Title

Energy Modeling for LLM Inference without Per-Workload Profiling

ISBN

9798190919998

Author

Yao, Emily

Publication Date

2026-09-04

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Energy Modeling for LLM Inference without Per-Workload Profiling

A thesis submitted in partial satisfaction of the requirements for the degree of

Master of Science in Electrical and Computer Engineering

by

Emily Yao

2026

ABSTRACT OF THE THESIS

Energy Modeling for LLM Inference without Per-Workload Profiling

by

Emily Yao

Master of Science in Electrical and Computer Engineering

University of California, Los Angeles, 2026

Professor Puneet Gupta, Chair

Predicting energy for graphics processing units before a large language model workload is deployed requires both its active duration and the activity generated while it runs. This work develops an interpretable energy model for inference on NVIDIA A100 graphics processing units. It separates static energy, which is proportional to active time, from dynamic energy, whose compute term scales with issued floating-point operations and whose memory terms scale with bytes transferred at shared memory, the level-two cache, and high-bandwidth memory. A memory-level count denotes bytes crossing that level's interface with the level beneath it; loads and stores therefore use the same per-byte coefficient. Published microbenchmarks of graphics processors supply compute, memory, latency, bandwidth, and event-energy parameters, limiting local profiling to idle static power and one fixed timing overhead rather than fitting workload-specific energy curves.

For inference distributed across graphics processing units, this work additionally contributes a system-wide network model whose energy scales with collective active time, transmitted bytes, and reduction-memory traffic. The energy equations depend on an execution-aware Roofline analysis that predicts kernel active time and activity counters from thread-block geometry, resource concurrency, full and tail waves, pipeline fill and drain, and explicit memory traffic. RAPID-LLM composes these kernel profiles into large language model inference graphs and tracks the scheduled collective communication activity. On held-out A100 data, the model achieves 11.41% general

matrix multiplication energy mean absolute percentage error. End-to-end single-device energy mean absolute percentage error is 13.5% for Qwen workloads and 17.3% for Llama decode; two- and eight-device Qwen 2.5 72B experiments obtain 14–20% system-energy mean absolute percentage error. Independent counter validation shows close agreement for high-bandwidth-memory and level-two-cache traffic, supporting energy attribution to explicit compute, memory, and static terms.

The thesis of Emily Yao is approved.

Nader Sehatbakhsh

Blaise Tine

Puneet Gupta, Committee Chair

University of California, Los Angeles

2026

Contents

List of Figures

List of Tables

Introduction

Large language model (LLM) inference energy depends on more than parameter count. It changes with batch size, prompt and generation length, precision, kernel implementation, GPU generation, and the chosen parallel mapping. These choices must often be evaluated before the target workload is available to run at scale. Inference energy accumulates across repeated production queries, making this design-time question especially important for serving [24].

Runtime measurement, profiling, and carbon-accounting tools report telemetry from executed workloads [10, 38]. Using them therefore requires access to the workload and target hardware. GPU power interfaces such as the NVIDIA Management Library (NVML) return aggregate device and associated-circuitry power rather than the contributions of arithmetic, memory hierarchy, or interconnect activity [37]. Detailed GPU simulation offers more attribution, but trace-driven frameworks can require native-instruction traces and substantial simulation time [15].

This work develops an analytical GPU energy model for this design space. Starting from the Roofline view of compute and data movement [48], the model makes the execution structure that a conventional Roofline hides explicit: CTA and warp geometry, Tensor Core and vector-engine service, shared-memory and L2 traffic, concurrency limits, full and tail waves, and prologue and epilogue costs. It predicts latency and activity only as inputs to event-based energy prediction for structured GPU operations, then composes the resulting local energy profiles with communication dependencies for workload-level prediction. Static energy scales with kernel active time; dynamic compute energy scales with issued floating-point operations; and dynamic memory energy scales with traffic at each modeled memory level.

The guiding principle is first-principles, shape-dependent prediction: an operation's dimensions and selected kernel configuration determine the work, traffic, and execution constraints used by the model, rather than serving only as keys for extrapolating a database of profiled workloads.

This work complements measurement with an interpretable model whose calibration boundary

is explicit. Calibration is limited to six explicitly scoped terms: fixed kernel overhead, idle static power, effective network utilization, fixed network-transfer latency, per-GPU network active power, and energy per transmitted network byte. All other hardware and model parameters are documented values with explicit provenance, rather than fitted inputs. Calibration rows are kept separate from held-out validation. This separation evaluates predictions on kernels held out from calibration and makes each residual attributable to a modeled or unmodeled resource.

The paper makes three contributions:

- A GPU-local energy model comprising static, issued-compute, and per-level memory-traffic energy, driven by execution-aware latency and activity predictions.
- A calibration boundary that pairs limited device- and system-specific fitting with documented architectural specifications and held-out validation.
- A system-wide network energy model that RAPID-LLM composes with local operation profiles and scheduled communication for tensor-parallel workload prediction.

The GPU energy model is described in Chapter 2, and the execution-aware provider of its time and activity inputs appears in Chapter 3. RAPID-LLM composition is described in Chapter 4; energy-model inputs and calibration are presented in Chapter 5, followed by held-out results in Chapter 6.

Chapter 1 Background

1.1 LLM Workload Decomposition

A decoder-only Transformer begins with token embedding, repeats Transformer layers, and ends with final normalization and a language-model head [43]. Each repeated layer contains normalization, query–key–value (QKV) projections, attention, output projection, feed-forward projections, residual paths, and elementwise operations [46]. Normalization is therefore a distinct layer operation rather than another name for embedding.

Let L_{in} be the number of prompt tokens supplied before generation and L_{out} the number of requested generated tokens. Prefill processes all L_{in} prompt tokens and generates output token 1. Decode performs the remaining $L_{\mathrm{out}} - 1$ one-token steps while growing the key–value cache [16]. Figure 1.1 shows how the same layer operations acquire different dimensions in these phases.

This decomposition turns a complete request into repeated instances of a small operation set. Each operation executes as one or more CUDA kernels: projections use tiled matrix multiplication [35], while normalization and attention may use reduction or fused kernels [42, 40, 6]. Operation energy is additive across every executed instance.

1.2 Roofline Model

The Roofline model relates useful arithmetic work to the data movement required to perform it [48]. For an operation with F useful floating-point operations and $\mathcal{T}_{\mathrm{HBM}}$ HBM bytes, its arithmetic intensity is $F/\mathcal{T}_{\mathrm{HBM}}$. The standard Roofline estimate compares this intensity with peak compute throughput and memory bandwidth, assigning the operation to the slower compute or memory path.

This estimate is a useful first performance bound, but it presumes that the kernel exposes enough work to sustain the selected peak rate. GPU kernels can fall below that bound when limited

concurrency cannot hide latency, a CTA grid ends in an underfilled tail wave, or instruction scheduling and memory-access behavior prevent efficient execution. Such effects include launch and dispatch overhead, pipeline fill and drain, dependency chains, and poorly coalesced or poorly reused global-memory accesses [3, 4]. The execution model must therefore account for the GPU work hierarchy and the kernel schedule rather than treat peak throughput and bandwidth as immediately available.

The same distinction matters for energy. An estimate based only on ideal Roofline latency, useful FLOPs, and HBM bytes omits time-dependent static energy as well as activity not represented by those logical counts, including work issued by the kernel and traffic at lower memory interfaces. Accurate GPU-local energy consequently requires execution-aware latency and activity predictions [4, 1].

1.3 GPU Execution and CUDA Programming Model

GPU peak rates are aggregate capabilities of many streaming multiprocessors (SMs), rather than the rate of an individual operation. In the Compute Unified Device Architecture (CUDA), host code launches a kernel as a grid of thread blocks [30]. A block, also called a cooperative thread array (CTA), is assigned to one SM for its lifetime and is partitioned into 32-thread warps. The hardware distributes CTAs across SMs, while each SM issues work from its resident warps to distinct vector, Tensor Core, and memory resources [25].

Consequently, a kernel reaches a device-level peak only when its grid and CTA shape provide suitable work for all of those resources. Registers, shared memory, and CTA limits cap the number of CTAs that may reside on an SM. A small grid or a final partial wave can leave SMs without work; within a busy SM, dependencies or a stalled memory access can leave a warp unable to issue. CTA tiling also determines the instruction mix, operand reuse, and memory-access pattern. These choices are specific to the kernel implementation even when two operations have the same FLOP and HBM-byte counts [30, 4].

The CUDA hierarchy thus explains why GPU performance requires more than an arithmetic-

intensity classification: the mapping from an operation to CTAs, warps, and on-chip resources determines the work in flight and the resources activated by the selected kernel. The execution-aware treatment of these effects appears in Section 3.4.

1.4 Memory Hierarchy

GPU memory is a hierarchy of storage and transfer interfaces with very different capacities, bandwidths, and energy costs. Registers retain the operand fragments and accumulators used by an executing warp. Shared memory holds tiles shared by CTAs, while global-memory instructions reach L2 before high-bandwidth memory (HBM), the backing store for data not supplied by L2. An operand therefore may be reused from registers or shared memory many times after it is fetched from a lower level [30, 25].

Logical tensor size does not determine data-movement activity at each level. CTA tiling, warp access patterns, and the amount of reuse determine which interface supplies an access and how many transactions it requires. For example, a tiled matrix multiplication can fetch a tile once from L2 and reuse it from shared memory across several warp tiles; the same tile can be reused from L2 rather than fetched again from HBM [29, 25]. Conversely, accesses that do not combine efficiently into memory transactions consume more interface traffic than their logical byte count suggests [30, 29].

Data-movement accounting must consequently distinguish traffic crossing each interface. At a given level, the relevant bytes are those transferred from the level beneath it, regardless of whether the request originates as a load or a store. This per-level view supplies the activity needed to attribute dynamic memory energy and avoids treating useful FLOPs or HBM bytes alone as a complete description of a kernel's data movement.

1.5 Distributed LLM Execution and RAPID-LLM

Parallelism distributes an LLM workload across devices when the model and its inference state exceed one GPU's memory capacity, or when multiple devices can accelerate execution. A parallelization strategy specifies the work and state held by each device, the communication needed to coordinate them, and the resulting dependencies. It may improve request throughput, single-request latency, memory capacity, or some combination of these goals, but introduces communication and synchronization costs.

Common strategies partition different units of work. Data parallelism stores a complete model replica on each GPU and assigns independent requests or batches to replicas; it improves aggregate throughput but does not divide one request's layer computation. Pipeline parallelism assigns groups of consecutive layers to different GPUs, reducing per-device model storage but introducing stage dependencies [23, 12]. Tensor parallelism instead shards the tensors and matrix operations within each layer across participating GPUs [44]. Each GPU executes a smaller local operation, then devices exchange partial results through collectives such as all-reduce, reduce-scatter, or all-gather [36]. This makes tensor parallelism well suited to serving a model that does not fit on one GPU and to reducing the local work of a latency-sensitive request, provided that collective communication does not dominate the savings [18].

Parallelism is therefore another dimension of workload performance, not merely a device-count choice. A parallel plan changes local tensor dimensions, kernel shapes, CTA grids, and memory traffic while adding communication dependencies. Its system-level tradeoffs can be evaluated credibly only when the resulting device-local operations are predicted faithfully. This work focuses its GPU model on those local operations; a robust local predictor provides the basis for evaluating how alternative parallel partitions affect end-to-end latency and energy.

RAPID-LLM is the distributed-inference analysis framework used by this work [14, 5]. Given the model, request, tensor-parallel configuration, and hardware topology, it decomposes inference into local operations and topology-aware collective operations, preserving their dependencies in a

distributed schedule. This work uses those local operations as inputs to the execution-aware predictor and uses the scheduled collective time, per-link traffic, and reduction-memory activity to obtain GPU-local and system-wide energy.

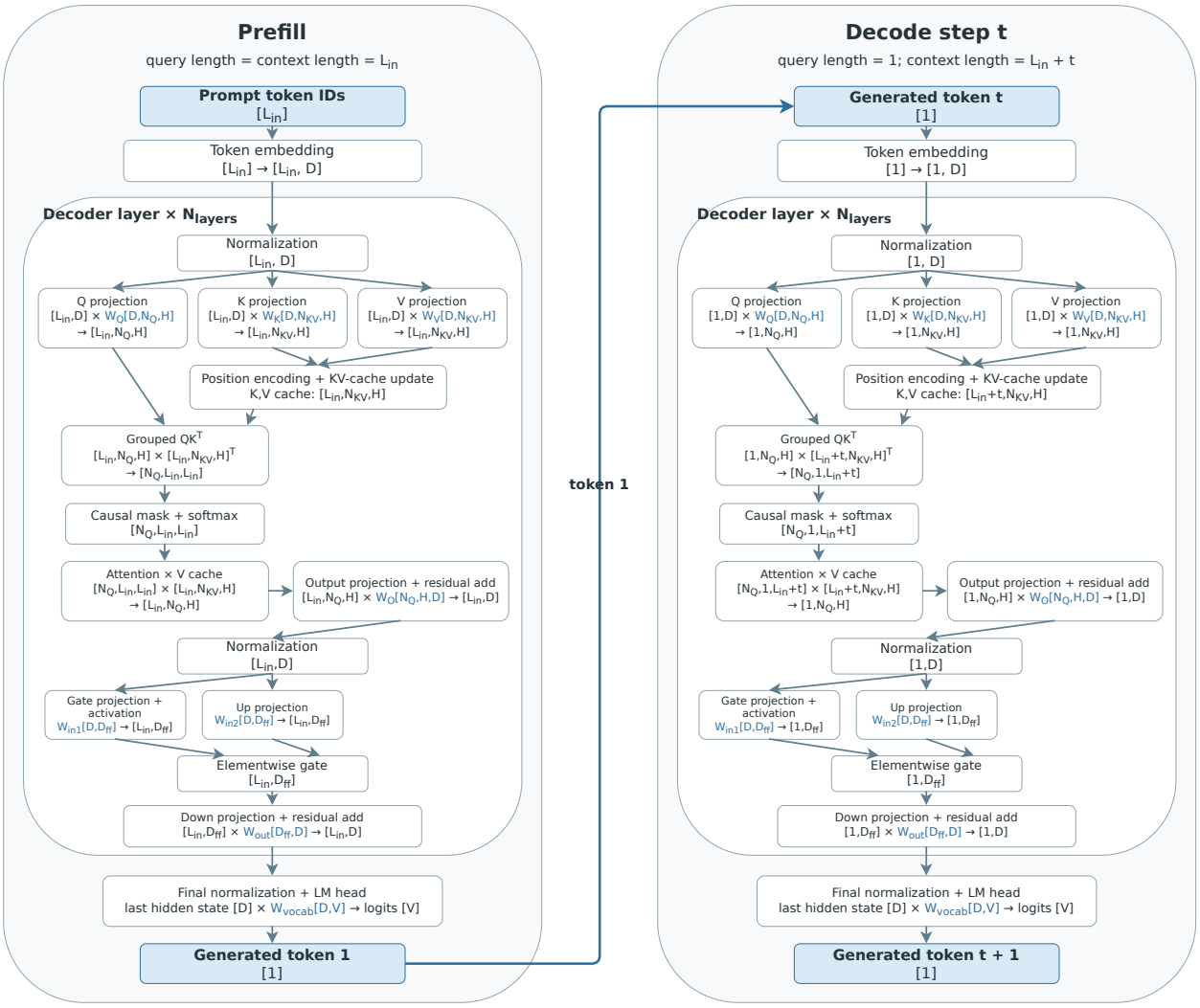

Figure 1.1. Phase-dependent semantic-operation graphs for decoder-only Transformer inference, adapted from Austin et al. [2]. Prefill applies the repeated decoder-layer operations to all L_{in} prompt tokens and produces output token 1, which becomes the one-token input to the first decode step. Decode applies the same operations to one token while attention reads the growing key–value cache. Here D and H are the embedding and attention-head dimensions, D_{ff} is the MLP hidden dimension, N_Q and N_{KV} are the numbers of query and key/value heads, N_{layers} is the number of repeated decoder layers, and V is the vocabulary size.

Chapter 2 GPU Energy Prediction Model

The model separates energy by accounting boundary. Static, compute, and memory energy are GPU-local: each is attributed to the device executing a kernel. Network energy is system-wide because a collective activates multiple GPUs and the links connecting them. Keeping these boundaries explicit prevents device-local memory traffic or collective traffic from being counted twice.

GPU-local energy has static and dynamic components [1]. Static energy accumulates over active time. Dynamic compute energy scales with issued floating-point operations (FLOPs), while dynamic memory energy scales with interface traffic at shared memory, L2, and HBM. The energy model therefore requires both kernel time and the characteristic activity predicted for its implementation. Figure 2.1 summarizes these GPU-local inputs and the separate topology-aware activity used for system-wide communication energy.

2.1 GPU-Local Energy

Let $\mathcal{P}$ denote the set of GPUs participating in the modeled workload, with $n_{\mathrm{GPU}} = |\mathcal{P}|$. For GPU $g \in \mathcal{P}$ executing kernel k, local energy is the sum of a time-dependent static term and activity-dependent dynamic terms:

$$\begin{aligned} E^{(g)}_{k,\mathrm{local}} &= E^{(g)}_{k,\mathrm{static}} + E^{(g)}_{k,\mathrm{dynamic}}, \\ E^{(g)}_{k,\mathrm{dynamic}} &= E^{(g)}_{k,\mathrm{compute}} + E^{(g)}_{k,\mathrm{memory}}. \end{aligned} \tag{2.1}$$

2.1.1 Static Energy. Static energy scales with the time for which a GPU remains active:

$$E^{(g)}_{k,\mathrm{static}} = P^{(g)}_{\mathrm{idle}} t^{(g)}_{k,\mathrm{active}}. \tag{2.2}$$

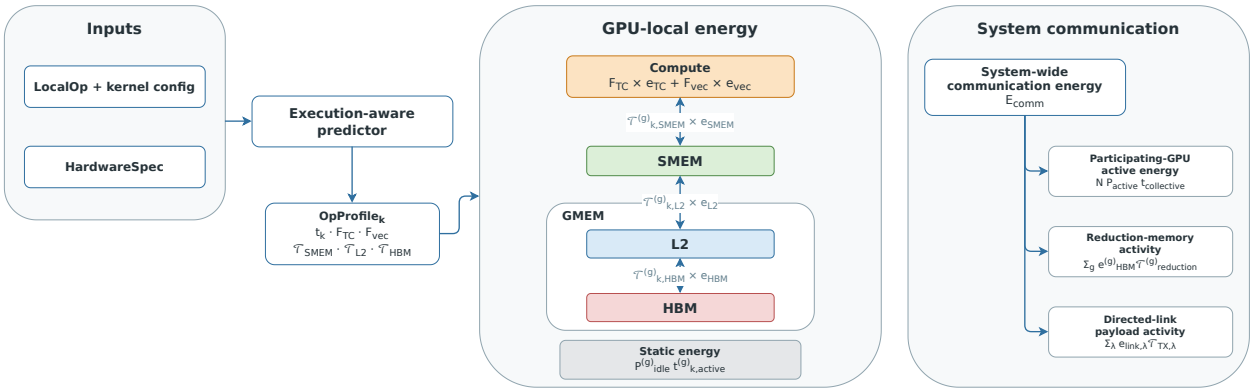

Figure 2.1. Inputs to end-to-end energy from the execution-aware model. The execution-aware predictor returns an `OpProfile` containing kernel time, issued compute work, and per-interface traffic. GPU-local energy applies the corresponding static, compute, and memory coefficients. The separate system-wide communication tree combines participating-GPU active energy, reduction-memory activity, and directed-link payload activity.

$P_{\text{idle}}^{(g)}$ is the measured idle static power charged over active wall time. Active wall time is the elapsed interval from the kernel's start until its completion, including time spent waiting for a modeled dependency. It is not computed by adding resource service times, because compute and data movement can proceed concurrently within a kernel.

2.1.2 Dynamic Compute Energy. Dynamic compute energy scales with arithmetic instructions issued by the selected kernel implementation, rather than only the useful mathematical work:

$$E_{k,\text{compute}}^{(g)} = e_{\text{TC}} F_{k,\text{TC}}^{(g)} + e_{\text{vec}} F_{k,\text{vec}}^{(g)}. \tag{2.3}$$

$F_{k,\text{TC}}^{(g)}$ and $F_{k,\text{vec}}^{(g)}$ are issued Tensor Core and vector floating-point operations (FLOPs). Complete instruction tiles are charged even when padding or boundary lanes do not contribute to useful output. For the evaluated BF16 configurations, the generic vector coefficient is specialized to the documented FP16 measurement:

$$e_{\text{vec}} = e_{\text{FP16}}. \tag{2.4}$$

This makes energy sensitive to kernel selection and tile geometry.

2.1.3 Dynamic Memory Energy. Dynamic memory energy scales independently with bytes accessed at each modeled memory interface. A byte may legitimately be charged at more than one level when it crosses several interfaces, but reuse can make the byte counts differ substantially between levels:

$$E^{(g)}_{k,\mathrm{memory}} = \sum_{\ell \in \{\mathrm{SMEM,L2,HBM}\}} e_\ell \mathcal{T}^{(g)}_{k,\ell}. \tag{2.5}$$

$\mathcal{T}^{(g)}_{k,\ell}$ is absolute byte traffic entering ℓ from underneath:

$$\mathcal{T}^{(g)}_{k,\ell} = \mathcal{T}^{(g)}_{k,\ell,\mathrm{load}} + \mathcal{T}^{(g)}_{k,\ell,\mathrm{store}}, \qquad E^{(g)}_{k,\ell} = e_\ell \mathcal{T}^{(g)}_{k,\ell}. \tag{2.6}$$

Loads and stores use the same e_ℓ because the coefficient represents an interface access rather than a direction-specific operation. Both directions are counted in $\mathcal{T}^{(g)}_{k,\ell}$. "Entering from underneath" fixes the accounting boundary: shared-memory bytes come from the register-side request stream, L2 bytes from requesting SMs, and HBM bytes from the cache/memory interface. The execution-aware predictor supplies these traffic totals.

The local dynamic term is

$$E^{(g)}_{k,\mathrm{dynamic}} = E^{(g)}_{k,\mathrm{compute}} + E^{(g)}_{k,\mathrm{memory}}. \tag{2.7}$$

2.2 System-Wide Network Energy

For a collective over the participating-GPU set $\mathcal{P}$ and directed-link set $\mathcal{E}$, the system-wide communication term combines per-GPU active energy, per-GPU HBM traffic used to reduce partial results, and link-resolved payload traffic:

$$E_{\mathrm{comm}} = n_{\mathrm{GPU}} \left(P_{\mathrm{idle}} + P_{\mathrm{active}}\right) t_{\mathrm{collective}} + \sum_{g \in \mathcal{P}} e^{(g)}_{\mathrm{HBM}} \mathcal{T}^{(g)}_{\mathrm{reduction}} + \sum_{\lambda \in \mathcal{E}} e_{\mathrm{link},\lambda} \mathcal{T}_{\mathrm{TX},\lambda}. \tag{2.8}$$

P_{idle} is the per-GPU idle static power also used for local kernels, while P_{active} is the additional, idle-subtracted collective power per participating GPU. Thus, idle static power continues to accrue while GPUs participate in communication. $t_{\text{collective}}$ is the collective completion time for which every participant is charged. Reduction traffic $\mathcal{T}^{(g)}_{\text{reduction}}$ and its HBM coefficient $e^{(g)}_{\text{HBM}}$ are GPU-local quantities and remain under a device sum because reduction work can be asymmetric. $\mathcal{T}_{\text{TX},\lambda}$ is payload traffic transmitted over directed link λ. A directed link connects a source GPU to a destination GPU without traversing another GPU; its physical path may nevertheless pass through one or more intermediate switches. $e_{\text{link},\lambda}$ is the path's energy per transmitted payload byte in J/B. Switch energy is not modeled as a separate component and is instead absorbed into this link coefficient. The link sum permits different routed traffic and technologies at each topology level.

"System-wide" therefore describes the final accounting boundary, not the granularity of the parameters: active and reduction-memory terms originate on GPUs, while payload energy originates on links. The aggregate is added once. Reduction HBM traffic is excluded from GPU-local kernel memory energy when charged here.

When all participating GPUs share the same HBM energy coefficient e_{HBM} and all directed links share the same transmitted-payload coefficient e_{link}, the hardware is homogeneous for energy accounting. This does not require equal traffic on every GPU or link. Let $\mathcal{T}_{\text{TX}} = \sum_{\lambda\in\mathcal{E}} \mathcal{T}_{\text{TX},\lambda}$; the general form then reduces to

$$E_{\text{comm}} = n_{\text{GPU}} \left(P_{\text{idle}} + P_{\text{active}}\right) t_{\text{collective}} + e_{\text{HBM}} \sum_{g\in\mathcal{P}} \mathcal{T}^{(g)}_{\text{reduction}} + e_{\text{link}} \mathcal{T}_{\text{TX}}. \tag{2.9}$$

Let $\mathcal{K}_g$ be the set of GPU-local operation kernels executed by GPU g in a complete modeled workload, and let $\mathcal{C}$ be the set of collective instances in that workload. Each $E^{(g)}_{k,\text{local}}$ has a single-GPU accounting boundary, whereas each $E_{c,\text{comm}}$ already has the system-wide accounting boundary of Equation (2.8). The total workload energy is therefore

$$E_{\text{workload}} = \sum_{g\in\mathcal{P}} \sum_{k\in\mathcal{K}_g} E^{(g)}_{k,\text{local}} + \sum_{c\in\mathcal{C}} E_{c,\text{comm}}. \tag{2.10}$$

Thus, Equation (2.10) is not the energy of one kernel or one collective. It aggregates every GPU-local kernel and every system-level collective exactly once over the full workload. Latency overlap changes the workload critical path but does not erase either executed activity: an overlapped local kernel remains in the first sum and the collective remains in the second sum. Reduction HBM traffic charged inside $E_{c,\mathrm{comm}}$ is not also included in a GPU-local kernel term.

Chapter 3 Execution-aware inputs to GPU-local energy prediction

3.1 Energy-Facing Prediction Contract

This section provides the execution-dependent inputs required to evaluate the GPU-local kernel-energy equations in Section 2.1. It does not provide the system-level collective terms defined in Equation (2.8). GPU-local energy requires more than logical FLOPs and tensor sizes: static energy requires kernel time, compute energy requires issued rather than merely useful work, and memory energy requires traffic at each accounting interface. For every local kernel k, the execution-aware predictor therefore supplies

$$\texttt{OpProfile}_k = (t_k, F_{k,\mathrm{TC}}, F_{k,\mathrm{vec}}, \mathcal{T}_{k,\mathrm{SMEM}}, \mathcal{T}_{k,\mathrm{L2}}, \mathcal{T}_{k,\mathrm{HBM}})\,. \tag{3.1}$$

Its software interface is

$$\texttt{LocalOp} + \texttt{HardwareSpec} \longrightarrow \texttt{OpProfile}. \tag{3.2}$$

A `LocalOp` supplies the operation, shapes, data types, and a concrete kernel configuration. A `HardwareSpec` supplies clock frequency, residency limits, resource rates and latencies, memory hierarchy, and energy coefficients; Table 5.1 lists the values used for the A100 80GB evaluation. For validation, the measured kernel name and launch dimensions select the implementation directly. Otherwise, the predictor tries Ampere kernel configurations and selects the one with the fastest predicted runtime. CUTLASS name parsing and configuration details appear in Appendix A.

The schedule is used twice. For latency, work and traffic are converted to phase-local service times and composed according to dependencies and legal overlap. For energy, all issued work and per-level traffic are accumulated over the complete kernel, including full and tail waves. Overlap can shorten elapsed time, but it does not erase activity:

Predicted quantity	Energy use
t_k	Static energy $P_{\text{idle}} t_k$
Issued TC/vector work	Compute event energy
SMEM request bytes	Shared-memory event energy
L2 request bytes	L2-interface event energy
HBM-interface bytes	HBM event energy

Complete padded MMA tiles are charged even when some lanes do not contribute to useful output. A byte can be charged at more than one memory interface when it crosses multiple accounting boundaries, while reuse can make the totals at those boundaries differ.

3.2 Why Peak Roofline Is Insufficient

A simple Roofline estimator converts work and traffic directly to time. For work F, effective compute rate $\bar{P}^{\text{eff}}$, and traffic $\mathcal{T}_\ell$ at memory level ℓ, it uses

$$\begin{aligned} C_{\text{roof}} &= \max\left(\frac{F}{\bar{P}^{\text{eff}}}, \max_{\ell} \left\{ \frac{\mathcal{T}_\ell}{\bar{\beta}_\ell^{\text{eff}}} + C_\ell^{\text{lat}} \right\} \right), \\ t_{\text{roof}} &= \frac{C_{\text{roof}}}{f_{\text{clk}}}. \end{aligned} \tag{3.3}$$

This is a useful latency baseline, but it treats each resource as one complete path. It does not expose which instructions are issued by a concrete tile, how activity changes in a tail wave, or where prologue, body, and epilogue services overlap. Those omissions affect both static energy through t_k and dynamic energy through issued work and per-level traffic. Figure 2.1 summarizes the execution-aware path from operation and hardware inputs to the timing and activity quantities consumed by GPU-local energy prediction.

3.3 Effective Rates and Wave Composition

Table inputs are normalized to the scope and units consumed by the service equations. Let n_{SM} be the GPU's SM count, $n_{\text{SMSP/SM}}$ its SMSP count per SM, f_{ref} the reference clock associated with

the advertised Tensor Core peak, and f_{clk} the configured clock. The per-SMSP Tensor Core peak and per-cycle memory rates are

$$\begin{aligned}
\bar{\pi}^{\mathrm{peak}}_{\mathrm{TC},j} &= \frac{\Pi^{\mathrm{peak}}_{\mathrm{TC,ref}}}{n_{\mathrm{SM}} n_{\mathrm{SMSP/SM}} f_{\mathrm{ref}}}, \\
\Pi^{\mathrm{peak}}_{\mathrm{TC}}(f_{\mathrm{clk}}) &= n_{\mathrm{SM}} n_{\mathrm{SMSP/SM}} \bar{\pi}^{\mathrm{peak}}_{\mathrm{TC},j} f_{\mathrm{clk}}, \\
\bar{\beta}^{\mathrm{peak}}_{\mathrm{SMEM,SM}} &= \frac{\beta^{\mathrm{peak}}_{\mathrm{SMEM,device}}}{n_{\mathrm{SM}} f_{\mathrm{clk}}}, \\
\bar{\beta}^{\mathrm{peak}}_{\ell} &= \frac{\beta^{\mathrm{peak}}_{\ell}}{f_{\mathrm{clk}}}, \qquad \ell \in \{\mathrm{L2}, \mathrm{HBM}\}.
\end{aligned} \tag{3.4}$$

For the A100 values in Table 5.1, the first line gives $\bar{\pi}^{\mathrm{peak}}_{\mathrm{TC},j} = 512$ FLOP/cycle/SMSP. Tensor Core throughput scales with the configured core clock through the second line; the measured or documented memory rates remain rates per second and are expressed in bytes per cycle for cycle-domain service calculations.

An advertised peak is an upper bound, not a throughput guarantee. For resource r, let Q_r be its serviced quantity, $Q_{r,\mathrm{inflight}}$ the independently serviceable quantity simultaneously visible to it, and C^{lat}_r its service or dependency latency in cycles. Little's Law gives [19]

$$\begin{aligned}
\bar{\mu}^{\mathrm{eff}}_r &= \min\left(\bar{\mu}^{\mathrm{peak}}_r, \frac{Q_{r,\mathrm{inflight}}}{C^{\mathrm{lat}}_r}\right) \le \bar{\mu}^{\mathrm{peak}}_r, \\
C_r &= \frac{Q_r}{\bar{\mu}^{\mathrm{eff}}_r}.
\end{aligned} \tag{3.5}$$

Reaching peak requires $Q_{r,\mathrm{inflight}} \ge \bar{\mu}^{\mathrm{peak}}_r C^{\mathrm{lat}}_r$. Unavoidable memory-access and dependency latencies therefore lower the effective rate when a kernel exposes insufficient independent work. The predictor derives in-flight work from resident warps, accumulator chains, memory stages, or active rows.

Within a wave, causally ordered services add and legally concurrent services take a maximum. Waves themselves serialize. If C_w is the duration of wave w, the kernel time used by the static-energy

term is

$$
\begin{aligned}
C_k &= \sum_w C_w + C_{\text{fixed}}, \\
t_{\text{fixed}} &= \frac{C_{\text{fixed}}}{f_{\text{clk}}}, \\
t_k &= \frac{\sum_w C_w}{f_{\text{clk}}} + t_{\text{fixed}} = \frac{C_k}{f_{\text{clk}}}.
\end{aligned}
\tag{3.6}
$$

C_{fixed} is the profiled device overhead in cycles, added once per kernel; t_{fixed} is its corresponding time in seconds. Bars denote rates per cycle, as made explicit in Equation (3.4).

3.4 Kernel Geometry and Whole-Kernel Activity

For GEMM and FlashAttention, most schedules give each CTA exclusive ownership of one output tile. Kernel tiling determines the instructions and traffic of that CTA, while resource limits determine how many CTAs reside concurrently. The predictor evaluates complete and tail waves separately, then sums issued work and traffic over all physical CTAs. Detailed warp geometry, service equations, and schedule diagrams are provided in Appendix A.

3.4.1 GEMM. Let B be the dimensionless batch multiplicity, and let M, N, and K be the GEMM output-row, output-column, and reduction dimensions, respectively, each measured as an element count. Thus, each of the B GEMMs maps an $M \times K$ operand and a $K \times N$ operand to an $M \times N$ output. For CTA tile $(M_{\text{CTA}}, N_{\text{CTA}}, K_{\text{CTA}})$,

$$
G_M = \left\lceil \frac{M}{M_{\text{CTA}}} \right\rceil, \quad G_N = \left\lceil \frac{N}{N_{\text{CTA}}} \right\rceil, \quad G_K = \left\lceil \frac{K}{K_{\text{CTA}}} \right\rceil. \tag{3.7}
$$

The output grid contains $n_{\text{CTA}} = BG_MG_N$ CTAs; G_K is the number of successive reduction stages within each CTA. With $F_{\text{useful}} = 2BMNK$, complete instruction tiles give

$$
\begin{aligned}
F_{\text{issued}} &= 2BG_MG_NG_KM_{\text{CTA}}N_{\text{CTA}}K_{\text{CTA}}, \\
\eta_{\text{tile}} &= \frac{F_{\text{useful}}}{F_{\text{issued}}}.
\end{aligned}
\tag{3.8}
$$

Thus kernel selection and edge padding change the compute energy even when useful work is unchanged.

Let W_{CTA}, $S_{\mathrm{SMEM,CTA}}$, and $N_{\mathrm{reg,CTA}}$ be the warp, shared-memory, and register requirements of one CTA. The resident CTA count per SM is

$$R = \min\left\{ \begin{array}{l} R^{\max}_{\mathrm{CTA}}, \left\lfloor \dfrac{W^{\max}_{\mathrm{SM}}}{W_{\mathrm{CTA}}} \right\rfloor, \left\lfloor \dfrac{S^{\max}_{\mathrm{SMEM,SM}}}{S_{\mathrm{SMEM,CTA}}} \right\rfloor, \\ \left\lfloor \dfrac{N^{\max}_{\mathrm{reg,SM}}}{N_{\mathrm{reg,CTA}}} \right\rfloor \end{array} \right\}. \tag{3.9}$$

With R resident CTAs per SM and n_{SM} SMs,

$$n_{\mathrm{waves}} = \left\lceil \frac{n_{\mathrm{CTA}}}{n_{\mathrm{SM}} R} \right\rceil. \tag{3.10}$$

A tail wave has fewer resident warps, accumulator chains, and outstanding memory requests, so it is evaluated explicitly rather than as a fractional full wave. The same geometry counts SMEM requests, CTA-requested L2 bytes, and estimated HBM-interface bytes over every K-stage. The detailed local-service and transfer-window equations are in Equations (A.1) to (A.13).

Each wave consists of an input prologue, an overlapped compute/memory body, and an output epilogue. The body is the interval in which local computation and operand movement can overlap, so it is the only phase for which a Roofline maximum is appropriate. The prologue and epilogue are memory-only intervals; they serialize before and after the body and are not included in that Roofline maximum:

$$C^{\mathrm{GEMM}}_{\mathrm{wave}} = C_{\mathrm{prologue}} + C_{\mathrm{body}} + C_{\mathrm{epilogue}}. \tag{3.11}$$

Let $n_{\mathrm{waves,full}}$ be the number of complete waves and $\mathbf{1}_{\mathrm{tail}}$ indicate a tail wave. Specializing Equation (3.6) gives

$$C^{\mathrm{GEMM}}_{k} = n_{\mathrm{waves,full}} C^{\mathrm{GEMM}}_{\mathrm{wave,full}} + \mathbf{1}_{\mathrm{tail}} C^{\mathrm{GEMM}}_{\mathrm{wave,tail}} + C_{\mathrm{fixed}}. \tag{3.12}$$

Detailed phase definitions and two numerical GEMM examples appear in Appendix A.

3.4.2 FlashAttention. FlashAttention uses the same CTA tiling, residency, effective-rate, and whole-kernel activity accounting as GEMM. Each CTA owns one query/output tile and repeatedly performs a CTA-local QK product, online softmax update, and PV product while streaming key–value tiles [7]. The important difference is causal heterogeneity: CTAs for earlier query blocks have shorter legal key–value prefixes and can finish before other CTAs in the same wave.

For batch B, H_q query heads, query length S_q, query tile T_Q, and key–value tile T_{KV},

$$n_{\mathrm{CTA}}^{\mathrm{FA}} = BH_q \left\lceil \frac{S_q}{T_Q} \right\rceil . \tag{3.13}$$

Query block i, with legal prefix length L_i, executes

$$J_i = \left\lceil \frac{L_i}{T_{KV}} \right\rceil \tag{3.14}$$

KV-tile iterations. If $\mathcal{W}_w$ is the CTA set of wave w and J_a is the iteration count of CTA a, the active set at iteration j is

$$\mathcal{A}_{w,j} = \{a \in \mathcal{W}_w \mid j < J_a\}, \qquad J_w = \max_{a \in \mathcal{W}_w} J_a. \tag{3.15}$$

The model groups active CTAs with the same live tile shapes and execution schedule into class c. Its multiplicity $m_{w,j,c}$ counts the physical CTAs represented by that class. Multiplicity scales additive work and traffic, not class duration, because those CTAs execute concurrently within the wave.

For head dimension D, the CTA-local products have shapes $(T_Q \times D)(D \times T_{KV})$ and $(T_Q \times T_{KV})(T_{KV} \times D)$. They reuse the GEMM Tensor Core and SMEM service calculation; softmax scalar and special-function activity is accounted separately. Issued Tensor Core activity is

$$F_{\mathrm{TC}}^{\mathrm{FA}} = \sum_{w,j,c} m_{w,j,c} \left(F_{QK,w,j,c}^{\mathrm{issued}} + F_{PV,w,j,c}^{\mathrm{issued}} \right) . \tag{3.16}$$

L2 activity counts each active CTA's K/V requests. HBM activity assumes ideal reuse within a wave: active CTAs with the same batch index and KV head share one fetch of a given KV tile. The

detailed phase-overlap schedule is in Appendix A.

3.5 Scope and Limitations

A limitation is that the predictor requires compiler-dependent details of the selected kernel configuration. Its tiling, data type, staging structure, and operation mapping determine the geometry, issued work, traffic, dependency chains, and pipeline structure derived by the model. The predictor does not require an emitted instruction trace or dynamic execution trace; it derives these quantities analytically from `LocalOp`, the kernel configuration, and `HardwareSpec`. Bank conflicts, instruction replays, cache conflicts, framework activity, and undocumented scoreboard limits remain outside scope and appear as residual error in validation. The model is analytical rather than cycle-exact: equivalent CTA schedules may be aggregated by multiplicity, but their total work, traffic, occupancy, and critical-path service are preserved.

Chapter 4 LLM Inference Composition

The execution-aware model predicts one concrete GPU operation, whereas an LLM request contains many operation instances whose shapes and counts depend on the architecture, prompt, generated sequence, and tensor-parallel plan. RAPID-LLM bridges these scales by lowering the request to a dependency graph and attaching a predicted latency and activity profile to every local node. It then inserts the collectives required by tensor parallelism and schedules their transfers [14]. This work combines the GPU-local activity profiles with the scheduled system-wide communication activity to predict end-to-end energy.

4.1 Single-GPU Workload Graph Construction

Software and workload inputs specify the LLM architecture, tensor-parallel plan, and inference request. The architecture includes layer count, hidden and intermediate dimensions, attention and key/value head counts, vocabulary size, and data type; the request supplies batch size, prompt length, and output length. Hardware and system parameters specify the GPU and interconnect configuration, including compute and memory ceilings, energy coefficients, clock frequency, idle static power, device count, and topology.

For each Transformer layer, RAPID-LLM derives tensor shapes for QKV projections, attention score and value products, attention output projection, normalization, and feed-forward projections. Fused prefill attention becomes a FlashAttention `LocalOp`; otherwise, attention is represented by its constituent matrix multiplications, softmax, and elementwise operations. Each `LocalOp` returns an `OpProfile` containing the selected kernel, tile geometry, latency, issued FLOPs, and HBM, L2, and shared-memory traffic [14].

RAPID-LLM connects these nodes according to Transformer dataflow, repeats the layer template, and adds token embedding, final normalization, and the language-model head. The graph makespan

gives phase latency.

Prefill consumes the prompt and produces output token 1. A request for L_{out} output tokens then executes $L_{\text{out}} - 1$ decode graphs whose attention and key–value cache shapes grow with context length. RAPID-LLM evaluates configured decode sample points and integrates latency and activity between them rather than simulating every long-generation step independently [14].

4.2 Multi-GPU Workload Extension

Tensor parallelism first changes the local work assigned to each GPU by sharding operation dimensions. RAPID-LLM queries the local predictor with the shapes actually executed by each device, then inserts all-reduce, reduce-scatter, or all-gather nodes at the corresponding data dependencies [44, 36]. Communication is therefore not a post-processing penalty: a dependent kernel cannot begin until its inputs arrive, while graph-independent work may overlap when the represented compute and network resources permit it.

4.3 RAPID-LLM Network Scheduling

RAPID-LLM supplies local operation service times, collective type, message size, participating ranks, dependencies, and an interconnect description. Its network-scheduling component expands each collective into transfers and schedules them subject to dependencies and shared network resources. The selected collective algorithm and topology make transfers over directly connected device–device and device–switch links explicit. RAPID-LLM supports hierarchical networks composed from Ring, Fully Connected, and Switch building blocks [14, 5].

The interconnect can be viewed as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where vertices are devices or switches and directed link $\lambda \in \mathcal{E}$ has effective bandwidth $\beta_\lambda^{\text{eff}}$ and fixed per-transfer time $t_{\lambda,\text{fixed}}$ (s).

Table 4.1. Canonical collective payload per directed logical link. This table is a sanity check for this work's link-resolved byte accounting: dynamically accumulated $\mathcal{T}_\lambda$ matches these analytical expectations for the corresponding topology and collective schedule. n_{GPU} is the device count and $\mathcal{S}$ is collective payload size in bytes.

Collective	Unidirectional Ring	Bidirectional Ring	Fully Connected	Switch
All-gather	$\mathcal{S}(n_{\mathrm{GPU}}-1)/n_{\mathrm{GPU}}$	$\mathcal{S}(n_{\mathrm{GPU}}-1)/(2n_{\mathrm{GPU}})$	$\mathcal{S}/n_{\mathrm{GPU}}$	$\mathcal{S}(n_{\mathrm{GPU}}-1)/n_{\mathrm{GPU}}$
Reduce-scatter	$\mathcal{S}(n_{\mathrm{GPU}}-1)/n_{\mathrm{GPU}}$	$\mathcal{S}(n_{\mathrm{GPU}}-1)/(2n_{\mathrm{GPU}})$	$\mathcal{S}/n_{\mathrm{GPU}}$	$\mathcal{S}(n_{\mathrm{GPU}}-1)/n_{\mathrm{GPU}}$
All-reduce	$2\mathcal{S}(n_{\mathrm{GPU}}-1)/n_{\mathrm{GPU}}$	$\mathcal{S}(n_{\mathrm{GPU}}-1)/n_{\mathrm{GPU}}$	$2\mathcal{S}/n_{\mathrm{GPU}}$	$2\mathcal{S}(n_{\mathrm{GPU}}-1)/n_{\mathrm{GPU}}$

If schedule step s sends $\mathcal{T}_{\lambda,s}$ bytes in $n_{\lambda,s}$ transactions over link λ, its service time is

$$t_{\lambda,s} = n_{\lambda,s} t_{\lambda,\mathrm{fixed}} + \frac{\mathcal{T}_{\lambda,s}}{\beta_\lambda^{\mathrm{eff}}}. \tag{4.1}$$

Concurrent links progress together, whereas dependent steps serialize [5]. The maximum participating-rank completion time sets collective latency. The network schedule dynamically accumulates each link's transaction count $n_\lambda = \sum_s n_{\lambda,s}$ and payload traffic $\mathcal{T}_\lambda = \sum_s \mathcal{T}_{\lambda,s}$, in addition to reduction-memory activity. These direct-link statistics make the energy model topology-aware: links at different hierarchy levels or using different technologies, such as NVLink and PCIe, can have distinct traffic-energy costs. Aggregate link busy time is an activity statistic, not a workload makespan: it may sum simultaneous use of multiple links. The dependency graph determines how much collective latency is exposed on the request's critical path, while all executed traffic remains chargeable to energy.

All-reduce is the sum of its reduce-scatter and all-gather phases, which gives the final row of Table 4.1. The table reports payload bytes only; transaction counts remain specific to the selected algorithm and are obtained from the dynamic schedule [36].

4.4 Ring All-Reduce Network Activity

The unidirectional Ring all-reduce specialization of Table 4.1 has the following activity. For payload $\mathcal{S}$ on n_{GPU} GPUs, each GPU—and therefore each directed Ring link—transmits $2(n_{\mathrm{GPU}} -$

$1)\mathcal{S}/n_{\mathrm{GPU}}$ bytes. The system transmits $\mathcal{T}_{\mathrm{TX}} = 2(n_{\mathrm{GPU}} - 1)\mathcal{S}$ bytes, and each Ring link performs $2(n_{\mathrm{GPU}} - 1)$ sequential transfers. For this homogeneous specialization, $\beta_{\lambda}^{\mathrm{eff}} = \beta_{n_{\mathrm{GPU}}}$ and $t_{\lambda,\mathrm{fixed}} = t_{\mathrm{link,fixed}}$. Its latency is

$$t_{n_{\mathrm{GPU}}}(\mathcal{S}) = \frac{2(n_{\mathrm{GPU}} - 1)\mathcal{S}}{n_{\mathrm{GPU}}\beta_{n_{\mathrm{GPU}}}} + 2(n_{\mathrm{GPU}} - 1)t_{\mathrm{link,fixed}}. \tag{4.2}$$

The balanced schedule assigns each GPU $3(n_{\mathrm{GPU}}-1)\mathcal{S}/n_{\mathrm{GPU}}$ of reduction-related HBM traffic: two reads and one write for every reduced byte. Summing over GPUs gives $3(n_{\mathrm{GPU}} - 1)\mathcal{S}$. Substitution into the system-wide communication-energy equation in Chapter 2 gives

$$\begin{aligned} E_{n_{\mathrm{GPU}}}(\mathcal{S}) &= n_{\mathrm{GPU}}\,(P_{\mathrm{idle}} + P_{\mathrm{active}})\,t_{n_{\mathrm{GPU}}}(\mathcal{S}) + e_{\mathrm{link}}\,2(n_{\mathrm{GPU}} - 1)\mathcal{S} + e_{\mathrm{HBM}}\,3(n_{\mathrm{GPU}} - 1)\mathcal{S} \\ &= n_{\mathrm{GPU}}\,(P_{\mathrm{idle}} + P_{\mathrm{active}})\,t_{n_{\mathrm{GPU}}}(\mathcal{S}) + e_{\mathrm{traffic}}\,2(n_{\mathrm{GPU}} - 1)\mathcal{S}, \\ e_{\mathrm{traffic}} &= e_{\mathrm{link}} + \tfrac{3}{2}e_{\mathrm{HBM}}. \end{aligned} \tag{4.3}$$

Here e_{link} prices transmitted payload bytes only, whereas the derived e_{traffic} combines payload-link and reduction-memory energy for this balanced Ring schedule. The time-dependent term includes both the idle-static-power contribution and the incremental collective power; the calibration described in Chapter 5 fits only the latter from idle-subtracted measurements. For an asymmetric collective, the general per-GPU reduction sum and per-link traffic sum in Equation (2.8) apply instead.

4.5 Full-Workload Latency and Energy

RAPID-LLM propagates local and network profiles using the same layer, phase, and token multiplicities used to construct the request. Decode statistics at sampled context lengths use the same interpolation weights for latency, activity, and energy.

The scheduled graph makespan reports latency. The network term aggregates executed collective activity across participating devices and physical links, and is added once to GPU-local energy. Reduction-memory traffic is excluded from the local memory term when it has already been charged

by the network model. This composition is interpretable rather than cycle-exact: it assumes explicit architecture, tensor-parallel, collective, and topology inputs; permits overlap only through represented graph and resource dependencies; interpolates smoothly between simulated decode contexts; and applies calibrated network parameters only within the validated topology and operating point. Reduction-memory activity contributes energy but no separately modeled reduction-compute or memory-service latency.

Chapter 5 Energy-Model Inputs and Calibration

This section separates documented model inputs from the small set of profiled or fitted terms. Every input belongs to one of three evidence classes. Vendor specifications provide advertised capacities, instruction support, nominal peaks, and product topology. Published microbenchmarks provide dependency latency, attainable on-chip bandwidth, and per-event energy when specifications do not define a predictive value. This work's calibration measurements are reserved for fixed or system-specific residuals lacking a suitably matched published value: local profiling supplies only fixed kernel overhead and idle static power, while collective measurements identify effective network utilization, fixed transfer latency, per-GPU active network power, and energy per transmitted byte.

Under this evidence hierarchy, nominal rates, capacities, traffic rules, memory latencies, and event-energy coefficients remain documented inputs. Calibration rows are disjoint from held-out validation rows; the validation workloads are therefore not used to estimate the calibrated parameters.

5.1 A100 80GB Input Provenance

Table 5.1 lists the Ampere inputs shared by the evaluated A100 80GB systems. Architecture limits and nominal capacities come from product documentation; quantities without a vendor-defined predictive service value come from A100 microbenchmarks [21, 8]. Cycle latencies are used directly. Per-second memory bandwidths are converted at the configured clock, whereas Tensor Core throughput is first normalized at its advertised reference clock. The memory-energy coefficients represent coalesced block-sized transfers described by the analytical schedule, rather than fitting coefficients to the LLM validation workloads.

CTA and warp tiles, pipeline depth, alignment, data type, issued work, and traffic are `LocalOp` properties rather than hardware constants. Effective rates are derived from the documented peaks

Table 5.1. Documented Ampere `HardwareSpec` inputs shared by the A100 PCIe 80GB and A100 SXM4 80GB systems.

Input	Variable	Ampere Parameters	Evidence	Source
Compute and execution				
GPU organization	$n_{\mathrm{SM}}, n_{\mathrm{SMSP/SM}}$	108 SMs; 4 SMSPs/SM	Vendor specification	[28]
Warp residency per SM	$W_{\mathrm{SM}}^{\max}$	64 warps/SM	Vendor specification	[26]
CTA residency per SM	$R_{\mathrm{CTA}}^{\max}$	32 CTAs/SM	Vendor specification	[26]
Shared-memory capacity per SM	$S_{\mathrm{SMEM,SM}}^{\max}$	164 KiB/SM	Vendor specification	[26]
Register capacity per SM	$N_{\mathrm{reg,SM}}^{\max}$	64 Ki 32-bit registers/SM (256 KiB)	Vendor specification	[26]
BF16 Tensor Core peak; reference clock	$\Pi_{\mathrm{TC,ref}}^{\mathrm{peak}}, f_{\mathrm{ref}}$	312 TFLOP/s; 1.41 GHz	Vendor specification	[28]
BF16 MMA latency	$C_{\mathrm{TC}}^{\mathrm{lat}}$	8 cycles	External microbenchmark	[45]
BF16 Tensor Core energy	e_{TC}	0.70 pJ/FLOP	External microbenchmark	[1]
FP16 vector energy	e_{FP16}	12.97 pJ/FLOP	External microbenchmark	[1]
Memory hierarchy and data movement				
Memory capacity	—	40 MiB L2; 80 GiB HBM2e	Vendor specification	[28]
Device-wide shared-memory bandwidth; latency	$\beta_{\mathrm{SMEM,device}}^{\mathrm{peak}}, C_{\mathrm{SMEM}}^{\mathrm{lat}}$	14.8 TB/s; 29.0 cycles	External microbenchmark	[1, 21]
Device-wide L2 bandwidth; latency	$\beta_{\mathrm{L2}}^{\mathrm{peak}}, C_{\mathrm{L2}}^{\mathrm{lat}}$	4.3 TB/s; 202.8 cycles	External microbenchmark	[1, 21]
Device-wide HBM bandwidth; latency	$\beta_{\mathrm{HBM}}^{\mathrm{peak}}, C_{\mathrm{HBM}}^{\mathrm{lat}}$	1.935 TB/s; 566 cycles	Vendor; external microbenchmark	[28, 21]
Shared-memory energy	e_{SMEM}	12.72 pJ/B	External microbenchmark	[8]
L2 energy	e_{L2}	37.68 pJ/B	External microbenchmark	[8]
HBM energy	e_{HBM}	65.3125 pJ/B	External microbenchmark	[8]

and latencies together with the operation's in-flight work; they are not fitted. The organization and rate inputs enter Equation (3.4), the residency limits determine R through Equation (3.9), the latency inputs enter Equations (A.3), (A.5) and (A.11), and the event-energy coefficients enter Equations (2.3) and (2.5).

5.2 Local-Model Profiled Measurements

Local measurements supply two profiled inputs: idle static power $P_{\mathrm{idle}}^{(g)}$ in Equation (2.2), and the fixed timing input C_{fixed} in Equation (3.6). A tiny one-warp kernel isolates launch-to-completion overhead without introducing a shape-dependent correction. Its fixed-time component is $t_{\mathrm{fixed}} = C_{\mathrm{fixed}}/f_{\mathrm{clk}}$ seconds, as defined in Equation (3.6), and contributes to $t_{k,\mathrm{active}}^{(g)}$. A quiescent measurement supplies idle static power at the evaluated operating point. Multi-GPU entries average

Table 5.2. Profiled local-model inputs, averaged by system: idle static power $P_{\text{idle}}^{(g)}$ and fixed kernel cycles C_{fixed}. The latter contributes to active time $t_{k,\text{active}}^{(g)}$ through Equation (3.6), and both enter Equation (2.2). The A100 fixed-overhead values are comparable to the 2.2605 μs warmed null-kernel launch overhead reported on a separate A100 platform [47].

System	Clock	$P_{\text{idle}}^{(g)}$	C_{fixed}
1× A100 PCIe 80GB	1065 MHz	55.00 W	2500 cycles (2.347 μs)
2× A100 SXM4 80GB	1275 MHz	84.25 W	3609 cycles (2.831 μs)
8× A100 SXM4 80GB	1275 MHz	78.57 W	3331 cycles (2.613 μs)

per-device values for configuration, while workload energy still sums all participating devices.

5.2.1 Fixed kernel overhead. t_{fixed} uses warmed repeated one-warp launches timed by CUDA events. It represents launch, dispatch, driver, and device-control work that does not scale with modeled operation work. Treating it as one device-level residual preserves the first-principles shape-dependent terms [27].

5.2.2 Idle static power. NVIDIA Management Library (NVML) supplies device-power telemetry, which is sampled during a quiescent interval to estimate idle static power [37]. The A100 product specification reports maximum thermal design power rather than an idle-power value [28]. Temperature changes static leakage power, while clock and power state affect the measured board baseline [20, 39]. Idle static power is charged over active wall time separately from documented event energy.

5.3 Collective Calibration Methodology

The topology-independent network-energy framework in Section 4.2 already prices activity on explicit directed links. This subsection calibrates its system-specific parameters. NVIDIA Collective Communications Library (NCCL) Ring all-reduce is a convenient measurement workload because

its transfer count and per-link payload are explicit; it does not introduce a separate Ring energy model. Instead, the measurements calibrate the parameters used by the Ring all-reduce latency and energy specializations in Equations (4.2) and (4.3).

Ring all-reduce sweeps span 1 KiB–1 GiB on two and eight A100 SXM4 80GB GPUs. The systems have different active-port counts, reference bandwidths, and Ring transfer counts, but share one parameterization. Documented third-generation NVLink port counts and 25 GB/s unidirectional bandwidth per port define $\beta_{\mathrm{ref},n_{\mathrm{GPU}}}$ [25]. The four fitted network parameters are shared across both GPU counts and are estimated jointly from the latency and energy sweeps.

5.3.1 Effective network utilization. The large-message latency slope identifies effective network utilization u_{eff} in $\beta_{n_{\mathrm{GPU}}} = \beta_{\mathrm{ref},n_{\mathrm{GPU}}} u_{\mathrm{eff}}$. For these Ring all-reduce measurements, u_{eff} summarizes the resulting protocol synchronization and chunking, channel and CTA parallelism, and other implementation effects rather than an intrinsic fraction of NVLink peak bandwidth [11].

5.3.2 Fixed network-transfer latency. The fixed per-transfer time $t_{\mathrm{link,fixed}}$ represents the size-independent latency of each serial Ring step and enters the Ring all-reduce completion time in Equation (4.2). At small message sizes, this fixed term dominates collective time, explaining the constant-time plateau observed in Figure 5.1.

5.3.3 Per-GPU active network power. After documented reduction-memory energy is accounted for, the collective energy residual is decomposed into time- and payload-proportional terms. The time-proportional component identifies P_{active} per participating GPU.

Because P_{active} is calibrated from idle-subtracted energy, it does not replace the per-GPU idle static power P_{idle}. Full-workload energy charges $P_{\mathrm{idle}} + P_{\mathrm{active}}$ during a collective; the plots in Figure 5.1 isolate P_{active} and traffic-dependent energy.

Table 5.3. Shared two- and eight-GPU A100 SXM4 80GB Ring all-reduce calibration parameters. u_{eff}, $t_{\text{link,fixed}}$, P_{active}, and e_{link} are fitted jointly across GPU counts; the effective bandwidth follows from the documented third-generation NVLink reference bandwidth and u_{eff}.

System	GPU count n_{GPU}	Active ports	NVLink 3 / port (GB/s)	Reference bandwidth (GB/s)	Effective utilization u_{eff}	Effective bandwidth (GB/s)	$t_{\text{link,fixed}}$ per transfer (μs)	P_{active} per GPU (W/GPU)	e_{link} (pJ/B)
A100 SXM4 80GB	2 8	4 12	25	100 300	0.71	70.68 212.05	10.38	8.70	214.03

Table 5.4. Derived A100 SXM4 80GB Ring all-reduce latency and energy floors from the shared calibration parameters. Serial Ring steps set the latency floor, and charging P_{active} on every participating GPU over that duration gives the corresponding energy floor.

n_{GPU}	Serial Ring steps $2(n_{\text{GPU}}-1)$	$t_{\text{link,fixed}}$ (μs)	Latency floor (μs) $2(n_{\text{GPU}}-1)t_{\text{link,fixed}}$	P_{active} (W/GPU)	Energy floor (mJ)
2 8	2 14	10.38	20.76 145.34	8.70	0.36 10.12

5.3.4 Energy per transmitted network byte. The payload-proportional component of the same energy fit identifies e_{link}, the energy per byte transmitted over a direct GPU–GPU link. Documented reduction-memory energy is kept separate, so this coefficient prices routed link activity without absorbing the HBM traffic used to reduce partial results.

Table 5.3 separates the documented, GPU-count-dependent reference inputs from the fitted terms shared across rows. The active-port count and 25 GB/s per-port rate set $\beta_{\text{ref},n_{\text{GPU}}}$; u_{eff} converts it to $\beta_{n_{\text{GPU}}}$ in Equation (4.2). The same $t_{\text{link,fixed}}$, P_{active}, and e_{link} enter Equations (4.2) and (4.3) for both GPU counts.

The shared fit provides a transfer test across GPU count: active ports, reference bandwidth, and serial Ring steps change from two to eight GPUs while the fitted utilization, per-transfer latency, active power, and byte-energy parameters remain fixed. These parameters are intended to transfer across GPU counts and topologies when the underlying link technology is unchanged and the physical topology and collective schedule are explicit. The calibration does not establish transfer across protocols, collective algorithms, or fabric generations without new measurements.

Table 5.4 is a derived zero-payload diagnostic, not an additional fit. As $\mathcal{S}$ approaches zero, the

serialization term in Equation (4.2) vanishes and the $2(n_{\mathrm{GPU}} - 1)$ serial transfers leave the latency floor shown in the table. Equation (4.3) then charges P_{active} on all n_{GPU} participating GPUs over that floor, which explains why the eight-GPU energy floor grows faster than the latency floor.

5.3.5 All-Reduce Calibration Evidence and Scope. The calibration experiments force the NCCL Simple protocol for all Ring all-reduce measurements. The two-rank placement activates four 25 GB/s third-generation NVLink ports per GPU, while the eight-rank placement activates twelve. Data Center GPU Manager (DCGM) supplies per-GPU accumulated energy and NVLink transmit/receive payload rates [32, 33]. The analysis sums energy across participating GPUs, subtracts idle static power in post-processing, and integrates the per-link rates over each measurement interval. A separate `nvidia-smi` power integration provides a consistency cross-check, not an independent sensor measurement, because `nvidia-smi` is built on NVML [39]. A100 power sampling can omit short-duration variation, so this cross-check is not used as the calibration reference [49].

For collective payload size $\mathcal{S}$, every directed Ring link transmits $2(n_{\mathrm{GPU}}-1)\mathcal{S}/n_{\mathrm{GPU}}$ bytes, while the system-wide transmitted traffic is $\mathcal{T}_{\mathrm{TX}} = 2(n_{\mathrm{GPU}}-1)\mathcal{S}$. The per-link counters therefore validate the traffic used by the serialization term in Equation (4.2), and their system sum gives the payload term in Equation (4.3). Equation (4.2) produces the calibrated latency curves; Equation (4.3) adds active GPU energy, transmitted-payload energy, and reduction-HBM energy to produce the system-energy curves. The calibrated e_{link} prices the routed path of a direct GPU–GPU link, including NVSwitch energy, so the model has no separate NVSwitch-energy term.

At small messages, serialization contributes little and the fixed per-transfer term governs latency; once messages saturate the links, the predicted slope follows configured serialization bandwidth. Energy has the complementary separation: a small-message floor captures elapsed collective activity, while the near-linear large-message regime is explained by transmitted bytes and documented reduction-memory traffic. These roles are visible in Figure 5.1. The plots therefore validate the narrow calibration boundary rather than a per-size fitted surrogate. The calibration and validation measurements cover only Ring all-reduce on the evaluated NVSwitch systems. This work neverthe-

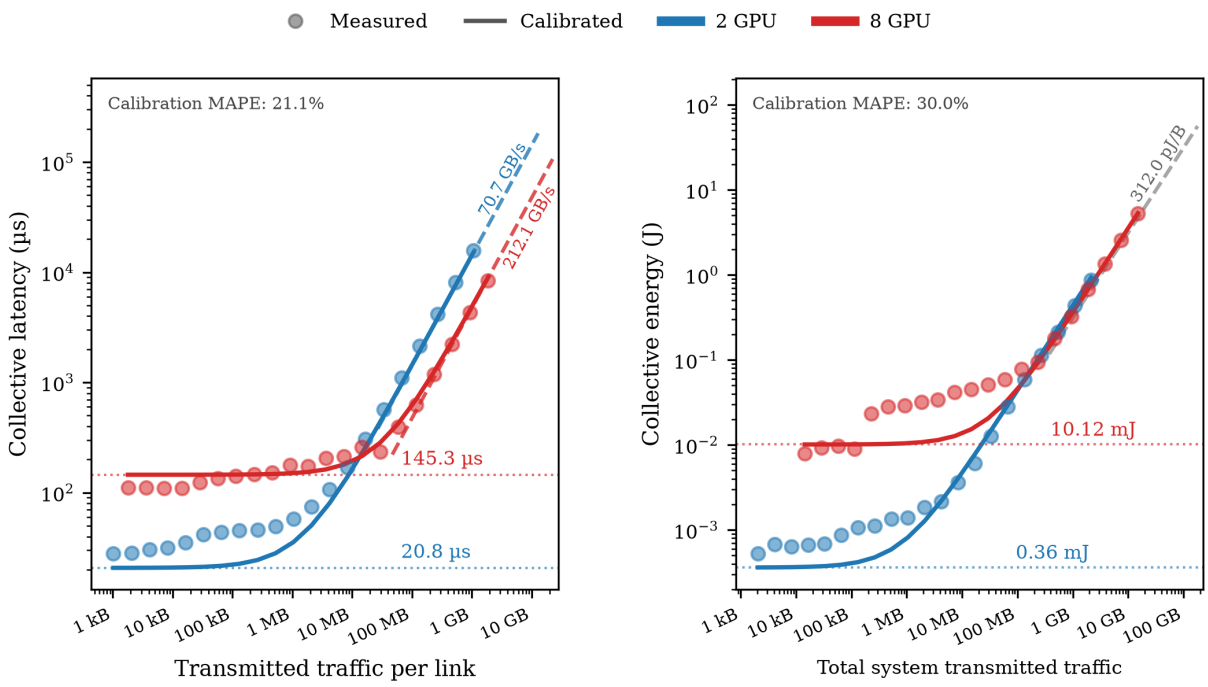

Figure 5.1. Measured and modeled Ring all-reduce latency and system energy from Equations (4.2) and (4.3) for 1 KiB–1 GiB messages on two- and eight-GPU A100 SXM4 80GB NVSwitch systems. DCGM energy is idle-subtracted; per-link payload counters agree with modeled traffic within 0.0183% above 64 MiB.

less applies the calibrated per-link parameters to another explicit topology when it uses the same link technology: its collective schedule determines the link-level activity supplied to the general model. This topology transfer is an intended modeling assumption, not an independently validated result; validation on additional topologies remains future work.

Chapter 6 Validation

This section evaluates the calibrated model on held-out operator and workload energy measurements and separately checks the latency and activity predictions on which energy depends. Calibration uses only the fixed residual terms and measurement protocols specified in Chapter 5; no workload-specific energy correction is fit to the results below.

6.1 Kernel-Level Latency and Energy Validation on A100 PCIe 40GB

Kernel validation uses held-out BF16 GEMM and FlashAttention rows from the published EnergAIzer A100 PCIe 40GB database [17]. The dataset spans regular, skinny, small-K, and vector-like shapes. Kernel selection, CTA tiling, issued work, memory traffic, occupancy, and full/tail waves are fixed before comparison with measurement. Sixteen disjoint kernels calibrate only fixed latency overhead; event-energy coefficients are not fitted. Measurements use 900 MHz, 47.94 W idle static power, and a 23,181-cycle fixed overhead.

FlashAttention results exclude kernels whose output requires cross-block accumulation because that schedule is outside the supported analytical path. The ablations below use the same held-out GEMMs to isolate what the model adds beyond ideal Roofline work counts. The latency ablation holds the kernel set fixed while adding fixed overhead and then the execution-aware schedule. The energy ablation accumulates the model's static and dynamic components before evaluating the complete prediction.

Figure 6.1 shows that a peak Roofline systematically misses kernels whose ideal work time is small relative to fixed overhead. Adding that overhead lowers MAPE from 75.2% to 17.7%, but it still assumes peak service rates. The execution-aware model lowers MAPE further to 9.4% by accounting for the selected tile, CTA waves, and constrained resource service.

Table 6.1. Held-out kernel-level validation against published A100 PCIe 40GB measurements [17]. MAPE denotes mean absolute percentage error.

Operation	Rows	Latency MAPE	Energy MAPE
BF16 GEMM	3,520	9.44%	11.41%
BF16 FlashAttention	534	8.20%	15.32%

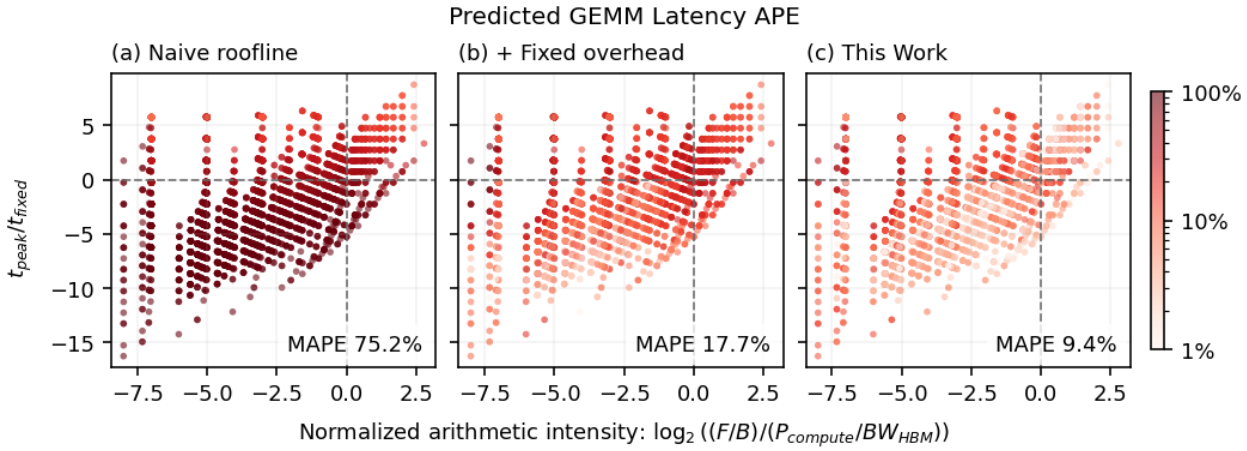

Figure 6.1. Latency ablation for the same held-out BF16 GEMMs in all panels on an A100 PCIe 40GB at 900 MHz. The evaluation uses the documented hardware inputs in Table 5.1, except for its 40GB HBM capacity. Panel (a) is a peak Roofline estimate; panel (b) adds the profiled fixed kernel overhead; panel (c) is the complete execution-aware model. Each point is one GEMM. The horizontal coordinate is arithmetic intensity normalized to the compute–HBM Roofline crossover, and the vertical coordinate is $\log_2(t_{\text{peak}}/t_{\text{fixed}})$, where t_{peak} is the naive peak-Roofline estimate; negative values identify kernels whose peak-Roofline time is below the fixed-overhead scale. Point color gives absolute percentage error on the 1–100% logarithmic scale; each panel reports MAPE over the same cases.

Figure 6.2 makes the same point for energy. The static-energy-only model has 91.0% MAPE. Adding only HBM and then compute activity reduces that error to 62.4% and 54.1%, respectively, but omits the execution-dependent time and the remaining memory interfaces. The complete model reaches 11.4% MAPE without a per-workload energy correction.

6.2 Single-GPU Inference Validation

Single-GPU validation uses BF16 on the A100 PCIe 80GB system in Table 5.2. Qwen 1.5 7B and Qwen 2.5 7B use batch sizes 1, 4, and 8; input lengths 128, 512, and 2048; and output lengths 32,

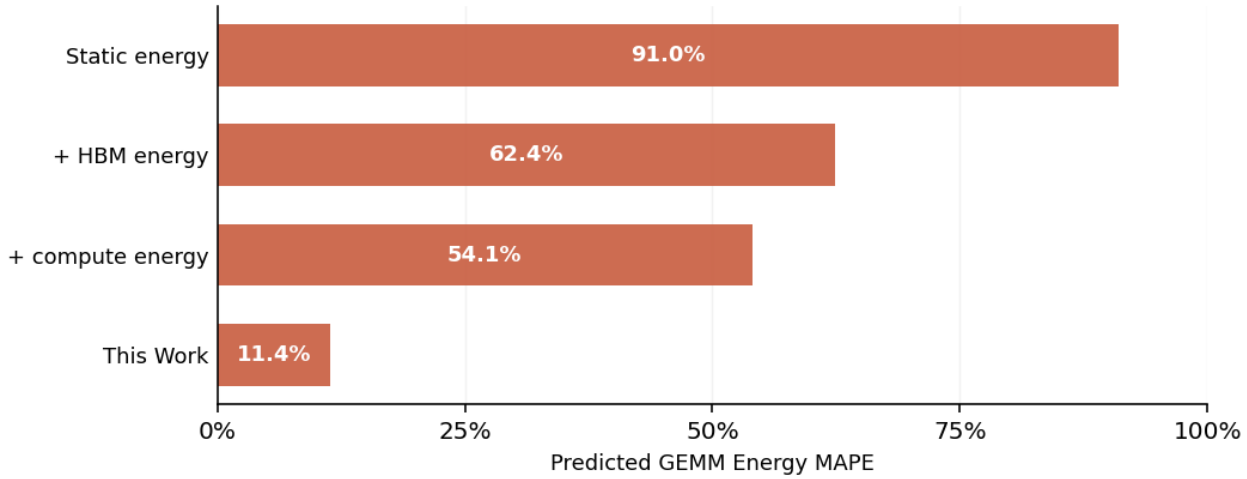

Figure 6.2. Cumulative energy ablation for the same held-out BF16 GEMMs as Figure 6.1. Each bar reports MAPE over the same cases. "Static energy" charges only idle static power over predicted active time; "+ HBM energy" additionally charges unique operand bytes with the same e_{HBM} used by the complete model; and "+ compute energy" additionally charges logical useful FLOPs with the same compute-event coefficient. "This Work" uses the complete execution-aware prediction: fixed and wave-aware latency, issued compute work, and shared-memory, L2, and HBM traffic. The progressive bars show the error left after each incomplete energy-accounting choice.

128, and 512. Optimum-Benchmark defines prefill as prompt processing plus first-token generation and decode as the remaining output-token iterations [13]. Solid bars are predictions, hatched bars are measurements, and colored segments separate the two phases.

Figure 6.3 shows that, across these configurations, the model preserves the relative effects of batch size, prompt length, generation length, and model architecture without fitting an end-to-end correction. The companion latency validation appears in Figure A1.

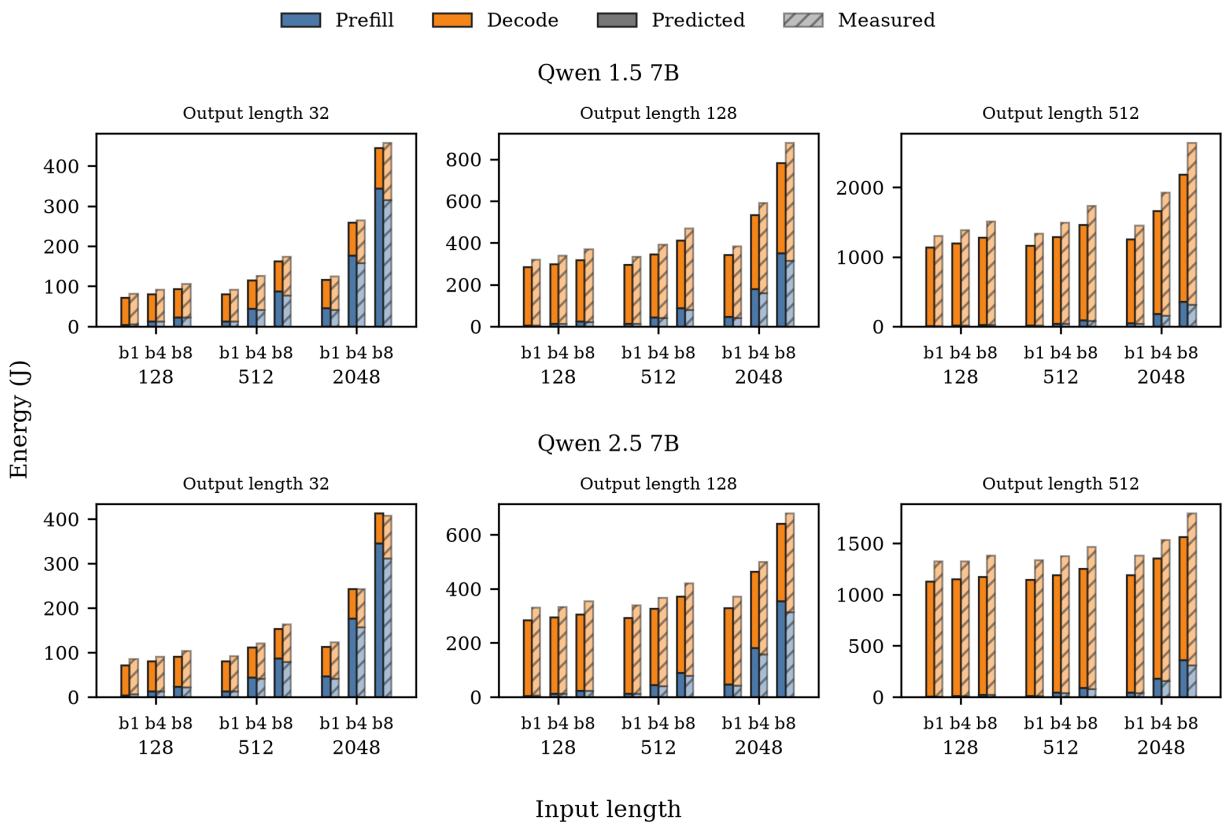

Figure 6.3. Single-GPU BF16 end-to-end GPU-energy validation for Qwen 1.5 7B and Qwen 2.5 7B on one A100 PCIe 80GB. The 54 configurations cover batch sizes 1/4/8, input lengths 128/512/2048, and output lengths 32/128/512. Prefill includes first-token generation; decode contains later tokens. Solid bars are predicted and hatched bars are measured. MAPE is 13.5%.

Batch-one Llama 3.2 1B/3B, Llama 2 7B/13B, and Llama 3 8B use 32–1024 prefill input tokens and one generated token. Decode uses a 128-token input and 4–128 output tokens; the reported decode component excludes prefill and the first generated token. Figures 6.4 and 6.5 report the corresponding prefill and cumulative-decode energy validation.

The larger short-prefill discrepancy motivates the counter attribution below. Short requests contain relatively little modeled activity, so launch, framework, synchronization, and pipeline fill/drain costs represent a larger share of measured energy. Companion latency plots appear in Figures A2 and A3.

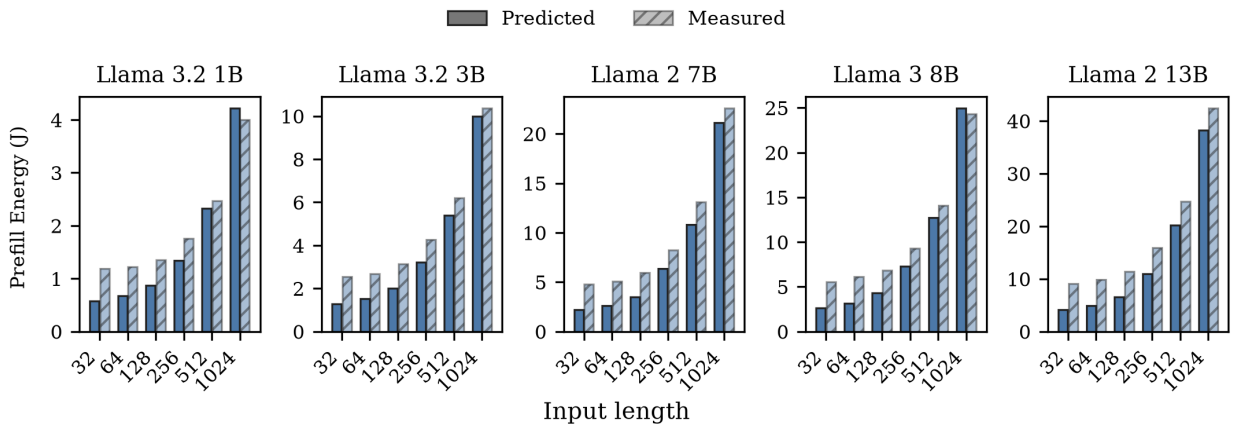

Figure 6.4. Single-GPU BF16 prefill GPU-energy validation on one A100 PCIe 80GB for batch-one Llama 3.2 1B/3B, Llama 2 7B/13B, and Llama 3 8B with 32–1024 input tokens and one generated token. Each bar contains prompt processing and first-token energy. MAPE is 31.5%.

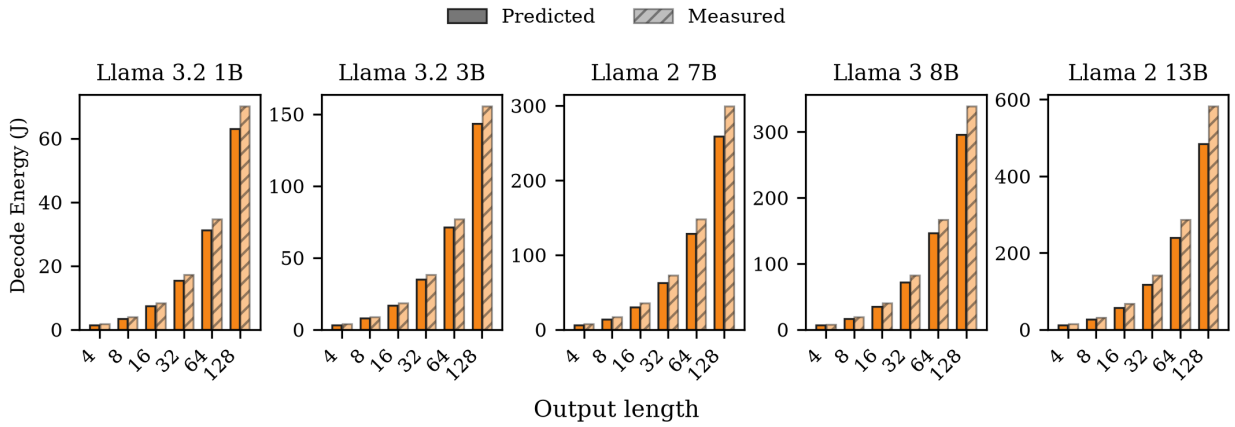

Figure 6.5. Single-GPU BF16 cumulative decode GPU-energy validation on one A100 PCIe 80GB for the same batch-one Llama models. Requests use 128 input tokens and 4–128 output tokens. Prefill and first-token energy are excluded; MAPE is 17.3%.

6.3 Activity-Driven Energy Validation

The model derives dynamic energy from predicted compute and memory activity and charges idle static power separately. The corresponding measured dynamic-energy reference is

$$E_{\text{dyn,meas}} = E_{\text{meas}} - P_{\text{idle}} t_{\text{meas}}, \tag{6.1}$$

where P_{idle} is the calibrated one-GPU idle static power and t_{meas} is measured phase runtime. This subtraction prevents longer execution from being interpreted as additional modeled activity.

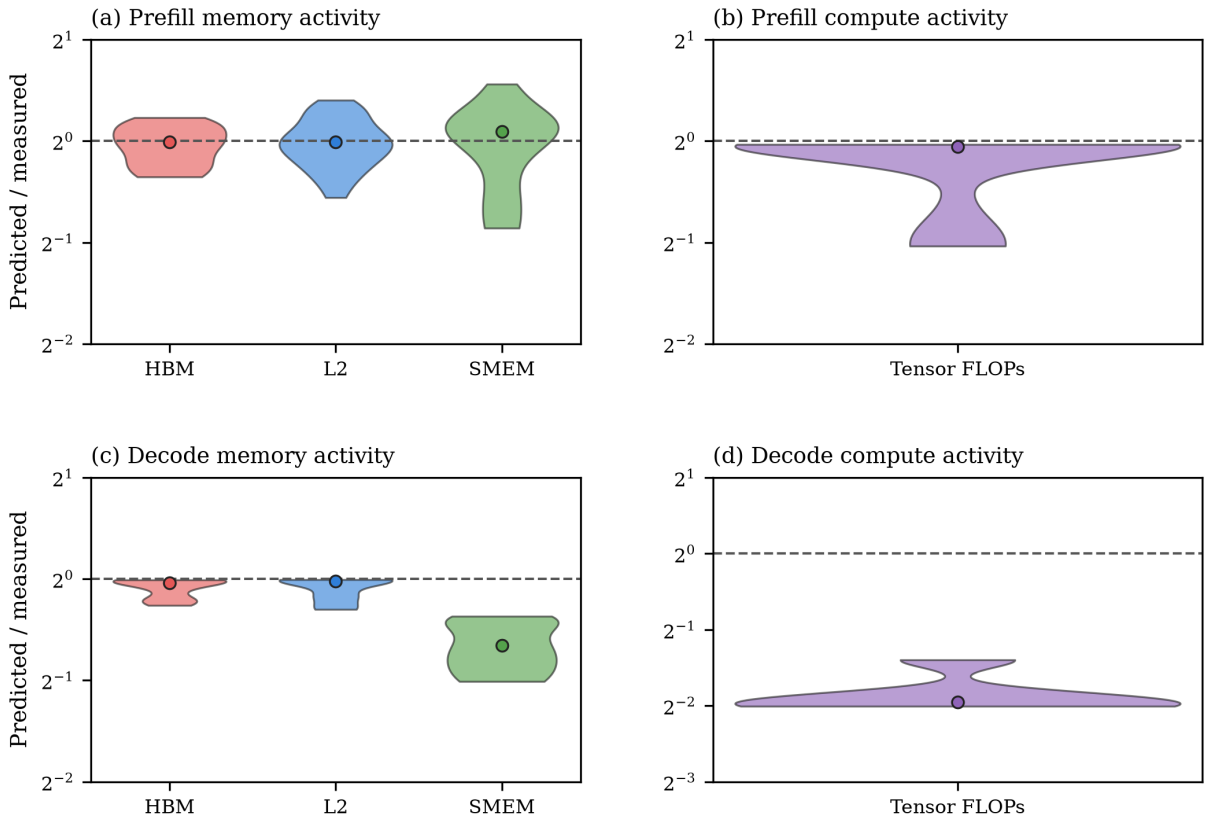

Figure 6.6. Activity validation against phase-resolved NCU counters for the BF16 batch-one Llama configurations on one A100 PCIe 80GB. Prefill uses 32–1024 input tokens and one output; decode uses 128 inputs and 4–128 outputs. Prefill HBM/L2/Tensor-FLOP MAPE is 10.6/12.8/12.3%; decode MAPE is 5.9/6.5/72.0%.

For the same BF16 batch-one Llama configurations on one A100 PCIe 80GB, Nsight Compute (NCU) supplies phase-resolved traffic and issued instructions. HBM and L2 traffic sum read and write sectors and use the hardware 32-byte sector size. Shared-memory traffic includes explicit shared loads, stores, and matrix-load bytes while avoiding traffic already charged on the global path. Issued Tensor Core FLOPs multiply the measured MMA instruction count by its complete instruction tile [38]. Figure 6.6 compares these predicted activity quantities with the NCU measurements.

The HBM and L2 agreement in Figure 6.6 is important because these quantities are predicted from the schedule rather than inferred from total measured energy. Issued-FLOP error reflects a different limitation: framework-dependent kernel heuristics choose tilings for skinny decode GEMMs, and each MMA instruction executes its complete tile even when boundary lanes do not contribute useful

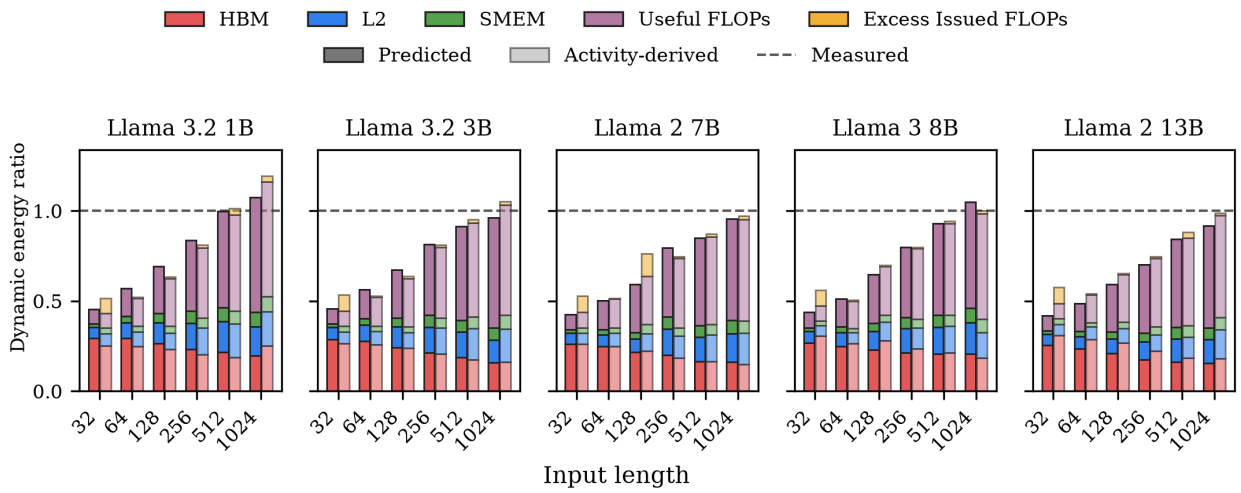

Figure 6.7. Normalized BF16 batch-one Llama prefill dynamic energy on one A100 PCIe 80GB for 32–1024 input tokens and one output. Modeled and activity-derived components are compared with measured dynamic energy; Tensor energy separates logical work from issued-MMA padding.

work.

Together, Figures 6.7 and 6.8 expose the model boundary. Framework and synchronization residuals are large relative to short-prefill activity. As prefill or cumulative decode work grows, activity-derived energy accounts for more of measured dynamic energy without a shape-specific fit.

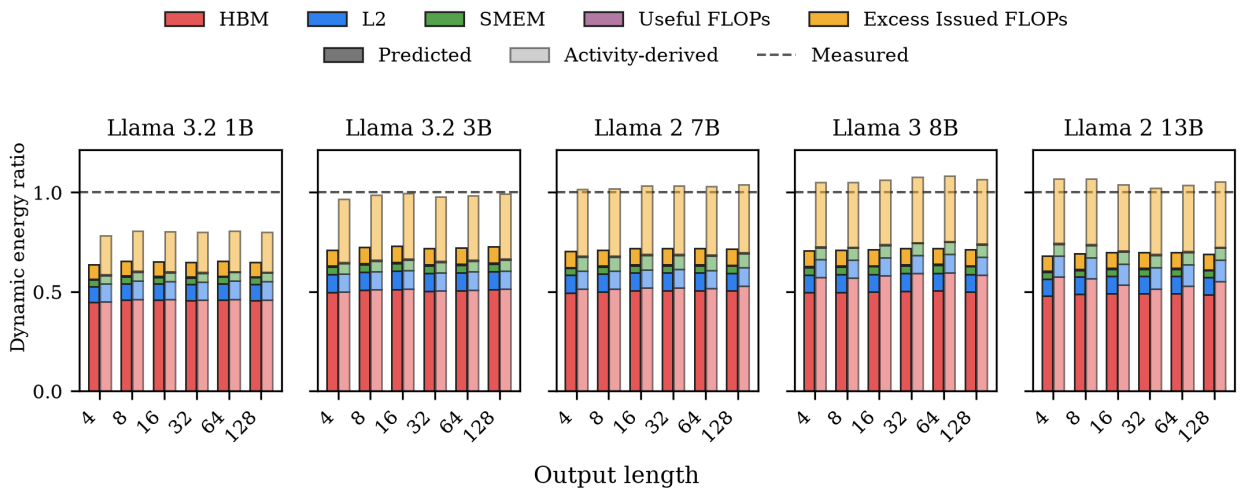

Figure 6.8. Normalized BF16 batch-one Llama cumulative-decode dynamic energy on one A100 PCIe 80GB for 128 input tokens and 4–128 output tokens. Prefill is excluded, and Tensor energy separates logical work from issued-MMA padding.

6.4 Two- and Eight-GPU Full-Workload Validation on A100 SXM4 80GB

This experiment tests whether independently estimated local operations and calibrated collectives compose into end-to-end distributed inference. Held-out BF16 Qwen 2.5 72B requests run tensor parallelism 2 or 8 on two- and eight-GPU NVSwitch-based A100 SXM4 80GB systems. Each panel labels the evaluated batch, input, and output lengths. Predicted latency is the scheduled graph makespan defined in Section 4.5. Predicted energy follows Equation (2.10): it sums the canonical $E^{(g)}_{k,\mathrm{local}}$ term over every executed GPU-local kernel and adds each system-wide $E_{c,\mathrm{comm}}$ collective term once. This is the same GPU-only boundary used by the CodeCarbon measurement. CodeCarbon is restricted to the participating GPU identifiers, and the reference sums its device-level GPU-energy values rather than the CPU- and RAM-inclusive total-energy field [22, 41]. Figure 6.9 compares the resulting end-to-end energy prediction with measurement. The companion latency validation appears in the appendix.

The eight-GPU decode uses its observed direct reduction schedule rather than applying a 14-transfer Ring all-reduce latency to small reductions. This distinction preserves the measured scale without hiding the mismatch in a workload-level fit. Remaining framework, synchronization, and unsupported fused/runtime work is retained as an end-to-end residual.

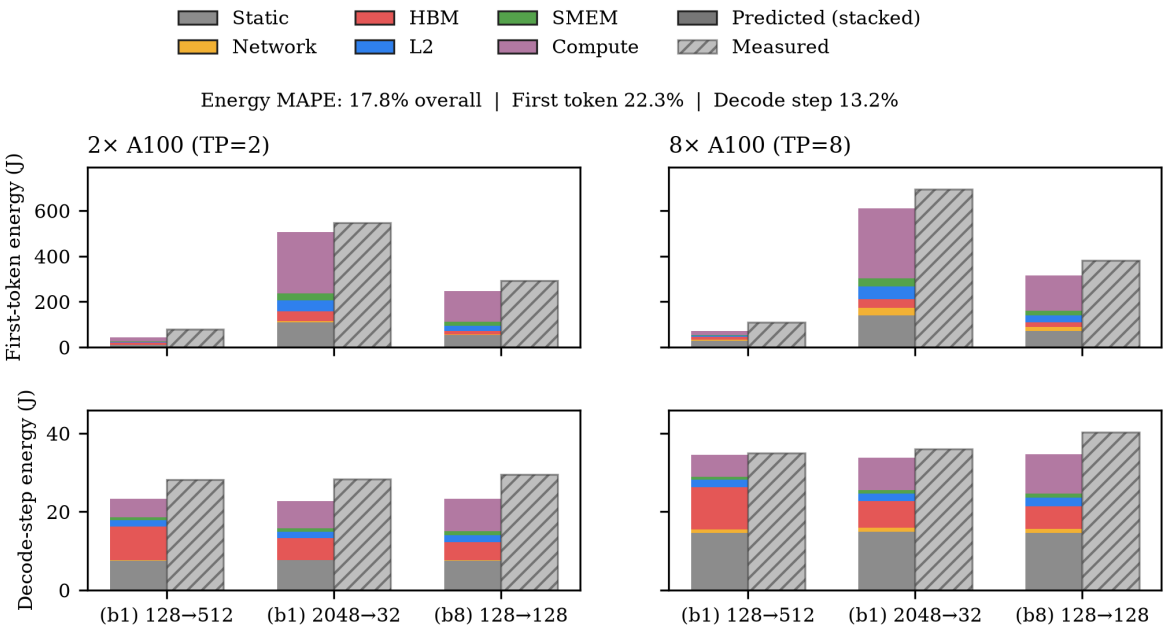

Figure 6.9. BF16 Qwen 2.5 72B system-energy validation for the same labeled batch, input, and output lengths on two and eight A100 SXM4 80GB GPUs. CodeCarbon device-level GPU energy sums all participating GPUs. The prediction sums GPU-local energy and adds system-wide network energy once.

Chapter 7 Discussion and Limitations

The value of the model is comparative rather than universal: it exposes how an operation's shape and selected kernel configuration change GPU-local time, activity, and energy before the workload is deployed. The validation shows why this distinction matters. Ideal work counts alone do not capture fixed overhead, wave utilization, or per-level memory traffic. Within the supported structured-kernel scope, the resulting profiles can therefore compare local kernel choices and tensor-parallel configurations while retaining explicit residual categories.

7.1 Design-Space Analysis

The validated model can diagnose configuration and shape effects without presenting those analyses as additional validation data.

7.1.1 Latency–energy tradeoff across GEMM kernels. Five BF16 prefill GEMM shapes sweep their legal tile, pipeline, residency, warp, and shared-memory configurations. Selecting for energy rather than latency can expose kernel strategies that a latency-only selector discards.

The design-space analysis illustrates this use. A latency-oriented GEMM configuration and an energy-oriented configuration can differ because larger tiles may reduce HBM and L2 traffic while reducing occupancy or wave utilization. The selected objective therefore changes both the activity charged by the energy model and the active time over which idle static power accumulates.

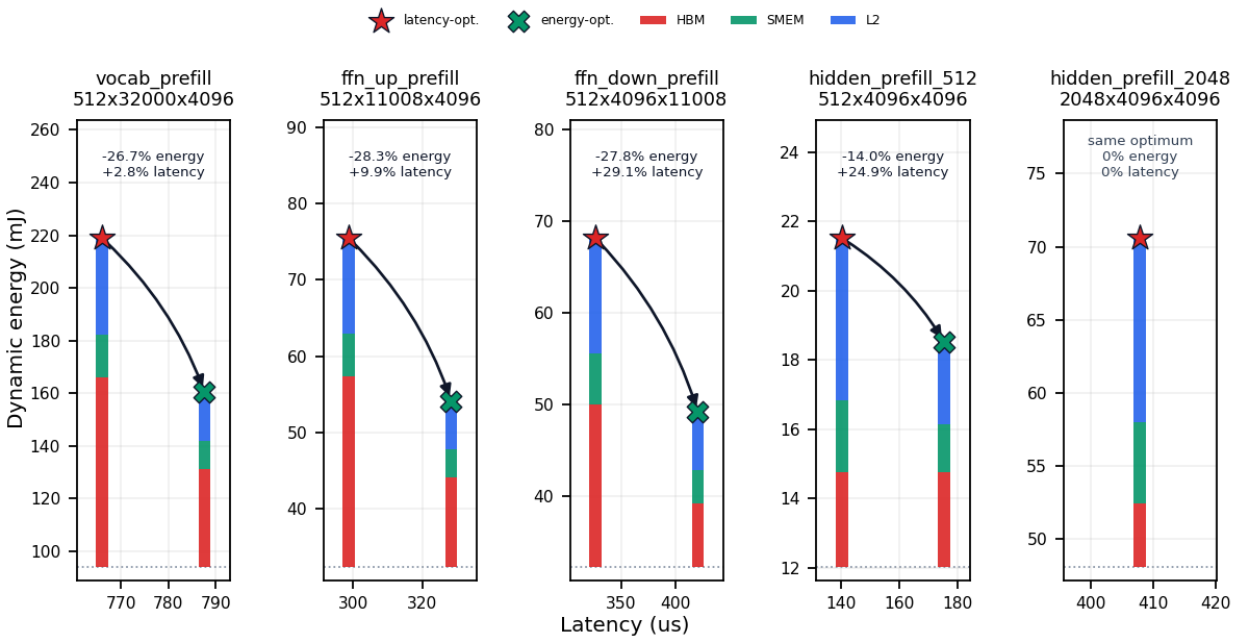

Figure 7.1. Modeled latency–energy tradeoffs among legal BF16 prefill GEMM strategies. Each panel compares the latency-optimal and energy-optimal kernels. Bar tops give dynamic energy after subtracting idle static power over active time, decomposed into HBM, L2, and shared-memory contributions.

7.1.2 Full-workload effect of energy-oriented kernel selection. The operator-level choices matter only if their effects survive composition across a request. Fifty-four BF16 prefill-only workloads compare the objectives for Llama 3.2 1B, Llama 3 8B, and Llama 2 13B; batch sizes 1/2/4; input lengths 32–1024; and one output token. The workload graph, hardware, candidate-kernel space, and non-GEMM modeling remain fixed; only the selection objective changes.

Dynamic energy includes modeled compute and memory activity but excludes idle static energy. A longer runtime therefore cannot appear as a dynamic-energy saving through the static-energy accounting.

The latency cost and dynamic-energy saving vary with model and tensor shape. A kernel can reduce memory traffic while losing occupancy or wave utilization, causing latency to rise faster than activity energy falls. The meaningful decision is consequently made after composing these local effects across a request, rather than from an isolated operator.

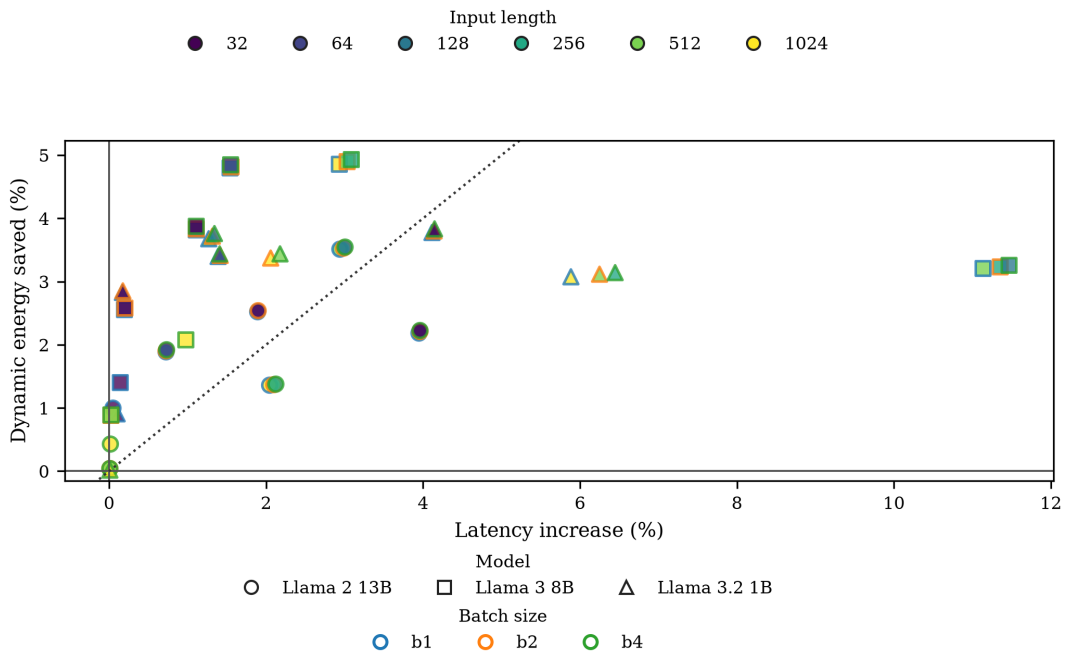

Figure 7.2. Request-level effect of selecting GEMM kernels for dynamic energy rather than latency across 54 BF16 Llama prefill workloads. Axes report latency increase and dynamic energy saved relative to latency-oriented selection. The dotted $y = x$ line is a visual tradeoff guide.

7.2 Scope, Transfer, and Next Steps

GPU-local predictions require structured kernels whose issued work, traffic, latency, independent chains, pipeline depth, occupancy, and wave behavior can be derived statically. Short dependent loops, hard outstanding-transaction limits, bank conflicts, replays, barriers, cache conflicts, and instruction-mix pressure are not identifiable from the present analytical inputs. They should remain explicit residuals rather than be absorbed into effective rates.

The network calibration uses Ring all-reduce on the measured topology, protocol, GPU counts, and operating point. The per-link activity formulation is topology-independent: an explicit schedule on another topology using the same link technology can supply its link payloads and busy times to the same energy framework. This transfer has not yet been validated across collective algorithms, protocols, fabrics, GPU counts, or fabric generations. Collective traffic and busy-time accounting also do not by themselves establish complete distributed-inference accuracy; framework activity, CPU work, graph compilation, and unsupported operations remain end-to-end residuals.

Useful next steps are to validate the supported fused-attention path, extend operation coverage only when its execution structure is explicit, and broaden multi-GPU workload validation across topologies and parallelization strategies using the same held-out discipline used for kernels and collectives.

Chapter 8 Conclusion

The GPU energy model separates GPU-local static, issued-compute, and memory-traffic energy from a contributed system-wide network term. The execution-aware model supplies kernel time for static energy and whole-kernel issued work and traffic for dynamic energy. RAPID-LLM embeds the resulting operation profiles in an LLM workload graph and extends them with topology-aware network operations. The calibration boundary remains deliberately narrow, allowing measured errors to be attributed to explicit modeled resources or stated residuals. The two- and eight-device Qwen evaluations demonstrate that these independently modeled local and network components compose to 9–21% end-to-end latency MAPE and 14–20% system-energy MAPE without workload-level fitting. Remaining work includes the planned fused-attention validation and broader multi-GPU workload evaluation across GPU counts, topologies, and collective implementations under the same held-out protocol.

Appendices

Appendix A Detailed execution-aware scheduling model

This appendix gives the resource- and phase-level derivations summarized in Chapter 3. These details establish how the predictor obtains kernel time and whole-kernel activity; the energy model consumes the resulting profile as described in Chapter 2.

A.1 Kernel Configuration and Execution Geometry

For GEMM, `LocalOp` attributes identify a concrete CUTLASS configuration. CUTLASS is NVIDIA's C++ template library for composing tiled CUDA kernels from architecture-, data-type-, instruction-, and thread-block-level building blocks [9]. Generated kernel names retain these choices. For example, in `cutlass_80_tensorop_bf16_s16816gemm_bf16_64x256_32x4_nn_align8`, `80` identifies the target architecture, `tensorop` and `bf16` identify the execution path and data type, `s16816gemm` identifies a $16 \times 8 \times 16$ MMA instruction, `64x256` identifies the CTA output tile, `32x4` identifies the K tile and pipeline depth, and `nn` and `align8` record operand layouts and alignment. The predictor combines these tokens with the measured grid and block dimensions to recover the CTA tile, MMA shape, pipeline stages, thread/warp count, and compatible warp tiling. The tensor shape alone does not select these properties. This interpretation follows CUTLASS's hierarchical GEMM decomposition and official Ampere profiler examples [34, 31].

For kernel-level validation, the recorded kernel name, grid, and block dimensions select the measured configuration directly. Otherwise, the predictor tries a catalog of configurations known for Ampere and selects the one with the fastest predicted runtime.

Within a CTA, 32-thread warps own smaller output tiles. Ampere divides an SM into four

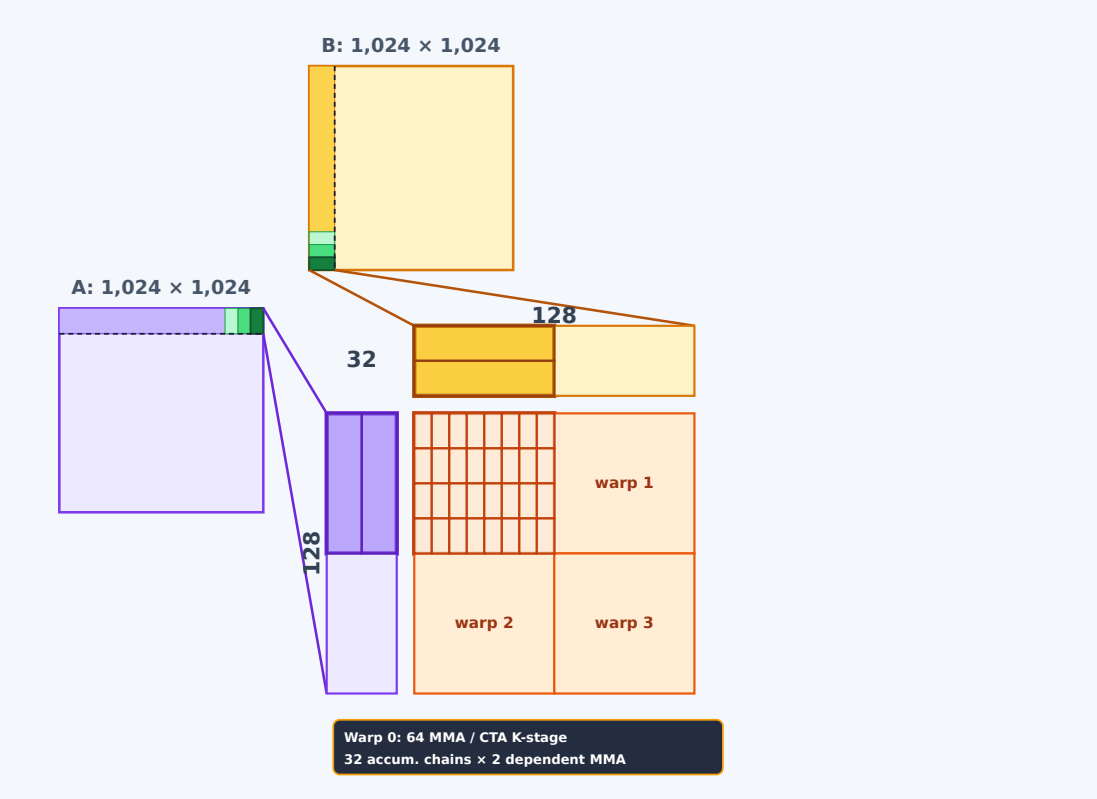

Figure A1. GEMM-to-MMA execution geometry for a $1024 \times 1024 \times 1024$ GEMM using the `sm80_128x128x32` CTA configuration. Highlighted operand bands feed one $128 \times 128 \times 32$ CTA stage, whose output tile is divided among four $64 \times 64 \times 32$ warp tiles. Warp 0 is shown in the hovered state: its tile is decomposed into $16 \times 8 \times 16$ MMA instructions, and the popup reports 64 MMA instructions and 32 two-instruction accumulator chains per CTA K-stage.

SM sub-partitions (SMSPs), each with instruction-issue, arithmetic, Tensor Core, and memory resources [25]. An accumulator chain is the sequence of MMA instructions updating the same accumulator registers. Each MMA consumes its predecessor's partial sum, while chains for disjoint output fragments can be interleaved to hide Tensor Core latency.

A.2 GEMM Local Service

The GEMM wave has a local path and a global path. The local path begins after operands reach shared memory and covers fragment loads and dependent MMA instructions. The global path moves operand traffic through HBM and L2 into the CTA's shared-memory buffers. They expose different independent work and are evaluated separately before legal overlap is applied.

For warp tile $(M_{\text{warp}}, N_{\text{warp}}, K_{\text{warp}})$ and MMA shape $(M_{\text{mma}}, N_{\text{mma}}, K_{\text{mma}})$, one MMA issues

$$f_{\text{mma}} = 2M_{\text{mma}}N_{\text{mma}}K_{\text{mma}} \tag{A.1}$$

FLOPs, and the warp exposes

$$n_{MN} = \left\lceil \frac{M_{\text{warp}}}{M_{\text{mma}}} \right\rceil \left\lceil \frac{N_{\text{warp}}}{N_{\text{mma}}} \right\rceil \tag{A.2}$$

independent accumulator chains. For $n_{\text{warp},j}$ resident warps on SMSP j,

$$\bar{\pi}^{\text{eff}}_{\text{TC},j} = \min\left(\bar{\pi}^{\text{peak}}_{\text{TC},j}, \frac{n_{\text{warp},j}n_{MN}f_{\text{mma}}}{C^{\text{lat}}_{\text{TC}}}\right). \tag{A.3}$$

Additional MMA steps along K add dependent work but do not create more accumulator chains.

Within one CTA K-stage, a warp advances in K_{warp}-deep K-groups. For input element width b bytes, shared-memory traffic per resident warp and K-group is

$$\mathcal{T}_{\text{SMEM,warp}} = (M_{\text{warp}} + N_{\text{warp}})K_{\text{warp}}b, \tag{A.4}$$

giving

$$\bar{\beta}^{\text{eff}}_{\text{SMEM,SM}} = \min\left(\bar{\beta}^{\text{peak}}_{\text{SMEM,SM}}, \frac{n_{\text{warp,SM}}\mathcal{T}_{\text{SMEM,warp}}}{C^{\text{lat}}_{\text{SMEM}}}\right). \tag{A.5}$$

Dividing issued work and traffic by these effective rates gives C_{TC} and C_{SMEM}. A finite MMA sequence also has a configuration-derived fill/drain correction $C_{\text{MMA,fd}}$ for its terminal instruction group, so the finite local path is

$$C_{\text{local,finite}} = \max(C_{\text{TC}}, C_{\text{SMEM}}) + C_{\text{MMA,fd}}. \tag{A.6}$$

A.3 GEMM L2 and HBM Transfer Windows

A transfer window contains the consecutive K-stages that the kernel's shared-memory buffers can keep in flight. Let S_{pipe} be the pipeline-stage count parsed from the kernel name and S_{total} the total number of K-stages. The number of windows is

$$G_{\mathrm{mem}} = \left\lceil \frac{S_{\mathrm{total}}}{S_{\mathrm{pipe}}} \right\rceil, \tag{A.7}$$

and the final window contains

$$S_{\mathrm{last}} = S_{\mathrm{total}} - (G_{\mathrm{mem}} - 1)S_{\mathrm{pipe}} \tag{A.8}$$

stage equivalents. Active CTAs are the resident CTAs summed over all busy SMs in the wave. Their requested L2 operand traffic is

$$\begin{aligned} \mathcal{T}_{\mathrm{L2,full}} &= n_{\mathrm{CTA,active}} S_{\mathrm{pipe}} b K_{\mathrm{CTA}} (M_{\mathrm{CTA}} + N_{\mathrm{CTA}}), \\ \mathcal{T}_{\mathrm{L2,last}} &= n_{\mathrm{CTA,active}} S_{\mathrm{last}} b K_{\mathrm{CTA}} (M_{\mathrm{CTA}} + N_{\mathrm{CTA}}). \end{aligned} \tag{A.9}$$

HBM traffic per window cannot be determined analytically because it depends on evolving L2 state and CTA execution order. The model first estimates aggregate input traffic $\widehat{\mathcal{T}}_{\mathrm{HBM,input}}$ over the kernel, then assumes uniform traffic per CTA-stage equivalent:

$$\begin{aligned} \widehat{\mathcal{T}}_{\mathrm{HBM,full}} &= \frac{n_{\mathrm{CTA,active}} S_{\mathrm{pipe}}}{n_{\mathrm{CTA}} S_{\mathrm{total}}} \widehat{\mathcal{T}}_{\mathrm{HBM,input}}, \\ \widehat{\mathcal{T}}_{\mathrm{HBM,last}} &= \frac{n_{\mathrm{CTA,active}} S_{\mathrm{last}}}{n_{\mathrm{CTA}} S_{\mathrm{total}}} \widehat{\mathcal{T}}_{\mathrm{HBM,input}}. \end{aligned} \tag{A.10}$$

This is an even-distribution assumption rather than a per-window cache-state prediction. Output traffic is charged to the epilogue. The hat is omitted below when these assigned quantities enter the shared service equations.

For $\ell \in \{\mathrm{L2}, \mathrm{HBM}\}$,

$$\begin{aligned} \bar{\beta}^{\mathrm{eff}}_{\ell,\mathrm{full}} &= \min\left(\bar{\beta}^{\mathrm{peak}}_{\ell}, \frac{\mathcal{T}_{\ell,\mathrm{full}}}{C^{\mathrm{lat}}_{\ell}}\right), \\ C_{\ell,\mathrm{full}} &= \frac{\mathcal{T}_{\ell,\mathrm{full}}}{\bar{\beta}^{\mathrm{eff}}_{\ell,\mathrm{full}}}, \\ C_{\ell,\mathrm{last}} &= \frac{\mathcal{T}_{\ell,\mathrm{last}}}{\bar{\beta}^{\mathrm{eff}}_{\ell,\mathrm{full}}}. \end{aligned} \tag{A.11}$$

The final window inherits the persistent pipeline's full-window rate. Total service is

$$C_{\ell} = (G_{\mathrm{mem}} - 1)C_{\ell,\mathrm{full}} + C_{\ell,\mathrm{last}}. \tag{A.12}$$

If all windows sustain peak bandwidth, this simplifies to

$$C^{\mathrm{peak}}_{\ell} = \frac{(G_{\mathrm{mem}} - 1)\mathcal{T}_{\ell,\mathrm{full}} + \mathcal{T}_{\ell,\mathrm{last}}}{\bar{\beta}^{\mathrm{peak}}_{\ell}} = \frac{\mathcal{T}_{\ell,\mathrm{total}}}{\bar{\beta}^{\mathrm{peak}}_{\ell}}. \tag{A.13}$$

A.4 GEMM Phase Composition

The Roofline maximum applies only to the body, where local computation can overlap operand movement. The prologue and epilogue are memory-only intervals that serialize outside the body; placing either inside the Roofline maximum would incorrectly allow computation to hide required boundary traffic.

The first input window must complete before MMA work begins:

$$C_{\mathrm{prologue}} = \max(C_{\mathrm{L2},0}, C_{\mathrm{HBM},0}), \tag{A.14}$$

$$C_{\mathrm{memory,rem}} = \max(C_{\mathrm{L2,rem}}, C_{\mathrm{HBM,rem}}). \tag{A.15}$$

Remaining memory service overlaps local execution, but the final local window still depends on the final operands:

$$C_{\mathrm{body}} = \max(C_{\mathrm{local,finite}}, C_{\mathrm{memory,rem}} + C_{\mathrm{local,last}})\,. \tag{A.16}$$

After the final MMA, the epilogue serializes accumulator-fragment handling and the final output path:

$$C_{\text{epilogue}} = C_{\text{SMEM,frag}} + C_{\text{HBM},C/D}. \tag{A.17}$$

A.5 FlashAttention Phase Composition

The main text defines the causal active CTA set in Equation (3.15). For each active CTA class, the wave first loads Q and the initial K tile. In KV iteration j, the V load overlaps the score product and online softmax, while PV overlaps the next K load:

$$C_{A,j} = \max(C_{V,j}, C_{QK,j} + C_{S,j}), \tag{A.18}$$

$$C_{B,j} = \max(C_{PV,j}, C_{K,j+1}). \tag{A.19}$$

For the final iteration, $C_{B,J_w-1} = C_{PV,J_w-1}$. Wave duration is

$$C^{\text{FA}}_{\text{wave}} = C_Q + C_{K,0} + \sum_{j=0}^{J_w-1} (C_{A,j} + C_{B,j}) + C_{\text{epilogue}}. \tag{A.20}$$

The epilogue accounts for output normalization, optional log-sum-exp work, configured SMEM staging, and output stores.

A.6 Worked GEMM Examples

Two measured 900 MHz A100 PCIe 40GB BF16 examples use batch $B = 128$, $M = N = 1024$, and $K = 64$ or 2048. They share an output shape but have sharply different reduction depth. Both examples use peak SMEM, L2, and HBM bandwidths of 14.8, 4.3, and 1.935 TB/s, respectively. At

900 MHz, these hardware inputs give

$$\begin{aligned}
\bar{\pi}^{\text{peak}}_{\text{TC},j} &= 512 \text{ FLOP/cycle/SMSP}, \\
\Pi^{\text{peak}}_{\text{TC}} &= 512(4)(108) = 221{,}184 \text{ FLOP/cycle}, \\
\bar{\beta}^{\text{peak}}_{\text{SMEM,SM}} &= \frac{14.8 \times 10^{12}}{108(0.9 \times 10^{9})} = 152.26 \text{ B/cycle/SM}, \\
\bar{\beta}^{\text{peak}}_{\text{L2}} &= 4{,}777.78 \text{ B/cycle}, \\
\bar{\beta}^{\text{peak}}_{\text{HBM}} &= 2{,}150 \text{ B/cycle}.
\end{aligned} \tag{A.21}$$

Both selected configurations use a 64×64 warp tile with the $16 \times 8 \times 16$ MMA shape. They therefore expose $n_{MN} = (64/16)(64/8) = 32$ accumulator chains. Evaluating the terminal MMA group for either selected configuration gives $C_{\text{MMA,fd}} = 256$ fill/drain cycles per wave. The kernel-level methodology supplies the independently calibrated $C_{\text{fixed}} = 23{,}181$ cycles.

The diagnostics use Equations (3.7) and (3.8), the local-service construction in Equations (A.1) to (A.6), the window traffic in Equations (A.9) and (A.10), and the output path in Equation (A.17). Full and tail waves are composed using Equation (3.12).

For $K = 64$, the grid contains $128(16)(16) = 32{,}768$ CTAs. Eight CTAs can reside on each SM, so a full wave contains $108(8) = 864$ CTAs and the grid has 37 full waves followed by an 800-CTA tail wave. The full-wave first input window contains 13.5 MiB of L2 traffic and 864 KiB of HBM traffic. The L2 service time is 2,962.84 cycles, while the finite HBM window takes its 566-cycle latency floor; L2 therefore determines the prologue. On the busiest SM, Tensor Core service takes 2,048 cycles and SMEM operand service takes 860.82 cycles, giving a $2{,}048 + 256 = 2{,}304$-cycle body. The output path contains 64 KiB of SMEM fragment traffic per busy SM and 6.75 MiB of HBM traffic per full wave. Thus

$$\begin{aligned}
C_{\text{prologue,full}} &= \max\left(\frac{13.5(2^{20})}{4{,}777.78}, 566\right) = 2{,}962.84, \\
C_{\text{body,full}} &= \max(2{,}048, 860.82) + 256 = 2{,}304, \\
C_{\text{epilogue,full}} &= \frac{64(2^{10})}{152.26} + \frac{6.75(2^{20})}{2{,}150} = 3{,}722.45.
\end{aligned} \tag{A.22}$$

The 800-CTA tail similarly gives 2,743.37 prologue cycles and 3,478.60 epilogue cycles; its busiest SM still executes eight CTAs, so its body remains 2,304 cycles. The execution-aware wave accounting then gives

$$\frac{37(8{,}989.29) + 8{,}525.97 + 23{,}181}{0.9 \times 10^9} = 0.405 \text{ ms}, \tag{A.23}$$

whereas peak Roofline gives

$$\max\left(\frac{F_{\text{useful}}}{199.066 \text{ TFLOP/s}}, \frac{\mathcal{T}_{\text{HBM}}}{1.935 \text{ TB/s}}\right) = 0.156 \text{ ms}. \tag{A.24}$$

For $K = 2048$, the grid contains $128(4)(8) = 4{,}096$ CTAs. One CTA resides on each SM, again producing 37 full waves and a tail, now with 100 CTAs. A three-stage first window carries 15.1875 MiB through L2 and 2.53125 MiB through HBM, for respective service times of 3,333.19 and 1,234.52 cycles. Tensor Core and SMEM service require 65,536 and 27,546.37 cycles. After the first window, the L2 path requires another 32,220.85 cycles, and its last two-stage window enables the final 4,096 Tensor Core cycles. This 36,316.85-cycle dependency path remains shorter than the finite Tensor Core path, $65{,}536 + 256 = 65{,}792$ cycles. The output tile gives the same full-wave epilogue as the $K = 64$ case. Hence

$$\begin{aligned} C_{\text{prologue,full}} &= 3{,}333.19, \\ C_{\text{local,finite}} &= \max(65{,}536, 27{,}546.37) + 256 = 65{,}792, \\ C_{\text{body,full}} &= \max(65{,}792, 36{,}316.85) = 65{,}792, \\ C_{\text{epilogue,full}} &= 3{,}722.45. \end{aligned} \tag{A.25}$$

The corresponding tail-wave phases are 3,086.29, 65,792, and 3,478.60 cycles. The execution-aware accounting gives

$$\frac{37(72{,}847.64) + 72{,}356.89 + 23{,}181}{0.9 \times 10^9} = 3.101 \text{ ms}. \tag{A.26}$$

Table A.1. Measured and predicted A100 PCIe 40GB BF16 GEMMs.

Quantity	$K = 64$	$K = 2048$
CTA tile	$64\times64\times32$	$256\times128\times64$
Grid / stages / pipeline	16×16 / 2 / 2	4×8 / 32 / 3
Useful/issued FLOPs	17.18 GF	549.76 GF
HBM traffic	288 MiB	1.25 GiB
L2 traffic	512 MiB	6.00 GiB
SMEM traffic	768 MiB	16.25 GiB
Full-wave effective service rates		
Tensor Core	199.07 TFLOP/s	199.07 TFLOP/s
SMEM	14.80 TB/s	14.80 TB/s
L2	4.30 TB/s	4.30 TB/s
HBM	1.41 TB/s	1.94 TB/s
Dominant phase	Epilogue (41.4%)	TC body (90.3%)
Full-wave phase cycles		
Prologue	2,963	3,333
Body	2,304	65,792
Epilogue	3,722	3,722
Tail-wave phase cycles		
Prologue	2,743	3,086
Body	2,304	65,792
Epilogue	3,479	3,479
Peak Roofline	0.156 ms	2.762 ms
execution-aware	0.405 ms	3.101 ms
Measured	0.480 ms	3.138 ms
Roofline APE	67.49%	11.99%
execution-aware APE	15.67%	1.18%

For $K = 64$, Tensor Core, SMEM, and L2 service reach their configured peaks, while the finite input window limits HBM to 1.41 TB/s, or 72.7% of peak. The critical path is instead dominated by the 2,963-cycle L2 prologue and 3,722-cycle epilogue around a 2,304-cycle local body. For $K = 2048$, all four effective rates reach peak, but the Tensor Core path requires 65,536 cycles, exceeding 27,546 SMEM-service cycles and the 36,317-cycle memory-drain path. It produces a 65,792-cycle body occupying 90.3% of the wave.

A.6.1 Edge-padding example. Increasing the short-K example from $M = 1024$ to 1025 changes G_M from 16 to 17:

$$\eta_{\text{tile}} = \frac{F_{\text{useful}}}{F_{\text{issued}}} = \frac{1025}{17 \cdot 64} = 94.21\%. \tag{A.27}$$

The added output CTA raises issued work and traffic discontinuously; useful FLOPs alone do not capture this boundary effect.

Appendix B Latency Validation Details

The main validation presents energy estimates and their attribution to measured hardware activity. This appendix provides companion bfloat16 (BF16) latency plots for those validation workloads. The single-GPU Qwen and Llama plots use one A100 PCIe 80GB, whereas the distributed Qwen plot uses A100 SXM4 80GB systems. Together, they provide timing context for the corresponding energy results.

The single-GPU Qwen comparison covers the same model, batch-size, input-length, and output-length combinations as the corresponding energy validation.

The single-GPU Llama results separate prefill from cumulative decode. Prefill latency is more sensitive to short-run launch, framework, synchronization, and pipeline fill/drain costs, whereas cumulative decode contains more repeated modeled work.

Finally, the distributed Qwen comparison evaluates end-to-end latency on the same two- and eight-GPU tensor-parallel configurations used for system-energy validation.

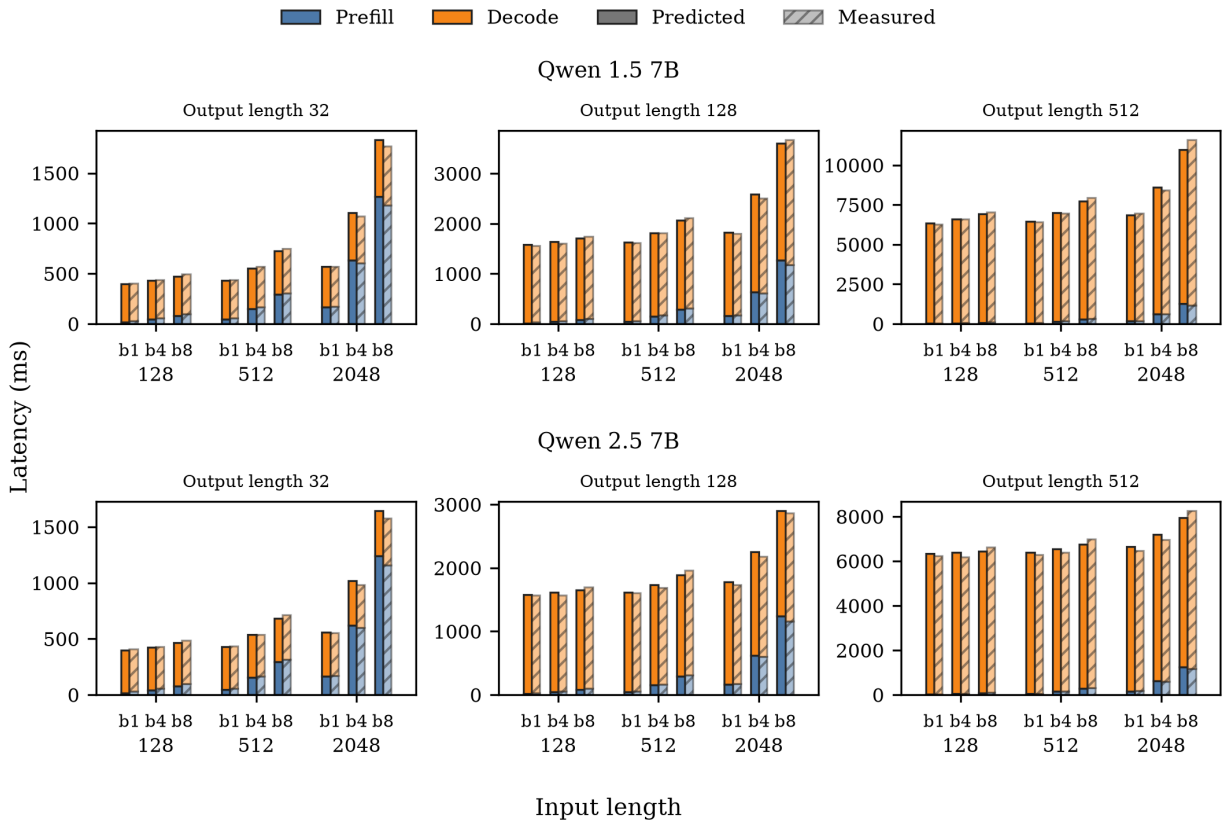

Figure A1. Single-GPU end-to-end latency validation for Qwen 1.5 7B and Qwen 2.5 7B. Each model is evaluated at batch sizes 1, 4, and 8, input lengths 128, 512, and 2048, and output lengths 32, 128, and 512 tokens. Within each input-length group, bars distinguish batch size; blue and orange segments denote prefill/first-token and post-first-token decode latency, respectively. Solid bars are predicted and hatched bars are measured. Mean absolute percentage error (MAPE) over the 54 configurations is 8.9%.

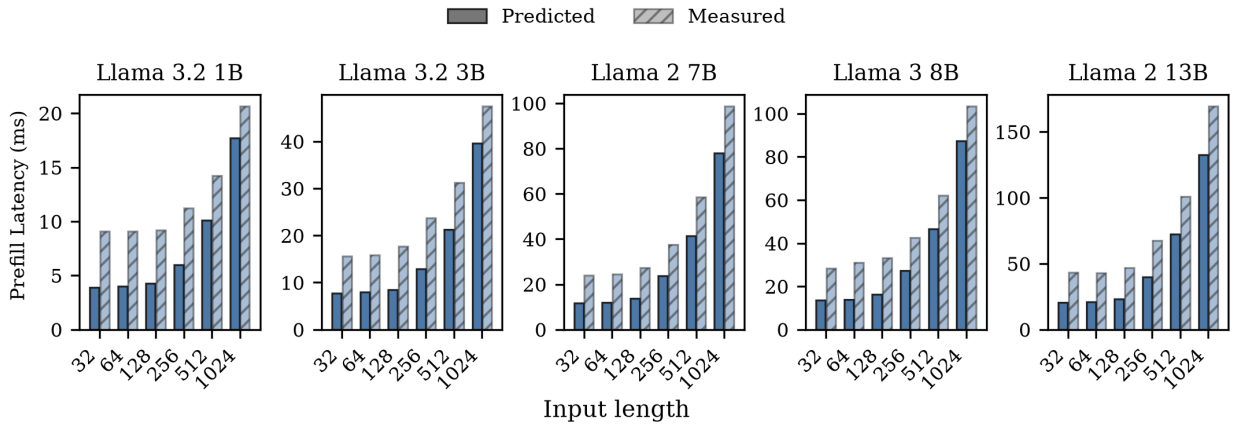

Figure A2. Single-GPU prefill latency validation for Llama 3.2 1B and 3B, Llama 2 7B and 13B, and Llama 3 8B. Batch-one requests sweep prompt length from 32 to 1024 tokens. Prefill is defined as prompt processing plus generation of the first output token; no post-first-token decode is included. Blue solid bars are predicted and blue hatched bars are measured. Latency MAPE across this sweep is 45.0%.

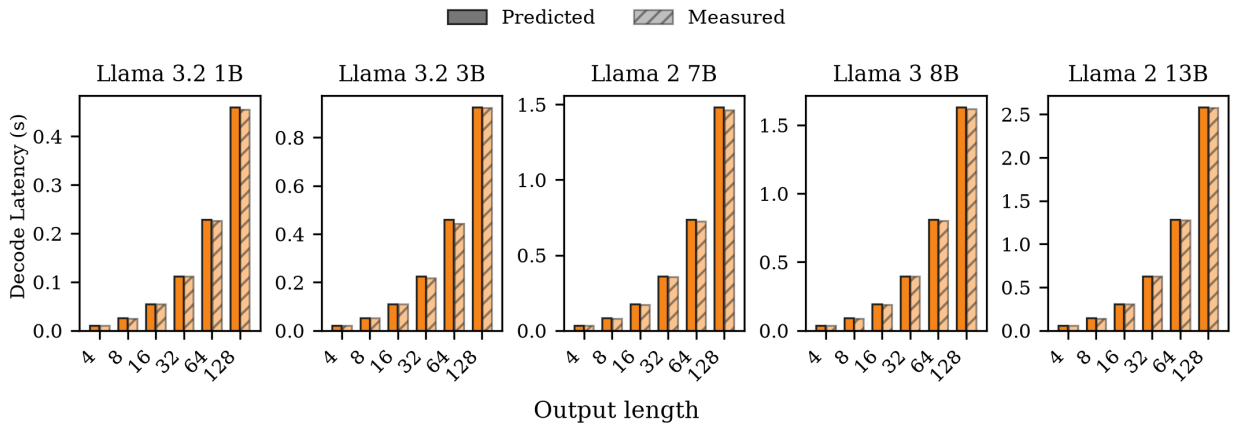

Figure A3. Single-GPU decode latency validation for the Llama suite. Batch-one requests use a 128-token prompt and sweep total output length from 4 to 128 tokens. The reported component contains only the post-first-token decode iterations; it excludes prefill and first-token generation. Orange solid bars are predicted and orange hatched bars are measured. Latency MAPE across this sweep is 13.1%.

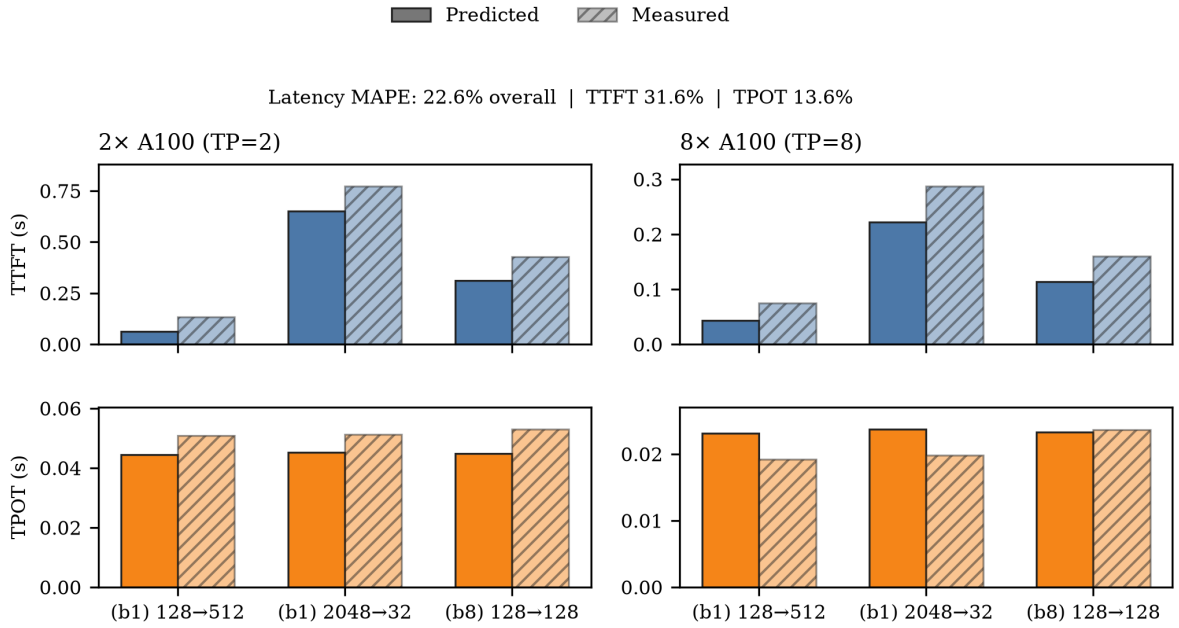

Figure A4. End-to-end BF16 Qwen 2.5 72B latency validation on NVSwitch-based A100 SXM4 80GB systems using tensor parallelism 2 and 8. Each panel reports its batch, input, and output lengths. Bars separate prefill, including first-token generation, from later decode iterations; solid bars are predicted and hatched bars are measured.

References

[1] Antepara, Oscar, et al. "Benchmark-Driven Models for Energy Analysis and Attribution of GPU-Accelerated Supercomputing." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Association for Computing Machinery, 2025-11-15, pp. 888–904. https://doi.org/10.1145/3712285.3759815.

[2] Austin, Jacob, et al. "All the Transformer Math You Need to Know." How To Scale Your Model, 2025-02-04. https://jax-ml.github.io/scaling-book/transformers/.

[3] Austin, Jacob, et al. "How to Think About GPUs." How To Scale Your Model, 2025-08-18. https://jax-ml.github.io/scaling-book/gpus/.

[4] Boughzala, Dorra, et al. "Predicting the Energy Consumption of CUDA Kernels Using SimGrid." *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020-09, pp. 191–198. https://doi.org/10.1109/SBAC-PAD49847.2020.00035.

[5] Chen, Binglu. *Performance Analysis of Infrastructure for Distributed Large Language Model Training and Inference*. University of California, Los Angeles, Ph.D. dissertation, 2026. https://www.proquest.com/LegacyDocView/DISSNUM/32737904. ProQuest ID: 32737904.

[6] Dao, Tri, et al. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness." *Advances in Neural Information Processing Systems 35*, Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2022, pp. 16344–16359. https://doi.org/10.52202/068431-1189.

[7] Dao, Tri. "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning." *International Conference on Learning Representations*, 2024, pp. 35549–35562. https://proceedings.iclr.cc/paper_files/paper/2024/file/98ed250b203d1ac6b24bbcf263e3d4a7-Paper-Conference.pdf.

[8] Delestrac, Paul, et al. "Analyzing GPU Energy Consumption in Data Movement and Storage." *2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2024-07, pp. 143–151. https://doi.org/10.1109/ASAP61560.2024.00038.

[9] "Functionality — NVIDIA CUTLASS Documentation." https://docs.nvidia.com/cutlass/4.2.1/media/docs/cpp/functionality.html#warp-level-matrix-multiply-with-tensor-cores.

[10] Henderson, Peter, et al. "Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning." *Journal of Machine Learning Research*, vol. 21, no. 248, 2020, pp. 1–43. http://jmlr.org/papers/v21/20-312.html.

[11] Hu, Zhiyi, et al. "Demystifying NCCL: An In-Depth Analysis of GPU Communication Protocols and Algorithms." *2025 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2025-08, pp. 48–59. https://doi.org/10.1109/HOTI66940.2025.00024.

[12] Huang, Yanping, et al. "GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism." *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2019-12, pp. 103–112. https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf.

[13] "Huggingface/Optimum-Benchmark: A Unified Multi-Backend Utility for Benchmarking Transformers, Timm, PEFT, Diffusers and Sentence-Transformers with Full Support of Optimum's Hardware Optimizations & Quantization Schemes.." https://github.com/huggingface/optimum-benchmark.

[14] Karfakis, George, et al. "RAPID-LLM: Resilience-Aware Performance Analysis of Infrastructure for Distributed LLM Training and Inference." 2026-08-15. https://doi.org/10.48550/arXiv.2512.19606.

[15] Khairy, Mahmoud, et al. "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling." *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020-05, pp. 473–486. https://doi.org/10.1109/ISCA45697.2020.00047.

[16] Kwon, Woosuk, et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention." *Proceedings of the 29th Symposium on Operating Systems Principles*, ACM, 2023-10-23, pp. 611–626. https://doi.org/10.1145/3600006.3613165.

[17] Lee, Kyungmi, et al. "EnergAIzer: Fast and Accurate GPU Power Estimation Framework for AI Workloads." *2026 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2026-04, pp. 435–447. https://doi.org/10.1109/ISPASS69572.2026.00049.

[18] Li, Zonghang, et al. "TPI-LLM: Serving 70B-Scale LLMs Efficiently on Low-Resource Mobile Devices." *IEEE Transactions on Services Computing*, vol. 18, no. 5, 2025-09, pp. 3321–3333. https://doi.org/10.1109/TSC.2025.3596892.

[19] Little, John D. C. "A Proof for the Queuing Formula: L = λW." *Operations Research*, vol. 9, no. 3, 1961-06, pp. 383–387. https://doi.org/10.1287/opre.9.3.383.

[20] Lucas, Jan, and Ben Juurlink. "MEMPower: Data-Aware GPU Memory Power Model." *Architecture of Computing Systems – ARCS 2019*, Springer International Publishing, 2019, pp. 195–207. https://doi.org/10.1007/978-3-030-18656-2_15.

[21] Luo, Weile, et al. "Dissecting the NVIDIA Hopper Architecture through Microbenchmarking and Multiple Level Analysis." 2025-09-04. https://doi.org/10.48550/arXiv.2501.12084.

[22] "Methodology - CodeCarbon." https://docs.codecarbon.io/3.2/explanation/methodology/.

[23] Narayanan, Deepak, et al. "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Association for Computing Machinery, 2021-11-13, pp. 1–15. https://doi.org/10.1145/3458817.3476209.

[24] Niu, Chenxu, et al. "TokenPowerBench: Benchmarking the Power Consumption of LLM Inference." *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 40, no. 38, 2026-03-14, pp. 32582–32590. https://doi.org/10.1609/aaai.v40i38.40535.

[25] NVIDIA Corporation. "NVIDIA A100 Tensor Core GPU Architecture." NVIDIA architecture whitepaper, 2020. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[26] NVIDIA Corporation. "NVIDIA Ampere GPU Architecture Tuning Guide." CUDA Toolkit Documentation, 2020. https://docs.nvidia.com/cuda/archive/11.0/ampere-tuning-guide/index.html. Compute capability 8.0 occupancy limits.

[27] NVIDIA Corporation. "Understanding the Visualization of Overhead and Latency in NVIDIA Nsight Systems." NVIDIA Technical Blog, 2020. https://forums.developer.nvidia.com/t/understanding-the-visualization-of-overhead-and-latency-in-nvidia-nsight-systems/154767/3.

[28] NVIDIA Corporation. "NVIDIA A100 Tensor Core GPU Datasheet." 2022. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/PB-10577-001_v02.pdf. A100 PCIe 80GB product specifications.

[29] NVIDIA Corporation. "CUDA C++ Best Practices Guide." 2026. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[30] NVIDIA Corporation. "CUDA C++ Programming Guide." 2026. https://docs.nvidia.com/cuda/cuda-programming-guide/index.html. Release 13.3.

[31] NVIDIA Corporation. "CUTLASS 4.2.1 Overview and Performance Profiling." 2026. https://docs.nvidia.com/cutlass/4.2.1/overview.html#performance-profiling.

[32] NVIDIA Corporation. "Data Center GPU Manager Field Identifiers." NVIDIA DCGM Documentation, 2026. https://docs.nvidia.com/datacenter/dcgm/latest/dcgm-api/dcgm-api-field-ids.html.

[33] NVIDIA Corporation. "Data Center GPU Manager Profiling Metrics." NVIDIA DCGM Documentation, 2026. https://docs.nvidia.com/datacenter/dcgm/latest/learn/modules/profiling.html.

[34] NVIDIA Corporation. "Efficient GEMM in CUDA." 2026. https://docs.nvidia.com/cutlass/4.2.1/media/docs/cpp/efficient_gemm.html.

[35] NVIDIA Corporation. "Matrix Multiplication Background User's Guide." 2026. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html.

[36] NVIDIA Corporation. "NCCL Collective Operations." 2026. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html. Version 2.31.2.

[37] NVIDIA Corporation. "NVIDIA Management Library API Reference Guide." GPU Deployment and Management Documentation, 2026. https://docs.nvidia.com/deploy/nvml-api/.

[38] NVIDIA Corporation. "NVIDIA Nsight Compute Kernel Profiling Guide." NVIDIA Developer Tools Documentation, 2026. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html.

[39] NVIDIA Corporation. "NVIDIA System Management Interface Documentation." GPU Deployment and Management Documentation, 2026. https://docs.nvidia.com/deploy/nvidia-smi/index.html.

[40] NVIDIA Corporation. "Transformer Engine Fused Layers Documentation." 2026. https://docs.nvidia.com/deeplearning/transformer-engine-releases/release-2.16/user-guide/features/low_precision_training/performance_considerations/performance_considerations.html.

[41] "Parameters - CodeCarbon." https://docs.codecarbon.io/latest/reference/api/.

[42] PyTorch Contributors. "LayerNorm Documentation." 2026. https://docs.pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html.

[43] Radford, Alec, et al. "Language Models are Unsupervised Multitask Learners." OpenAI technical report, 2019. https://llmrisks.github.io/docs/language-models.pdf.

[44] Shoeybi, Mohammad, et al. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism." 2020-03-13. https://doi.org/10.48550/arXiv.1909.08053.

[45] Sun, Wei, et al. "Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors." *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 1, 2023-01, pp. 246–261. https://doi.org/10.1109/TPDS.2022.3217824.

[46] Vaswani, Ashish, et al. "Attention Is All You Need." *Advances in Neural Information Processing Systems 30*, Curran Associates, Inc., 2017. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[47] Vellaisamy, Prabhu, et al. "Characterizing and Optimizing LLM Inference Workloads on CPU-GPU Coupled Architectures." *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2025-05, pp. 49–61. https://doi.org/10.1109/ISPASS64960.2025.00015.

[48] Williams, Samuel, et al. “Roofline: An Insightful Visual Performance Model for Multicore Architectures.” *Communications of the ACM*, vol. 52, no. 4, 2009-04, pp. 65–76. https://doi.org/10.1145/1498765.1498785.

[49] Yang, Zeyu, et al. “Accurate and Convenient Energy Measurements for GPUs: A Detailed Study of NVIDIA GPU’s Built-In Power Sensor.” *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2024-11, pp. 1–17. https://doi.org/10.1109/SC41406.2024.00028.