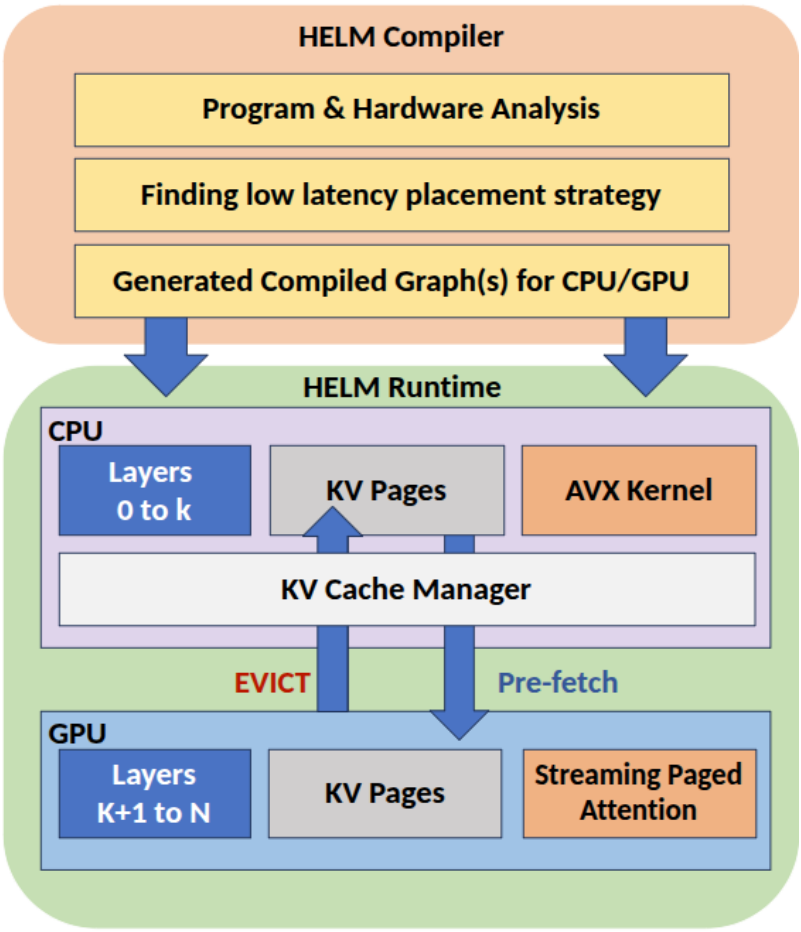

Figure 4.1: High-level overview of HELM. The compiler performs program and hardware analysis to find the optimal low-latency layer placement and generates static CPU and GPU execution graphs. The runtime executes these graphs, with the CPU holding layers 0 to k (using an AVX2 kernel) and the GPU holding layers k+1 to N, while the KV cache manager coordinates page eviction to CPU and prefetching back to GPU for streaming paged attention.

4.1 HELM Compiler

The compiler's goal is to determine, given a model and hardware where the model does not fit entirely on GPU VRAM, the ideal partition that yields minimum end-to-end latency for output token generation. This is done in three stages: **(i)** *model analysis*, which annotates each layer with its compute, memory, and communication

costs; **(ii)** *hardware profiling*, which measures the device capabilities against which those costs are evaluated; and **(iii)** *partition search*, which scores all feasible splits and selects the optimum. Qwen3-8B on an RTX 4060 (8 GB VRAM, 16 GB RAM) serves as the running example throughout. Figure 4.2 shows the complete compiler pipeline.

4.1.1 Program Analysis

To score a partition, the compiler must know what each layer costs: how many parameter bytes it reads per token, how large the activation tensor is at its output, and how much KV-cache traffic it generates per decode step. PyTorch's `torch.fx` tracer produces a compute graph of the model, but its nodes carry none of this information—a raw `torch.fx.Node` has no shape, no FLOP count, and no byte estimate. A placement decision cannot be made from such a graph.

HELM therefore extends the FX graph into a typed intermediate representation (**HelmGraph IR**). A shape-propagation pass infers the input and output tensor shapes of every node; from these, FLOP counts and parameter byte estimates are computed analytically. Each node is then annotated with its inferred shape, dtype, FLOP count, and parameter bytes. A subsequent data-flow analysis pass (**HybridAnalyzer**) walks the annotated graph and aggregates three quantities per layer: W_{param}, the weight bytes read per decode token; W_{act}, the activation bytes that would cross a stage boundary if the block were the last CPU-side block; and W_{kv}, the KV-cache bytes accessed per decode step. The annotated blocks are grouped into **Partition Units**—the atoms of the search—one unit per transformer block plus an embedding unit and an output-projection unit (38 partition units total for Qwen3-

8B).

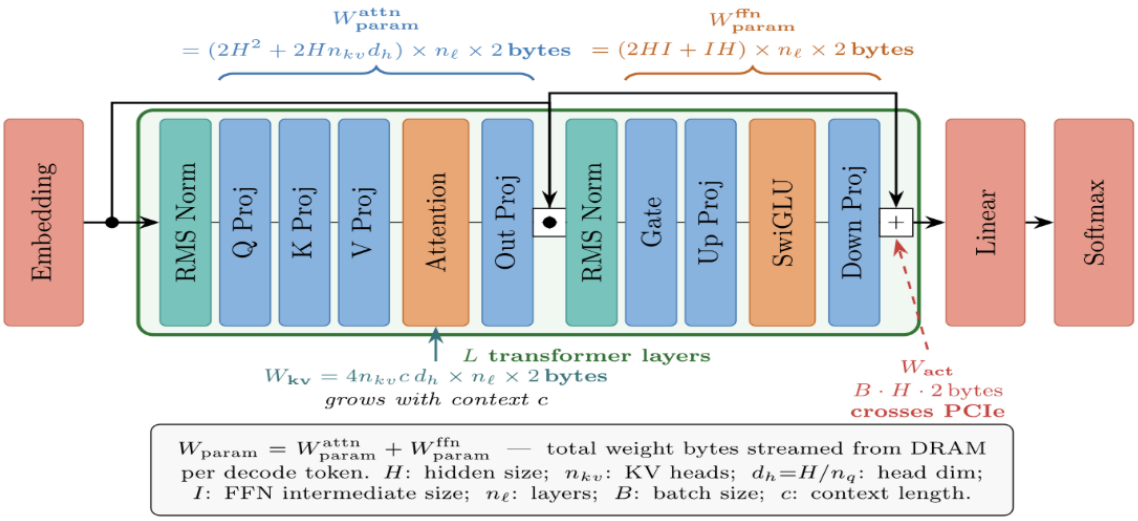

Figure 4.3: A single transformer block annotated with the three cost quantities used by HELM's cost model. W_{param} is the total weight bytes streamed from memory per decode token (dominant cost term). W_{act} is the activation tensor size at the CPU–GPU stage boundary (8 KB for Qwen3-8B at B=1). W_{kv} is the KV-cache bytes read during attention, growing linearly with context length c.

W_{param}: weight bytes per block. For Qwen3-8B (36 transformer blocks, bfloat16), the key dimensions are: H=4096 (hidden size), h_a=32 (query attention heads), h_{kv}=8 (number of key/value heads under Grouped-Query Attention), d_h=128 (dimension per head, so $h_a \cdot d_h = H$), d_{ff}=12288 (FFN intermediate width). Each transformer block's

weight footprint decomposes as:

$$\begin{aligned} W_{\text{attn}} &= \underbrace{H \cdot (h_a d_h)}_{W_Q} + \underbrace{H \cdot (h_{\text{kv}} d_h)}_{W_K} + \underbrace{H \cdot (h_{\text{kv}} d_h)}_{W_V} + \underbrace{(h_a d_h) \cdot H}_{W_O} \\ &= 4096{\times}4096 + 4096{\times}1024 + 4096{\times}1024 + 4096{\times}4096 \\ &= 41.9\text{M params} \times 2\,\text{bytes} = 83.9\,\text{MB} \end{aligned} \tag{4.1}$$

$$\begin{aligned} W_{\text{FFN}} &= \underbrace{H \cdot d_{\text{ff}}}_{W_{\text{gate}}} + \underbrace{H \cdot d_{\text{ff}}}_{W_{\text{up}}} + \underbrace{d_{\text{ff}} \cdot H}_{W_{\text{down}}} \\ &= 3 \times 4096 \times 12288 \\ &= 151.0\text{M params} \times 2\,\text{bytes} = 302.0\,\text{MB} \end{aligned} \tag{4.2}$$

Each block also has two RMSNorm weight vectors (one before attention, one before the FFN), each of length H, contributing $2H \times 2\,\text{bytes} = 0.016\,\text{MB}$—negligible but included for completeness. W_{param} is the total weight bytes the device must stream from memory to process one token through this block; it is the dominant cost term under the roofline model during decode:

$$W_{\text{param}} = W_{\text{attn}} + W_{\text{FFN}} + W_{\text{norm}} = 83.9 + 302.0 + 0.016 \approx 386\,\text{MB per block} \tag{4.3}$$

W_{act}: activation bytes at the CPU-GPU boundary. W_{act} is the activation tensor size at the boundary between the CPU and GPU stages—the tensor that must be transferred over PCIe when the model is split across devices. At decode time (B=1), this is a single hidden-state vector of dimension H:

$$W_{\text{act}} = B \times H \times 2\,\text{bytes} = 1 \times 4096 \times 2 = 8\,\text{KB} \tag{4.4}$$

W_{kv}: KV-cache bytes per block. W_{kv} is the KV-cache bytes that must be read from memory during attention for one block at context length c. Each of the h_{kv}=8

heads stores a key and value vector of length d_h=128 for every past token, in bfloat16:

$$W_{\text{kv}} = 2 \times h_{\text{kv}} \times d_h \times 2\,\text{bytes} \times c = 2 \times 8 \times 128 \times 2 \times c = 4c\,\text{bytes} \tag{4.5}$$

This grows linearly with context length—at c=4096 tokens it reaches 16 KB per block, still small relative to the 386 MB weight footprint. These three quantities W_{param}, W_{act}, W_{kv} are what the cost model consumes.

4.1.2 Hardware Profiling

Two kinds of inputs feed the cost equations. The first—model architecture parameters such as hidden size H, number of heads h_a, KV heads h_{kv}, head dimension d_h, and MLP width I—are read directly from the model's configuration file. From these, quantities like W_{param}, W_{act}, and W_{kv} are computed analytically by the IR analysis pass. No measurement is needed. The second kind—hardware capabilities—must be measured. Three benchmarks run at startup:

(i) DRAM bandwidth: A 64 MB tensor copy measures the sustained memory bandwidth M (GB/s) for each device. This fills the denominator of W_{param}/M in the projection cost term and $W_{\text{kv}}/\hat{M}$ in the attention cost term, where $\hat{M}$ is the effective KV bandwidth blended between DRAM and L3 cache depending on how much of the KV working set fits in cache.

(ii) Compute throughput: For CPU decode, P_{dec} (FLOP/s) is set equal to M directly, because HELM's AVX2+F16C kernel (Section 4.1.5) makes CPU linear layers memory-bandwidth-bound and the compute ceiling is never the bottleneck. For GPU prefill, a $(128, H) \times (H, H)$ matrix multiply measures P_{pre} (FLOP/s), the peak throughput available for processing the prompt.

(iii) PCIe bandwidth and latency: Separate host-to-device and device-to-host

transfers using pinned (page-locked) memory measure B_{link} (GB/s) and ℓ_{link} (ms). Both values feed directly into T_{comm}, the predicted time to transfer one activation tensor across the CPU-GPU boundary per decode step.

Additionally, C_{L3} (L3 cache size in bytes) is queried from `cpuinfo` at startup, and B_{L3} (L3 bandwidth) is estimated as $3 \times M$. Together these populate the *DeviceProfile*:

$$\mathcal{D} = (P_{\text{dec}},\ P_{\text{pre}},\ M,\ C_{\text{L3}},\ B_{\text{L3}}) \tag{4.6}$$

and the *LinkProfile* $\mathcal{L} = (B_{\text{link}},\ \ell_{\text{link}})$. For the running example, the micro-benchmarks yield $M_{\text{CPU}} = 45\,\text{GB/s}$ on host DDR4, $M_{\text{GPU}} = 218\,\text{GB/s}$ on RTX 4060 VRAM, and $B_{\text{link}} = 16\,\text{GB/s}$ on the PCIe interconnect. Combined with the analytically derived per-block costs from the IR analysis—386 MB parameters, 8 KB activation, $4c$ bytes KV—every term in the cost model is now fully specified.

Table 4.1: Cost Model Notation

Symbol	Meaning
From model config (analytic)	
H	Hidden size
$h_a,\ h_{\text{kv}},\ d_h$	Query heads, KV heads, head dimension
I	MLP intermediate width
d_{sz}	Bytes per element
W_{param}	Weight bytes per stage
W_{act}	Activation size at stage boundary
W_{kv}	KV-cache bytes read per decode step
From micro-benchmarks (measured)	
M	Sustained DRAM bandwidth
$P_{\text{dec}},\ P_{\text{pre}}$	Decode / prefill compute throughput
$B_{\text{link}},\ \ell_{\text{link}}$	PCIe bandwidth and round-trip latency
Derived	
$\hat{M}$	Effective KV bandwidth (L3-blended)
α	Fraction of KV working set in L3
$\eta_c,\ \eta_m$	Compute and memory efficiency coefficients
T_{plan}	Predicted per-token decode latency

4.1.3 Partition Search

The compiler now has everything it needs: per-block costs from the IR and hardware capabilities from the profiler. The intuition for the search is straightforward. Moving blocks from GPU to CPU reduces GPU memory pressure—making more configurations feasible—but increases CPU compute time. Moving blocks the other way reduces CPU time but eventually renders the plan infeasible when weights no longer fit in VRAM. An optimal boundary k balances these opposing forces, and it is found by evaluating the total predicted latency for every candidate split and selecting the minimum.

Total plan cost. k is the number of CPU-side transformer blocks, L the total number of blocks, and T_{comm} the activation transfer cost at the stage boundary. The total predicted per-token decode latency is:

$$T_{\text{plan}}(\pi) = T^{\text{CPU}}_{\text{decode}}(k) + T^{\text{GPU}}_{\text{decode}}(L - k) + T_{\text{comm}} \tag{4.7}$$

Each stage cost follows the roofline model: execution time is bounded by whichever of the compute throughput or the memory bandwidth is the tighter constraint. For a stage on device $\mathcal{D}$, processing a batch of B sequences at context length c, the per-token decode time decomposes into a projection term and an attention term:

$$T_{\text{decode}} = T_{\text{proj}} + T_{\text{attn}} \tag{4.8}$$

Projection term. H (hidden size), $d_{\text{kv}} = h_{\text{kv}} d_h$ (KV projection dimension), I (MLP intermediate size), and n_ℓ (number of blocks in the stage) are read from the model config. W_{param} is computed analytically; P_{dec} (FLOP/s) and M (DRAM bandwidth) come from the micro-benchmarks; $\eta_c, \eta_m \in (0, 1]$ are hardware efficiency

coefficients fit from profiling:

$$F_{\text{proj}} = 2B\big(2H^2 + 2Hd_{\text{kv}} + 3HI\big) \cdot n_\ell \tag{4.9}$$

$$T_{\text{proj}} = \max\left(\frac{F_{\text{proj}}}{P_{\text{dec}}\,\eta_c},\ \frac{W_{\text{param}}}{M\,\eta_m}\right) \tag{4.10}$$

During decode B=1 and each linear layer is a matrix-vector product—one multiply-accumulate per weight byte—so the compute-to-memory ratio is $O(1)$ and execution is almost always memory-bandwidth-bound. For Qwen3-8B: streaming 386 MB of weights takes $386/45 \approx 8.6$ ms on CPU and $386/218 \approx 1.8$ ms on GPU per block.

Attention term. h_a (query heads), h_{kv} (KV heads under Grouped-Query Attention), d_h (head dimension), and d_{sz} (bytes per element) are read from the model config. c comes from the workload description. F_{attn} and W_{kv} are derived analytically; $\hat{M}$ is the effective KV bandwidth:

$$F_{\text{attn}} = 4B\,h_a\,c\,d_h \cdot n_\ell \tag{4.11}$$

$$W_{\text{kv}} = 2\,B\,n_\ell\,c\,h_{\text{kv}}\,d_h\,d_{\text{sz}} \tag{4.12}$$

$$T_{\text{attn}} = \max\left(\frac{F_{\text{attn}}}{P_{\text{dec}}\,\eta_c},\ \frac{W_{\text{kv}}}{\hat{M}\,\eta_m}\right) \tag{4.13}$$

On CPU, the KV working set may fit in L3 cache at short contexts, making reads faster than DRAM. DRAM and L3 bandwidth are blended proportionally using α, the fraction of the KV working set that fits in cache:

$$\hat{M} = \alpha\,B_{\text{L3}} + (1-\alpha)\,M, \quad \alpha = \min\left(1,\ \frac{C_{\text{L3}}}{W_{\text{kv}}}\right) \tag{4.14}$$

As context grows, $\alpha \to 0$ and $\hat{M} \to M$ (pure DRAM bandwidth).

Communication term. W_{act} is the activation tensor size, derived analytically from the model config. ℓ_{link} (PCIe latency) and B_{link} (PCIe bandwidth) come from

the PCIe microbenchmark. For single-device plans $T_{\text{comm}} = 0$:

$$T_{\text{comm}} = \ell_{\text{link}} + \frac{W_{\text{act}}}{B_{\text{link}}} \tag{4.15}$$

For Qwen3-8B, $W_{\text{act}} = 8\,\text{KB}$, giving $T_{\text{comm}} \approx 5\,\mu\text{s}$—negligible relative to the compute terms.

Not every plan is feasible: the GPU-side weights plus KV cache must fit in VRAM. Feasibility is checked with:

$$W_{\text{param}}(L - k) + \mathbf{1}_{\text{kv-GPU}} \cdot W_{\text{kv}}^{\text{total}} \leq \text{VRAM}_{\text{avail}} \tag{4.16}$$

where $\mathbf{1}_{\text{kv-GPU}} = 0$ when KV offload is enabled and 1 otherwise. T_{plan} is evaluated for all feasible $k \in \{0, \ldots, L\}$ and the plan that minimizes it is returned.

Running example. All-GPU (k=0) is infeasible: $36 \times 386\,\text{MB} = 13.9\,\text{GB}$ exceeds the 8 GB VRAM budget. All-CPU (k=36) gives $T_{\text{plan}} = 36 \times 8.6 \approx 310\,\text{ms}$ ($\approx 3.2\,\text{tok/s}$). After reserving $\approx$1 GB for the CUDA context and runtime, $\text{VRAM}_{\text{avail}} \approx 7\,\text{GB}$ leaves room for at most $\lfloor 7000/386 \rfloor = 18$ transformer blocks on GPU as an initial estimate. The full search operates over all 38 partition units (including the embedding and output-projection units, which together add $\approx 1.2\,\text{GB}$), and selects k^*=21: 21 units on CPU ($T_{\text{cpu}} \approx 154.8\,\text{ms}$), 17 on GPU ($T_{\text{gpu}} \approx 30.6\,\text{ms}$), plus $T_{\text{comm}} \approx 5\,\mu\text{s}$, giving $T_{\text{plan}} \approx 186\,\text{ms}$ ($\approx 5.4\,\text{tok/s}$)—consistent with the measured partition in Table 6.5. Algorithm 1 gives the complete procedure.

4.1.4 Static Graph Reconstruction

Once the optimal k is determined, the compiler reconstructs the FX graph as two independent per-device subgraphs, eliminating all placement logic from the token loop. The flat graph is split after the last node belonging to block $k-1$; the activation

Algorithm 1 StrategySelector: Exhaustive Roofline Partition Search

Require: Partition units $\{u_i\}_{i=0}^{L}$, device profiles $\mathcal{D}_{\text{cpu}}$, $\mathcal{D}_{\text{gpu}}$, link profile $\mathcal{L}$, $\text{VRAM}_{\text{avail}}$, objective $\in$ {decode, prefill}

Ensure: Optimal partition plan π^*

$\pi^* \leftarrow \perp$; $T^* \leftarrow \infty$
for $k \leftarrow 0$ **to** L **do**
  $W_{\text{gpu}} \leftarrow \sum_{i=k}^{L} u_i.\text{param\_bytes}$
  if $W_{\text{gpu}} > \text{VRAM}_{\text{avail}}$ **then**
    continue
  end if
  $T_{\text{cpu}} \leftarrow \textsc{RooflineTime}(\{u_i\}_{i<k},\ \mathcal{D}_{\text{cpu}},\ \text{objective})$
  $T_{\text{gpu}} \leftarrow \textsc{RooflineTime}(\{u_i\}_{i\geq k},\ \mathcal{D}_{\text{gpu}},\ \text{objective})$
  if $k = 0$ **or** $k = L$ **then**
    $T_{\text{comm}} \leftarrow 0$
  else
    $T_{\text{comm}} \leftarrow \mathcal{L}.\ell + u_k.\text{act\_bytes} \,/\, \mathcal{L}.B$
  end if
  $T_{\text{plan}} \leftarrow T_{\text{cpu}} + T_{\text{gpu}} + T_{\text{comm}}$
  if $T_{\text{plan}} < T^*$ **then**
    $T^* \leftarrow T_{\text{plan}}$; $\pi^* \leftarrow (k,\ \mathcal{D}_{\text{cpu}},\ \mathcal{D}_{\text{gpu}})$
  end if
end for
return π^*

tensor at that point is marked as the output of the CPU subgraph and the input of the GPU subgraph. Each subgraph is a standalone `torch.fx.GraphModule` that can be executed independently on its assigned device.

Because placement is fixed at compile time, the token loop reduces to three operations: run the CPU subgraph, transfer the 8 KB activation tensor across PCIe, and run the GPU subgraph. There are no per-token placement decisions, no weight movement, and no graph recompilation. Before the first request, model weights are streamed to their target devices one submodule at a time rather than loading the full 16 GB model into CPU RAM at once, avoiding the transient memory spike that would otherwise exhaust host memory before any weights are sent to the GPU. Batch sizes $B > 1$ are supported; model weights are applied once across all B requests per step, amortising the per-token bandwidth cost by a factor of B.

4.1.5 CPU Execution and Roofline Validation

The cost model treats the CPU stage as memory-bandwidth-bound: the time to stream W_{param} bytes from DRAM should dominate over floating-point work. This assumption underpins the roofline prediction—but it only holds if the CPU linear layers actually reach the DRAM bandwidth ceiling.

On x86, PyTorch's fp16 linear layer falls back to a scalar loop because AVX2 has no native fp16 fused multiply-accumulate instruction. On the i7 test system, this achieves only ≈ 3.7 GB/s—roughly 12× below the 45 GB/s DRAM ceiling and well into the compute-bound regime. The roofline prediction would be wrong by the same factor.

HELM ships a JIT-compiled AVX2+F16C GEMV kernel that closes this gap.

Weights are loaded with `_mm256_cvtph_ps` (F16C), multiplied with `_mm256_fmadd_ps` (AVX2 FMA), and accumulated in fp32. On the i7 test system this achieves $\approx$ 50 GB/s—at the practical DDR4 ceiling—a **13**$\times$ speedup over the PyTorch fallback. For prefill (seq>1), a separate kernel casts fp16 weights to fp32 and calls Intel Math Kernel Library (MKL)'s `cblas_sgemm`, achieving a **21**$\times$ speedup over PyTorch's fp16 general matrix-matrix product (GEMM) path. The kernel is compiled on first use via `cpp_extension.load` and automatically disabled on non-AVX2 hardware.

Returning to the running example: without the kernel, the 21-block CPU stage would take $\approx$2.4 s per token, making the k=21 plan far worse than all-CPU PyTorch. With the kernel, it takes $\approx$ 196 ms, validating the cost model's memory-bandwidth-bound assumption and making the partition decision meaningful.

The five compiler steps described above form a complete pipeline. Algorithm 2 states it concisely. $\mathcal{M}$ is the PyTorch `nn.Module`; $\mathcal{W} = (S, T_{\text{out}}, B)$ is the workload descriptor (prompt length, max output tokens, batch size); $\mathcal{H}$ describes the available devices; τ_{wm} is the KV watermark threshold passed to the runtime allocator. The compile phase runs *once* per model/hardware pair; the runtime phase repeats per request.

Algorithm 2 HELM: Heterogeneous Compile and Execute

Require: $\mathcal{M}$: `nn.Module`, workload $\mathcal{W}$=(S, T_{out}, B), hardware $\mathcal{H}$

Ensure: Token stream for input prompt **Compile phase** (once per model / hardware pair)

$G \leftarrow \text{FXTrace}(\mathcal{M}, \mathcal{W})$

$\mathcal{IR} \leftarrow \text{HelmGraphIR}(G)$

$\{c_i\} \leftarrow \text{HybridAnalyze}(\mathcal{IR}, \mathcal{W})$

$\{u_i\}_{i=0}^{L} \leftarrow \text{PartitionUnits}(\{c_i\})$

$(\mathcal{D}_{\text{cpu}}, \mathcal{D}_{\text{gpu}}, \mathcal{L}) \leftarrow \text{DeviceProfile}(\mathcal{H})$

$\pi^* \leftarrow \text{StrategySelector}(\{u_i\}, \mathcal{D}_{\text{cpu}}, \mathcal{D}_{\text{gpu}}, \mathcal{L})$

$\{S_i\} \leftarrow \text{StageFXBuild}(G, \pi^*)$ **Runtime phase** (per request)

$\text{KV} \leftarrow \text{KVOffloadManager}(\pi^*, \tau_{\text{wm}})$

$\mathbf{h} \leftarrow \text{PrefillExecute}(\{S_i\}, \text{tokens}, \text{KV})$

while $\neg\, \text{Done}(\mathbf{h})$ **do**

  $(\mathbf{h}, t) \leftarrow \text{DecodeExecute}(\{S_i\}, \mathbf{h}, \text{KV})$

  yield t

end while

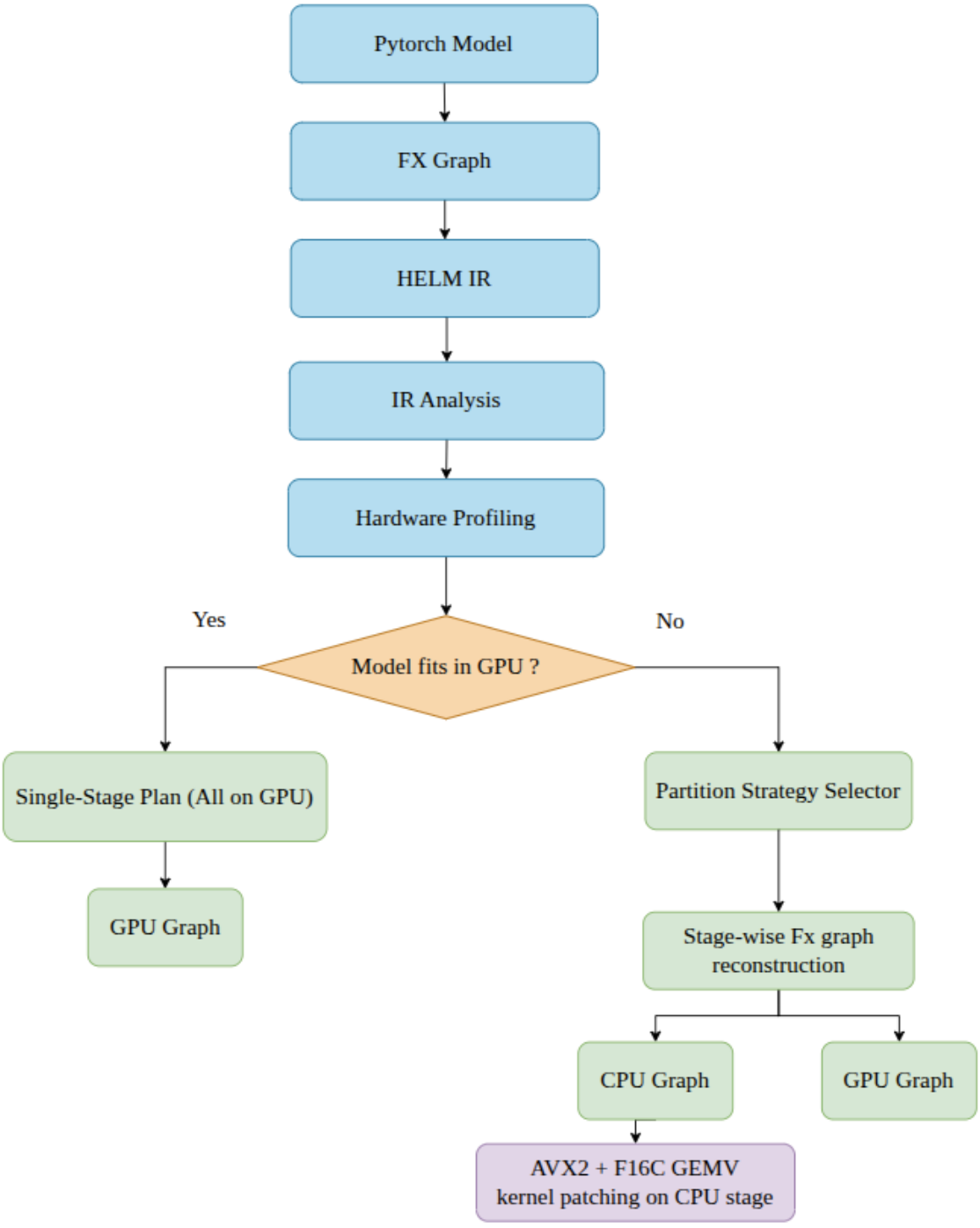

Figure 4.2: HELM compiler pipeline. A PyTorch model is traced into the HELM-Graph IR, annotated with per-layer costs, and combined with measured hardware profiles to search for the optimal CPU–GPU partition boundary k^*. The resulting subgraphs are compiled once and executed without modification at runtime.

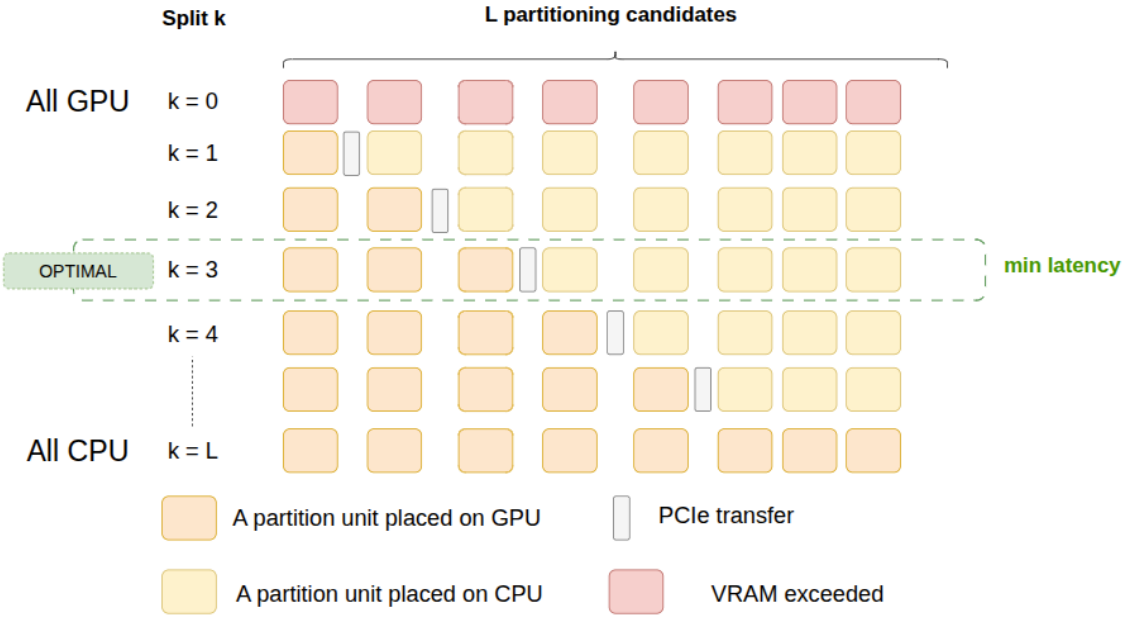

Figure 4.4: Partition search visualization for Qwen3-8B on an RTX 4060 (8 GB VRAM). Each row represents a candidate split k; the PCIe boundary slides right as more blocks move to the GPU. The optimal split k^*=18 minimises $T_{\text{plan}} = T_{\text{cpu}} + T_{\text{gpu}} + T_{\text{comm}}$.

4.2 Runtime

With the partition plan fixed and the graphs compiled, the runtime maintains a simple invariant: every decode step executes the same three steps in the same order—CPU stage, PCIe transfer, GPU stage—with no placement logic in the loop.

4.2.1 Pipeline Execution

One might ask whether the CPU and GPU stages could be pipelined to reduce latency—running the GPU stage for token t concurrently with the CPU stage for token $t + 1$. The answer is no: the GPU stage for token t produces the output embedding that the CPU stage for token $t+1$ needs as input. Autoregressive decode is serialised by data dependency; there is no useful parallelism to expose between stages. Sequential execution is therefore not a limitation of HELM's design but a consequence of the task.

Execution proceeds in two phases. During *prefill*, the full prompt of length S is processed in a single forward pass through both stages, populating the KV cache. During *decode*, one new token is processed per step. Both phases use the same compiled subgraphs with no recompilation. For Qwen3-8B the inter-stage activation is 8 KB—a PCIe transfer dominated by round-trip latency ($\approx 5\,\mu$s), negligible relative to the 196 ms CPU stage.

4.2.2 KV Cache Management

The KV cache problem is more severe than it might initially appear. For Qwen3-8B on the RTX 4060, the 17 GPU-side partition units (16 transformer blocks plus the output projection) already occupy 7.16 GB of the 8 GB VRAM budget, leaving

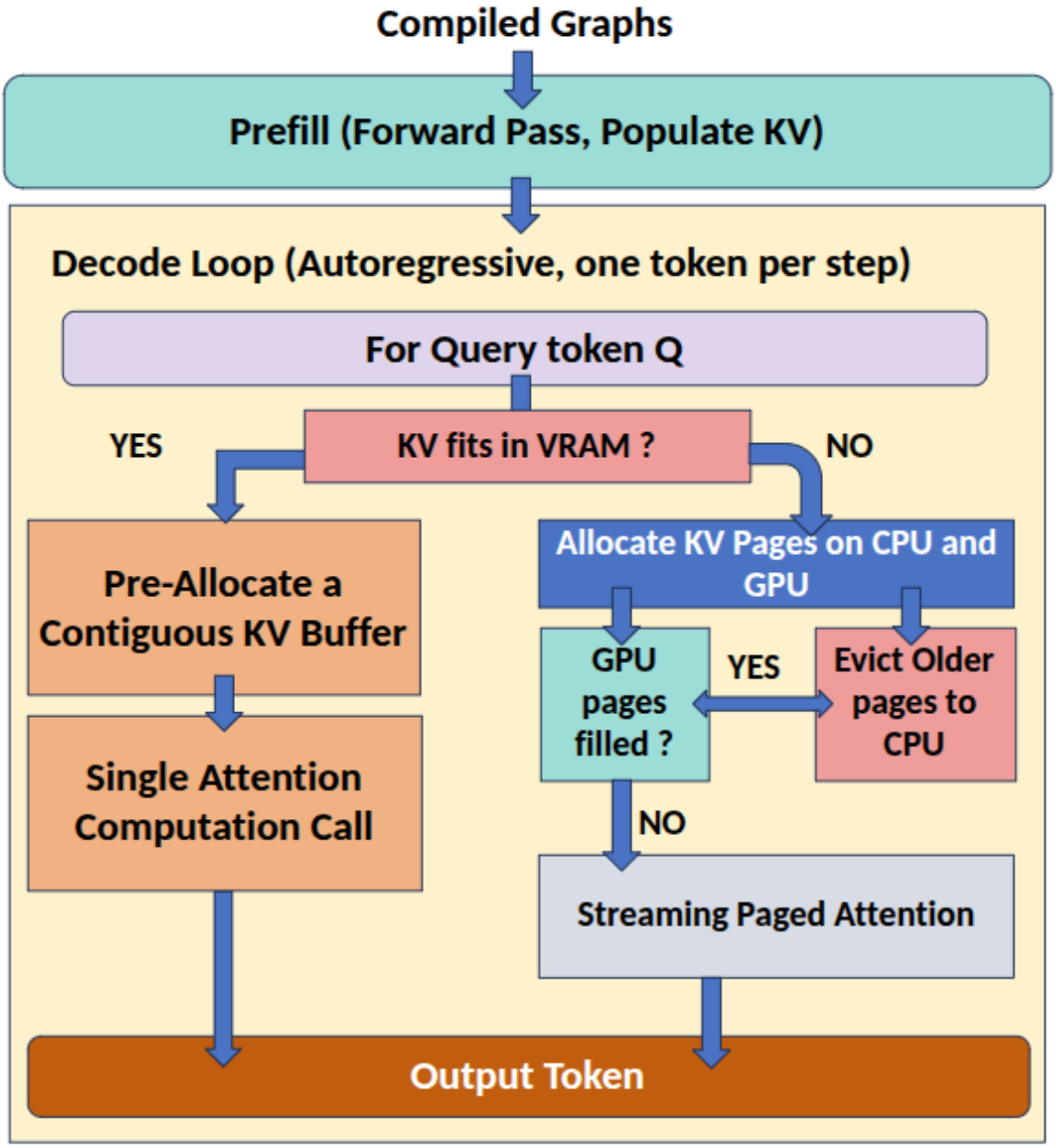

Figure 4.5: HELM runtime KV cache management. When KV state fits entirely in VRAM, all writes go through the contiguous pre-allocated buffer and a single Scaled Dot-Product Attention (SDPA) kernel handles attention. When VRAM pressure exceeds the watermark, the oldest full pages are evicted to CPU-pinned memory; streaming paged attention then reconstructs the full context via online softmax accumulation and asynchronous H2D prefetch over a dedicated CUDA copy stream.

under 1 GB for KV entries.

The obvious fix—a large contiguous KV buffer—does not work here. A contiguous buffer must be allocated in full at initialisation time (so the kernel has a fixed address range to index into), which immediately consumes the remaining VRAM and prevents any useful context growth. Alternatively, growing it dynamically requires GPU memory fragmentation that CUDA's allocator does not support efficiently, and evicting a contiguous buffer means evicting the *entire* KV history at once, discarding context irreversibly.

KV memory is managed in fixed-size *pages* instead, each holding 512 tokens' worth of keys and values across all layers. Pages can be independently allocated, evicted, and fetched back, giving fine-grained control over which portion of the context stays GPU-resident. A GPU page pool handles the active context window; a CPU pinned (page-locked) memory pool serves as overflow. When GPU KV usage exceeds a configurable watermark (default: 80% of available VRAM after weights), the oldest page is evicted to CPU via a non-blocking `cudaMemcpyAsync`. The watermark is set below 100% deliberately: triggering eviction before the buffer is full ensures there is always room for the current step's new KV entries, and the async transfer overlaps with the ongoing decode step, so eviction stays off the critical path in steady state.

4.2.3 Streaming Paged Attention

4.2.3.1 How Attention Is Computed

At each decode step the model holds a *query* vector $\mathbf{q}$—the question "what do I need right now?" Every token in the context has two vectors stored in the KV cache: a *key* $\mathbf{k}_i$ ("what do I represent?") and a *value* $\mathbf{v}_i$ ("what information do I carry?"). For

each past token i, the model computes an attention score $e_i = \mathbf{q} \cdot \mathbf{k}_i / \sqrt{d_h}$, measuring relevance to the current query. Scores are normalised via softmax into attention weights $a_i = e^{e_i} / \sum_j e^{e_j}$, and the attention output is the weighted mixture of values:

$$\mathbf{o} = \sum_i a_i \mathbf{v}_i \tag{4.17}$$

4.2.3.2 The Paging Problem

Paging solves the memory problem but creates an attention problem: at each decode step, scores are needed over the full context, yet some KV pages are no longer GPU-resident. Fetching all evicted pages back before the attention kernel runs requires assembling a contiguous $S \times d_{\text{model}}$ buffer on GPU—precisely the allocation paging was designed to avoid; at long contexts this buffer alone would exceed VRAM. Skipping evicted pages would silently drop context tokens, producing incorrect output. Neither option is acceptable.

4.2.3.3 Streaming Paged Attention Algorithm

The root obstacle is the softmax denominator $\sum_j e^{e_j}$: it requires every score before any weight can be finalised, so Equation (4.17) cannot be split across pages naively. HELM resolves this by *deferring the division*—accumulating a numerator and a denominator separately across pages and dividing only once at the end. Three scalars are maintained throughout the page loop:

- m — running maximum score seen so far. Kept purely for numerical safety: subtracting m before exponentiating prevents overflow without changing the final ratio.

- s — running sum of exponentiated (and m-shifted) scores, i.e. the softmax denominator.

- $\mathbf{o}$ — running weighted sum of value vectors, i.e. the softmax numerator.

After the last page, the final output is simply $\mathbf{o}/s$. The only subtlety arises when a new page reveals a score larger than the current m: all earlier contributions were scaled relative to the old maximum and must be corrected by multiplying both $\mathbf{o}$ and s by $\exp(m_{\text{old}} - m_{\text{new}})$ before merging the new page. Because $m_{\text{new}} > m_{\text{old}}$, this factor is less than one, shrinking stale contributions by exactly the right amount. GPU memory during the attention pass is bounded to one resident page at a time, regardless of total context length. Per-page the update rules are:

$$m_i = \max\left(m_{i-1},\ \max_j e_{ij}\right) \tag{4.18}$$

$$s_i = e^{m_{i-1}-m_i}\, s_{i-1} + \textstyle\sum_j e^{e_{ij}-m_i} \tag{4.19}$$

$$\mathbf{o}_i = e^{m_{i-1}-m_i}\, \mathbf{o}_{i-1} + \textstyle\sum_j e^{e_{ij}-m_i}\, \mathbf{v}_{ij} \tag{4.20}$$

where e_{ij} is the attention score of the j-th token in page i. The term $e^{m_{i-1}-m_i}$ is the rescaling factor: it equals 1 when m does not change, and shrinks the stale s and $\mathbf{o}$ when a larger score is found.

Pages not currently GPU-resident are returned via an *async prefetch*: a CUDA event inserted after the previous decode step triggers a host-to-device copy for the next required page immediately, hiding transfer latency behind GPU compute so the attention loop never stalls.

Algorithm 3 gives the complete procedure.

Algorithm 3 StreamingPagedAttention

Require: Query $\mathbf{q}$, ordered page list $\mathcal{P} = [P_1, \ldots, P_n]$, head dimension d_h

Ensure: Attention output $\mathbf{o}_{\text{out}}$

$m \leftarrow -\infty; \quad s \leftarrow 0; \quad \mathbf{o} \leftarrow \mathbf{0}$ {init accumulator}

AsyncPrefetch(P_1) {start first H2D transfer}

for $i \leftarrow 1$ **to** n **do**

  SyncEvent(P_i) {wait until P_i is GPU-resident}

  if $i < n$ **then**

    AsyncPrefetch(P_{i+1}) {overlap transfer with compute}

  end if

  $\mathbf{e} \leftarrow \mathbf{q}\, K_i^\top / \sqrt{d_h}$ {attention scores for tokens in P_i}

  $m_{\text{new}} \leftarrow \max(m,\ \max(\mathbf{e}))$

  if $m_{\text{new}} > m$ **then**

    $r \leftarrow \exp(m - m_{\text{new}}); \quad s \leftarrow r \cdot s; \quad \mathbf{o} \leftarrow r \cdot \mathbf{o}$ {rescale stale state}

    $m \leftarrow m_{\text{new}}$

  end if

  $\mathbf{w} \leftarrow \exp(\mathbf{e} - m)$ {m-shifted softmax weights}

  $s \leftarrow s + \sum_j w_j$

  $\mathbf{o} \leftarrow \mathbf{o} + \sum_j w_j\, \mathbf{v}_{i,j}$

end for

return $\mathbf{o} \,/\, s$

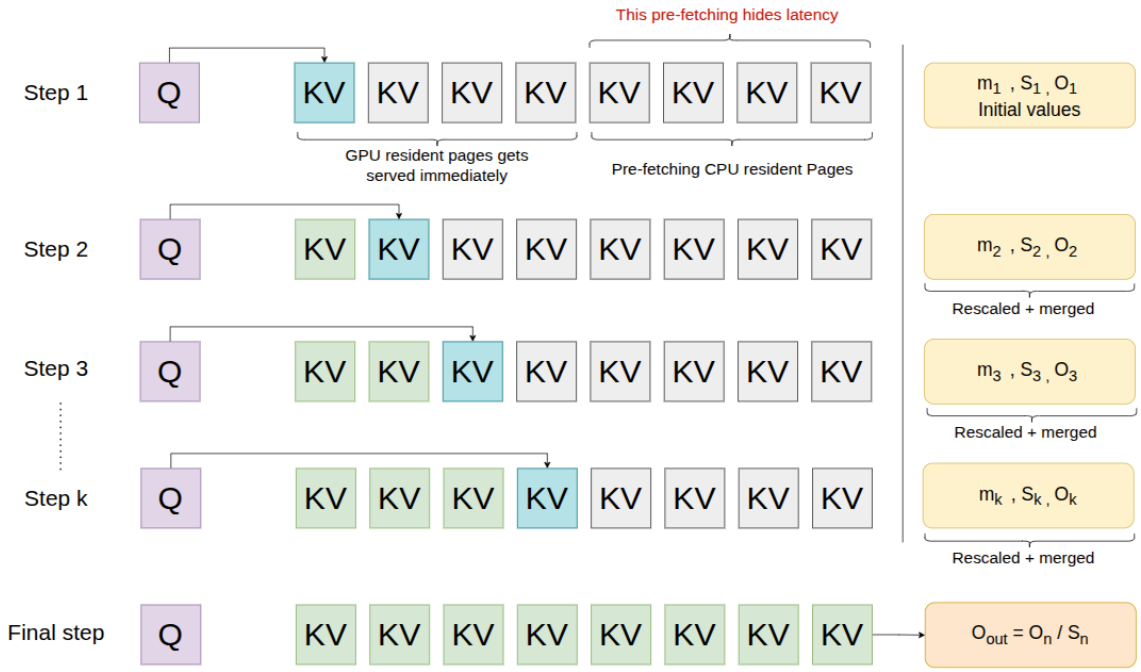

Figure 4.6: Streaming paged attention: the query attends to one KV page at a time while the online softmax accumulator $(m, s, \mathbf{o})$ is updated incrementally. Only the active page needs to be GPU-resident; all other pages remain in CPU-pinned memory and are prefetched asynchronously.

4.2.3.4 Running Example

The correctness of streaming paged attention is illustrated with a concrete six-token context. For clarity, each token's key vector is summarised by its scalar attention score $e_i = \mathbf{q} \cdot \mathbf{k}_i / \sqrt{d_h}$, and its value vector by a scalar v_i. The six tokens are:

Token	T_1	T_2	T_3	T_4	T_5	T_6
Score e_i	2	4	1	0	1	2
Value v_i	10	30	5	2	8	12

For streaming paged attention, the state is initialised to $m = -\infty$, $s = 0$, $o = 0$ before any page is processed. After each page i, the three-scalar state is updated by Equations (4.18)–(4.20). Table 4.2 traces both standard attention and streaming

paged attention token by token. Both methods arrive at the same output ≈ 24.2, confirming that streaming paged attention is mathematically identical to standard full-context attention. At no point during the page loop was more than one page resident on GPU; the only state carried between pages was the three-scalar tuple (m, s, o).

The combined result is that context length scales with CPU RAM, not VRAM: a 16 GB host supports over 30,000 tokens of context on a GPU that would otherwise out-of-memory at 500 tokens—a $60\times$ extension with no change to the partition plan or compiled graphs.

Table 4.2: Standard attention vs. streaming paged attention on the six-token example. Both methods produce the same output ≈ 24.2.

Token	**Standard Attention** (accumulate all scores, then normalise)	**Streaming Paged Attention** (update (m, s, o) per token, Eqs. (4.18)–(4.20))
T_1 e=2, v=10	scores $\leftarrow$ [2]	$m \leftarrow \max(-\infty, 2) = 2$ $s \leftarrow 0 + e^{2-2} = 1.000$ $o \leftarrow 0 + e^{2-2} \cdot 10 = 10.00$
T_2 e=4, v=30	scores $\leftarrow$ [2, 4]	$m \leftarrow \max(2, 4) = 4$ (m **grew**) rescale: $\alpha = e^{2-4} = 0.135$ $s \leftarrow 0.135 \cdot 1.000 + e^{4-4} = 1.135$ $o \leftarrow 0.135 \cdot 10.00 + e^{4-4} \cdot 30 = 31.35$
T_3 e=1, v=5	scores $\leftarrow$ [2, 4, 1]	$m \leftarrow \max(4, 1) = 4$ (m unchanged) $s \leftarrow 1.135 + e^{1-4} = 1.185$ $o \leftarrow 31.35 + e^{1-4} \cdot 5 = 31.60$
T_4 e=0, v=2	scores $\leftarrow$ [2, 4, 1, 0]	$m \leftarrow 4$ (m unchanged) $s \leftarrow 1.185 + e^{0-4} = 1.203$ $o \leftarrow 31.60 + e^{0-4} \cdot 2 = 31.64$
T_5 e=1, v=8	scores $\leftarrow$ [2, 4, 1, 0, 1]	$m \leftarrow 4$ (m unchanged) $s \leftarrow 1.203 + e^{1-4} = 1.253$ $o \leftarrow 31.64 + e^{1-4} \cdot 8 = 32.04$
T_6 e=2, v=12	scores $\leftarrow$ [2, 4, 1, 0, 1, 2]	$m \leftarrow 4$ (m unchanged) $s \leftarrow 1.253 + e^{2-4} = 1.388$ $o \leftarrow 32.04 + e^{2-4} \cdot 12 = 33.66$
Output	Global max = 4; weights = [0.135, 1, 0.050, 0.018, 0.050, 0.135]; $\sum w = 1.388$; $\mathbf{o}_{\text{out}} = 33.66/1.388 \approx \mathbf{24.2}$	$\mathbf{o}_{\text{out}} = o_6 \,/\, s_6 = 33.66/1.388 \approx \mathbf{24.2}$ ✓ At no step was more than one KV page resident on GPU.

Chapter 5

EXPERIMENTS

The experimental evaluation is designed to answer four questions:

1. Can HELM run models that exceed GPU VRAM when all other backends fail?
2. Does HELM provide higher decode throughput than feasible baselines on memory-constrained hardware?
3. How far does HELM extend the maximum supported context length?
4. Do HELM's individual design choices—the AVX2 kernel, KV offload, and partition strategy—each contribute meaningfully to performance?

5.1 Hardware Configurations

Three systems are evaluated to cover the range of consumer and prosumer GPU memory budgets where CPU offloading is relevant:

- **4060**: RTX 4060 Laptop GPU (8 GB VRAM, 256 GB/s) + Intel Core i7-13700H, 16 CPU cores, 16 GB DDR5 RAM. The most memory-constrained setting: no baseline other than Accelerate can load any model above 4B parameters here.
- **3090**: NVIDIA RTX 3090 (24 GB VRAM, 936 GB/s) + AMD EPYC 7H12, 128 CPU cores, 125 GB DDR4 RAM. A mid-range server where smaller models fit in VRAM but larger ones do not.

- **L40S**: NVIDIA L40S (48 GB VRAM, 864 GB/s) + AMD EPYC 7413, 24 cores, 60 GB DDR4 RAM. A high-end GPU where most models fit in VRAM entirely, allowing verification that HELM introduces negligible overhead when no CPU offloading is needed.

5.2 Models

Four models are tested spanning memory footprints that progressively exceed the VRAM budgets of the test systems: Qwen3-4B (8 GB), Qwen3-8B (16 GB), Qwen3-14B (28 GB), and Qwen3-32B (64 GB) Qwen Team (2025). Qwen3 is chosen because it uses a standard dense transformer with SwiGLU activations and Grouped-Query Attention, making it representative of mainstream open-weight models. Its four sizes provide configurations where the model fits entirely in VRAM alongside configurations where it does not, covering both HELM's target regime and the baseline-equivalent case.

5.3 Baselines

HELM is compared against three backends that represent the current state of practice for CPU offloading and memory-constrained inference:

- **vLLM** Kwon *et al.* (2023) v0.17.1, PagedAttention, fp16, default GPU memory utilisation (0.90).
- **Accelerate** Gugger *et al.* (2022) `device_map='auto'`, fp16.
- **DeepSpeed** Rajbhandari *et al.* (2020) ZeRO-Inference, fp16, CPU offload enabled.

All baselines run with default settings. llama.cpp is not included because it requires INT4/INT8 quantisation and does not support fp16 execution, making it incomparable to the fp16 setting used by all other backends. HELM runs with auto partition, KV offload enabled, AVX2 kernel enabled, and fp16 precision across all experiments.

5.4 Metrics and Methodology

All experiments use a fixed input prompt of 128 tokens drawn from a generic instruction-following template. The median (p50) over 10 independent requests is reported. The median is used rather than the mean because the first request in each run includes one-time overhead such as KV allocator initialisation and CUDA graph warm-up that inflates the mean.

The primary metric is *decode throughput* (output tokens per second, single request, batch=1), as it is the binding bottleneck for interactive use on commodity hardware. Time-to-first-token (TTFT) is also reported, maximum supported context length, and per-token latency distributions for memory-constrained configurations. Ablation studies isolate the contribution of each design component across both constrained and unconstrained configurations.

Chapter 6

RESULTS AND EVALUATION

The results are organised around four claims that correspond directly to the research questions posed in Chapter 5.

6.1 Claim 1: HELM is the Only Backend That Sustains Inference Across All Memory-Constrained Configurations

Table 6.1 records feasibility directly across all 12 model and hardware combinations. **OOM** indicates failure at load time and × indicates the backend failed to initialise on the platform entirely.

On the RTX 4060 (8 GB), vLLM and DeepSpeed fail to initialise on every model tested, producing no usable output. Accelerate loads only Qwen3-4B; HELM loads both Qwen3-4B and Qwen3-8B by partitioning layers across CPU and GPU at compile time. On the RTX 3090 (24 GB), vLLM and DeepSpeed load the two smaller models but OOM on Qwen3-14B and Qwen3-32B because both require all weights to reside in VRAM simultaneously. Accelerate loads Qwen3-14B but not 32B. HELM loads and runs all four models on the 3090 via CPU+GPU partitioning. On the L40S (48 GB), HELM is the only backend to run Qwen3-32B, doing so via a cpu(18)→cuda(48) partition at 1.1 tok/s with a 9.1 s TTFT; DeepSpeed and vLLM fail to initialise on 32B entirely. Across all three systems, HELM is the only backend that never produces an OOM failure on any configuration it attempts to run.

Table 6.1: Decode throughput (tok/s) and TTFT p50 (ms) at output length 128, batch size 1. Bold denotes the best result per model per GPU. OOM: backend fails to load the model. ×: backend failed to initialise. NR: metric not reported by backend.

		HELM		**Accelerate**		**DeepSpeed**		**vLLM**	
GPU	**Model**	tok/s	TTFT	tok/s	TTFT	tok/s	TTFT	tok/s	TTFT
4060 (8 GB) 16 GB RAM	Qwen3-4B	**17.2**	233	6.7	156	×	×	×	×
	Qwen3-8B	**4.1**	2,505	0.8	1,216	×	×	×	×
	Qwen3-14B	×	×	×	×	×	×	×	×
	Qwen3-32B	×	×	×	×	×	×	×	×
3090 (24 GB) 125 GB RAM	Qwen3-4B	27.1	50	26.6	41	24.3	44	**86.0**	NR
	Qwen3-8B	22.9	49	23.3	43	23.8	43	**50.6**	NR
	Qwen3-14B	**6.1**	870	1.3	780	×	×	OOM	OOM
	Qwen3-32B	**1.2**	8,146	×	×	×	×	OOM	OOM
L40S (48 GB) 60 GB RAM	Qwen3-4B	27.1	81	23.9	45	25.8	45	**79.4**	NR
	Qwen3-8B	25.1	83	23.3	46	24.3	46	**46.7**	NR
	Qwen3-14B	20.7	90	21.2	50	20.3	53	**26.3**	NR
	Qwen3-32B	**1.1**	9,147	0.5	2,092	×	×	×	×

6.2 Claim 2: HELM Significantly Extends Usable Context by Offloading KV Pages to Host RAM

All four Qwen3 models support a native context window of 32,768 tokens. Whether a backend can approach that limit depends entirely on how it manages the KV cache as the sequence grows. Backends that keep all KV entries in GPU VRAM exhaust the memory budget left after weights are loaded, producing an OOM error well short of 32K. HELM offloads KV pages to host RAM, shifting the bottleneck from GPU VRAM to the much larger host memory pool. On well-resourced systems this is enough to reach the full 32K native limit; on more constrained hardware HELM still extends usable context substantially beyond what any GPU-only backend achieves, with the actual ceiling set by available host RAM rather than GPU VRAM.

Figure 6.1 shows the maximum output length each backend sustains before hitting an OOM error. The y-axis is on a $\log_2$ scale; OOM entries are plotted at the axis

floor to make failures visible.

On the RTX 3090 (panel b), HELM reaches the full 32K native limit on Qwen3-4B, Qwen3-8B, and Qwen3-14B. vLLM caps at 8K on the two smaller models despite having 24 GB of VRAM, because its GPU-resident KV allocator exhausts the remaining memory after weights are loaded. Accelerate caps at 4K on 4B and 8B, and at only 128 tokens on 14B. Neither vLLM nor DeepSpeed can load 14B or 32B at all. HELM reaches 4K on Qwen3-32B, the limit imposed by the 125 GB of host RAM available after weights are partitioned.

On the L40S (panel c), models up to 14B fit entirely in VRAM. Here HELM and vLLM both reach 32K on 4B, 8B, and 14B, confirming that HELM does not impose overhead when no offloading is needed. For Qwen3-32B, HELM reaches 8K via KV offload while DeepSpeed OOMs and vLLM fails to initialise.

On the RTX 4060 (panel a), HELM is one of only two backends that can generate any output at all. Accelerate reaches 1K on Qwen3-4B and 128 tokens on Qwen3-8B before exhausting the 8 GB VRAM. HELM sustains 8K on Qwen3-4B and 8K on Qwen3-8B by paging KV entries to the 16 GB of host DDR5. DeepSpeed and vLLM produce no output on either model.

The consistent pattern across all three systems is that GPU-only KV management leaves the model's native context window largely unreachable on consumer hardware, while HELM's paged KV offload closes that gap.

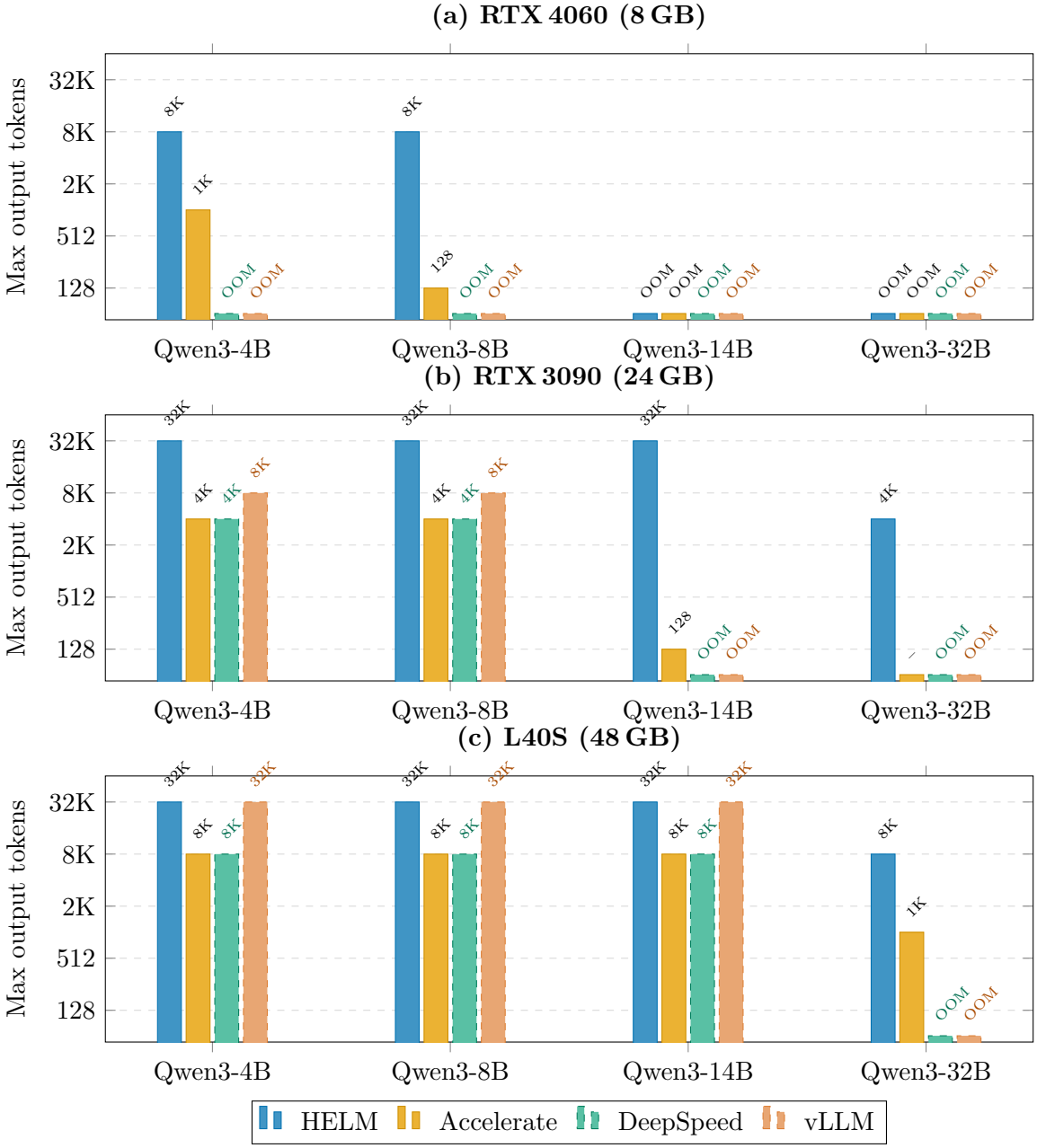

Figure 6.1: Maximum supported output length per backend ($\log_2$ scale). **(a)** RTX 4060 (8 GB): DeepSpeed and vLLM fail to load any model; HELM's paged KV offload extends context **8**× beyond Accelerate on Qwen3-4B (8K vs. 1K) and **64**× on Qwen3-8B (8K vs. 128). **(b)** RTX 3090 (24 GB): HELM is the only backend to run Qwen3-14B and 32B via CPU+GPU partitioning. **(c)** L40S (48 GB): HELM matches vLLM on 4B/8B and extends to 32K for 32B via CPU+GPU partitioning.

6.3 Claim 3: HELM Achieves up to 5.1× Higher Decode Throughput over Feasible Baselines

Table 6.1 reports decode throughput (tok/s) and TTFT p50 (ms) at output length 128, batch=1, for all model and hardware combinations. Figure 6.2 shows the empirical CDF of per-token latency for the four memory-constrained configurations where the model does not fit entirely in GPU VRAM.

On the RTX 4060, HELM is the only backend besides Accelerate that can run any model. For Qwen3-4B, HELM achieves 17.2 tok/s vs. Accelerate's 6.7 tok/s, a **2.6**× improvement; for Qwen3-8B, HELM achieves 4.1 tok/s vs. Accelerate's 0.8 tok/s, a **5.1**× improvement. Panel (a) and (b) of Figure 6.2 confirm these gains are consistent across all requests: p50 latency is **2.4**× lower on Qwen3-4B (60.7 ms vs. 149 ms) and **4.9**× lower on Qwen3-8B (248 ms vs. 1,209 ms).

On the RTX 3090, Qwen3-14B (28 GB) exceeds the 24 GB VRAM budget. HELM achieves 6.1 tok/s vs. Accelerate's 1.3 tok/s, a **4.7**× improvement, with p50 latency of 171 ms vs. 783 ms (**4.6**× lower), shown in panel (c). vLLM and DeepSpeed OOM entirely on this configuration.

On the L40S with Qwen3-32B (64 GB model, 48 GB VRAM), Accelerate is the only other feasible baseline. HELM achieves 1.1 tok/s vs. Accelerate's 0.5 tok/s, a **2.2**× improvement, with p50 latency of 909 ms vs. 2,060 ms (**2.3**× lower), shown in panel (d).

Averaged across the four memory-constrained configurations, HELM delivers a **3.6**× mean throughput improvement over the strongest feasible baseline. On configurations where all weights fit in VRAM, HELM performs within 5% of Accelerate and DeepSpeed, confirming that the compilation and execution overhead is negligible

when no CPU offloading is required.

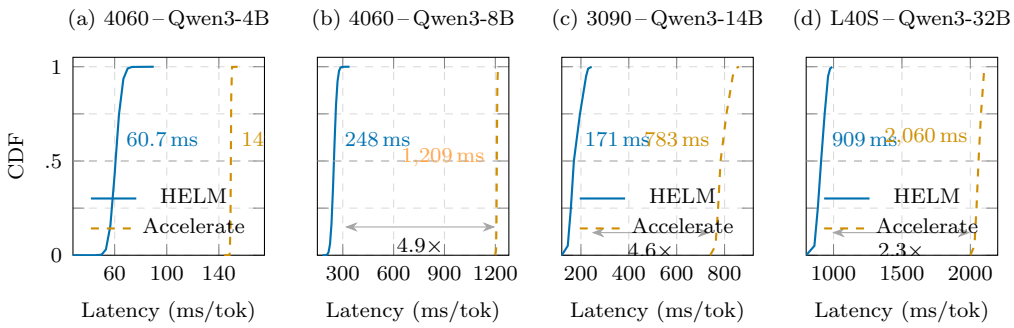

Figure 6.2: Empirical CDF of per-token decode latency (output length 128, batch=1, 10 requests). Only memory-constrained configurations shown. **(a)** RTX 4060/Qwen3-4B: HELM is **2.4**× faster. **(b)** RTX 4060/Qwen3-8B: HELM is **4.9**× faster. **(c)** RTX 3090/Qwen3-14B: HELM is **4.6**× faster; DeepSpeed and vLLM OOM. **(d)** L40S/Qwen3-32B: HELM is **2.3**× faster; DeepSpeed OOMs; vLLM fails to load.

6.4 Claim 4: Each Design Choice Contributes Independently to Performance

Each of HELM's design choices interacts with the partition plan assigned by the cost model. The following ablations isolate the contribution of each component individually, followed by cost model accuracy and output quality validation. In each ablation, only the component under study is toggled; all others remain at their default (enabled) state. Throughput values in ablation tables therefore reflect single-component isolation and may differ from Table 6.1, where all components are active simultaneously.

6.4.1 Batch Size

Table 6.2 shows aggregate decode throughput as batch size increases. On the RTX 3090 where Qwen3-4B and 8B fit entirely in VRAM, throughput scales **3.1**× at batch=8, closely matching the roofline prediction of near-linear scaling until DRAM

bandwidth saturates. On the RTX 4060 and for larger models on the 3090, batch>1 triggers OOM because the GPU holds the maximum number of layers its VRAM budget allows, leaving no headroom for a second request's KV cache.

Table 6.2: Effect of batch size on aggregate decode throughput (tok/s), output length 128.

GPU	Model	B=1	B=2	B=4	B=8
3090 (24 GB) 125 GB RAM	Qwen3-4B	21.1	34.0	53.6	**63.5**
	Qwen3-8B	22.6	34.1	55.4	**63.0**
	Qwen3-14B	6.1	OOM	OOM	OOM
	Qwen3-32B	1.2	OOM	OOM	OOM
4060 (8 GB) 16 GB RAM	Qwen3-4B	17.2	OOM	OOM	OOM
	Qwen3-8B	4.1	OOM	OOM	OOM
L40S (48 GB) 60 GB RAM	Qwen3-4B	27.1	43.7	68.8	**81.5**
	Qwen3-8B	25.1	37.9	61.5	**70.0**
	Qwen3-14B	20.1	25.1	40.1	**46.6**
	Qwen3-32B	1.1	OOM	OOM	OOM

6.4.2 CPU Kernel (AVX2 vs. PyTorch Fallback)

Table 6.3 shows the effect of disabling the AVX2+F16C kernel. The impact scales with the size of the CPU stage: on the RTX 3090, Qwen3-4B and 8B see moderate drops of −24.0% and −9.6% respectively, while the larger CPU stage for Qwen3-14B yields a −17.9% drop. On the RTX 4060 with Qwen3-4B (4 CPU layers) the drop is −57.0%, and −87.8% for Qwen3-8B (21 CPU layers): with 21 of 36 transformer blocks on CPU, disabling AVX2 raises per-token decode latency from 258 ms to over 2,087 ms, making the CPU stage the overwhelming bottleneck. Qwen3-32B on the 3090 (cpu(43)→cuda(23)) shows the most extreme drop at −91.7%.

Table 6.3: Effect of AVX2+F16C kernel on decode throughput (tok/s), output length 128, batch size 1.

GPU	Model	+AVX2	no AVX2	Δ
3090 (24 GB) 125 GB RAM	Qwen3-4B	27.1	20.6	−24.0%
	Qwen3-8B	22.9	20.7	−9.6%
	Qwen3-14B	6.1	2.3	−62.3%
	Qwen3-32B	1.2	0.1	−91.7%
4060 (8 GB) 16 GB RAM	Qwen3-4B	17.2	7.4	−57.0%
	Qwen3-8B	4.1	0.5	−87.8%
L40S (48 GB) 60 GB RAM	Qwen3-4B	27.1	27.0	−0.4%
	Qwen3-8B	25.1	25.1	+0.0%
	Qwen3-14B	20.7	20.1	−2.9%
	Qwen3-32B	1.1	0.4	−63.6%

6.4.3 KV Offload

Table 6.4 shows the effect of disabling KV offload. On the RTX 3090 with Qwen3-4B and 8B, disabling KV offload reduces throughput by −61.6% and −56.3% respectively, because HELM's streaming paged-attention path is replaced by a standard per-step HuggingFace `DynamicCache`, which lacks the pipeline overlap that hides PCIe transfer latency. For Qwen3-14B on the 3090 (cpu(8)→cuda(34)), disabling KV offload yields −14.6%. On the L40S (48 GB), where all three models fit entirely on-GPU, throughput drops by approximately 54% across all models.

Table 6.4: Effect of KV offload on decode throughput (tok/s), output length 128, batch size 1.

GPU	Model	+KV offload	no KV offload	Δ
3090 (24 GB) 125 GB RAM	Qwen3-4B	27.1	10.4	−61.6%
	Qwen3-8B	22.9	10.0	−56.3%
	Qwen3-14B	6.1	5.21	−14.6%
4060 (8 GB) 16 GB RAM	Qwen3-4B	17.2	12.1	−29.7%
	Qwen3-8B	4.1	2.9	−29.3%
L40S (48 GB) 60 GB RAM	Qwen3-4B	27.1	12.5	−54.0%
	Qwen3-8B	25.1	11.8	−53.0%
	Qwen3-14B	20.7	9.4	−54.6%

CPU thread count has a negligible effect when the partition is all-GPU, since the CPU stage is near-idle. For configurations with a large CPU stage, throughput scales with thread count until DRAM bandwidth saturates, confirming that the CPU stage is memory-bandwidth-bound rather than compute-bound.

6.4.4 Cost Model Accuracy

The roofline cost model does not account for context-independent per-layer overhead: Python dispatch, CUDA kernel launch latency, residual additions, normalisation, and Rotary Position Embedding (RoPE). This overhead is calibrated per-GPU as a fixed $\bar{\delta}$ per block from the all-GPU configurations, giving $\bar{\delta}_{3090} \approx 0.79\,\text{ms}$, $\bar{\delta}_{4060} \approx 0.14\,\text{ms}$, and $\bar{\delta}_{\text{L40S}} \approx 1.64\,\text{ms}$. Table 6.5 shows that the overhead-corrected prediction $\hat{T} = T_{\text{pred}} + N \cdot \bar{\delta}$ matches measured latency within 8% across all configurations.

Table 6.5: Cost model accuracy at batch size 1. T_{pred}: roofline estimate. $\hat{T}$: overhead-corrected prediction.

GPU	Model	Plan	Pred. (ms)	Meas. (ms)	Err.
3090 (24 GB) 125 GB RAM	Qwen3-4B	cuda(38)	48.2	50.5	+5%
	Qwen3-8B	cuda(38)	51.0	48.7	−4%
	Qwen3-14B	cpu(8)→cuda(34)	143.5	134.6	+6.6%
4060 (8 GB) 16 GB RAM	Qwen3-4B	cpu(4)→cuda(34)	80.1	80.1	+0%
	Qwen3-8B	cpu(21)→cuda(17)	263.5	244.3	−7%
L40S (48 GB) 60 GB RAM	Qwen3-4B	cuda(38)	36.9	36.9	+0%
	Qwen3-8B	cuda(38)	39.9	39.8	+0%
	Qwen3-14B	cuda(42)	46.3	49.8	+8%
	Qwen3-32B	cpu(18)→cuda(48)	908	909	+0%

Chapter 7

CONCLUSION

This thesis presented HELM, a compiler and runtime for heterogeneous LLM inference on memory-constrained CPU+GPU systems. The core insight is that CPU-GPU inference is a compile-time optimisation problem: by searching exhaustively over all feasible layer-to-device assignments before the first token is generated, HELM finds the globally optimal split rather than making greedy per-layer decisions at load time or streaming weights on every forward pass.

In the evaluation across three GPUs and four models ranging from 4B to 32B parameters, HELM is the only backend that loads and runs every feasible configuration without an out-of-memory failure. On the configurations where a baseline is feasible, HELM achieves up to **5.1**× higher decode throughput and up to **4.9**× lower per-token latency, with a mean speedup of **3.6**× over the strongest feasible baseline across all memory-constrained pairs. HELM enables models to use far more context than GPU VRAM alone permits, reaching the full native 32K window where host RAM allows and substantially extending context even on the most memory-constrained systems, where prior CPU-offload systems exhaust memory at as few as 128 tokens. On hardware where all weights fit in VRAM, HELM matches Accelerate and DeepSpeed within 5%, confirming that the compilation overhead is negligible when no CPU offloading is required.

These results show that the memory wall is not a hard barrier for LLM inference on consumer hardware. A principled compile-time approach, grounded in hardware mea-

surement rather than heuristics, can sustain efficient inference across a broad range of configurations without model quantization, multi-GPU infrastructure, or cloud resources. As CPU DRAM bandwidth and CPU-GPU interconnect bandwidth continue to improve, the compile-time partition approach scales automatically: HELM's profiler re-measures the hardware at startup and the cost model reflects the new capabilities in the partition plan without any manual tuning.

7.1 Limitations and Future Work

The DeviceProfiler estimates CPU DRAM bandwidth from a single contiguous 64 MB tensor read, which is representative for small CPU stages but overestimates effective bandwidth for very large ones (e.g. Qwen3-32B with 43 CPU layers), where Translation Lookaside Buffer (TLB) pressure and non-contiguous parameter allocations reduce sustained throughput to roughly one fifth of the micro-benchmark value. A future profiler will read non-contiguous buffers matching the actual inference access pattern.

Beyond this, four directions remain for future work:

1. **Multi-device topologies.** Extending the partition search to multi-GPU and heterogeneous multi-device topologies, enabling HELM to exploit additional heterogeneous hardware configurations beyond single-node CPU+GPU.

2. **Quantised execution.** Supporting quantised execution paths (INT4/INT8 via bitsandbytes, AWQ, or GGUF), whose custom kernels currently break FX symbolic tracing.

3. **Mixture-of-Experts (MoE) models.** Extending the partition search to sparse MoE architectures, where only a subset of experts is activated per token

and expert routing introduces dynamic sparsity that the current cost model does not account for.

4. **Vision-language models (VLMs).** Supporting models with a vision encoder prepended to the language backbone, where the image encoding stage has a different compute and memory profile than the text decode stage and may warrant a separate partition plan.

REFERENCES

Ainslie, J., J. Lee-Thorp, M. de Jong, Y. Zeiler, S. Sanghai and Y. Xu, "GQA: Training generalized multi-query transformer models from multi-head checkpoints", https://arxiv.org/abs/2305.13245 (2023).

Gao, L. *et al.*, "A framework for few-shot language model evaluation", https://github.com/EleutherAI/lm-evaluation-harness (2021).

Gerganov, G., "llama.cpp", https://github.com/ggerganov/llama.cpp (2023).

Gugger, S., L. Debut, T. Wolf, P. Schmid, Z. Mueller, S. Mangrulkar, M. Sun and B. Bossan, "Accelerate: Training and inference at scale made simple, efficient and adaptable", https://github.com/huggingface/accelerate (2022).

He, J. and J. Zhai, "FastDecode: High-throughput GPU-efficient LLM serving using heterogeneous pipelines", https://arxiv.org/abs/2403.11421 (2024).

Jiang, X., Y. Zhou, S. Cao, I. Stoica and M. Yu, "NEO: Saving GPU memory crisis with CPU offloading for online LLM inference", https://arxiv.org/abs/2411.01142 (2024).

Kwon, W., Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang and I. Stoica, "Efficient memory management for large language model serving with PagedAttention", in "Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles", (2023).

Liu, Y. *et al.*, "LMCache: An efficient KV cache layer for enterprise-scale LLM inference", https://arxiv.org/abs/2510.09665 (2024).

LMStudio, "LM Studio", Software. https://lmstudio.ai (2024).

Lu, K., Q. Wei, Y. Lin, P. Liu, H. Wang, J. Wan, T. Yao, H. Wu and D. Wang, "Q-Infer: Towards efficient GPU-CPU collaborative LLM inference via sparsity-aware dynamic scheduling", ACM Transactions on Architecture and Code Optimization (2025).

Luo, C., Z. Cai, H. Sun, J. Xiao, B. Yuan, W. Xiao, J. Hu, J. Zhao, B. Chen and A. Anandkumar, "HeadInfer: Memory-efficient LLM inference by head-wise offloading", https://arxiv.org/abs/2502.12574 (2025).

Ollama, "Ollama", Software. https://ollama.com (2024).

Openclaw, "OpenClaw", Software. https://github.com/openclaw (2024).

Qwen Team, "Qwen3 technical report", https://arxiv.org/abs/2505.09388 (2025).

Radford, A., J. Wu, R. Child, D. Luan, D. Amodei and I. Sutskever, "Language models are unsupervised multitask learners", OpenAI Blog **1**, 8 (2019).

Rajbhandari, S., J. Rasley, O. Ruwase and Y. He, "DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters", Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (2020).

Sheng, Y., L. Zheng, B. Yuan, Z. Li, M. Ryabinin, D. Y. Fu, Z. Ng, A. Mall, Y. Li, I. Stoica, C. Re and C. Zhang, "FlexGen: High-throughput generative inference of large language models with a single GPU", `https://arxiv.org/abs/2303.06865` (2023).

Song, Y., Z. Mi, H. Xie and H. Chen, "PowerInfer: Fast large language model serving with a consumer-grade GPU", `https://arxiv.org/abs/2312.12456` (2023).

Srivastava, G., S. Cao and X. Wang, "Towards reasoning ability of small language models", `https://arxiv.org/abs/2502.11569` (2025).

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, "Attention is all you need", Advances in Neural Information Processing Systems **30** (2017).

Xiong, Y., H. Wu, C. Shao, Z. Wang, R. Zhang, Y. Guo, J. Zhao, K. Zhang and Z. Pan, "LayerKV: Optimizing large language model serving with layer-wise KV cache management", `https://arxiv.org/abs/2410.00428` (2024).

Yu, C., T. Wang, Z. Shao, L. Zhu, X. Zhou and S. Jiang, "TwinPilots: A new computing paradigm for GPU-CPU parallel LLM inference", in "Proceedings of the 17th ACM International Systems and Storage Conference", pp. 91–103 (2024).

Zhao, X., B. Jia, H. Zhou, Z. Liu, S. Cheng and Y. You, "HeteGen: Heterogeneous parallel inference for large language models on resource-constrained devices", `https://arxiv.org/abs/2403.01164` (2024).

Zheng, Y., Y. Chen, B. Qian, X. Shi, Y. Shu and J. Chen, "A review on edge large language models: Design, execution, and applications", `https://arxiv.org/abs/2410.11845` (2024).