Together Compiler: Continual Compilation for AI Inference

Reyna Abhyankar* Dylan Lim*
Drew Wadsworth Venkata Hasith Vattikuti YuTong Julia Zhang Jung Hwan Heo
Taeksang Kim Kesavan Ramakrishnan Evangelene Stanley Xinyu Fang Daniel Y. Fu
Simran Arora

Together AI Research

November 4, 2026

Today's AI workloads are rapidly changing: new models are released more often, model architectures are increasingly diverse, and traffic distributions change frequently. The challenge we face at Together AI is continual optimization for these changing workloads. Compiling a given deployment into an optimized forward pass is a challenge that is well-suited for agent hill-climbing. Existing systems struggle with continual compilation due to a tradeoff between measurement throughput and fidelity. Many systems use agents to optimize individual kernels from the inference forward pass using offline benchmarks, but timings are low-fidelity because of data-dependent behavior in primitives like mixture-of-experts (MoE). On the other hand, systems use agents to directly optimize end-to-end engines but suffer from slow engine launch times. To address these challenges, we build Together Compiler (TKCC), the first *truly continual* agent loop designed for AI inference optimization. To address the throughput-fidelity tradeoff, we propose: (1) a Tiered Persistent Engine that decouples engine launch overheads from forward pass computation; (2) a cost model to select the parallelism configurations that seed the optimization process. We also conduct extensive ablations on agent context management and present our insights on deploying TKCC in production. TKCC outperforms highly-optimized hand-written Megakernels on the Llama 1B workhorse workload by up to 1.7× and the best open source engines on the latest frontier models (Kimi K3, GLM 5.3, MiniMax M3.1 Flash, DeepSeek v4.1 Flash) across hardware platforms by up to 2.6× higher tokens per second (TPS) on SemiAnalysis' InferenceX benchmark. TKCC also sets the state-of-the-art performance on low-latency, batch size 1 inference, generating 1111 TPS on MiniMax M3.1 Flash and 1205 TPS on DeepSeek v4.1 Flash. Finally, TKCC generalizes and provides speedup beyond frontier language model inference. TKCC serves trillions of tokens across our production fleet.

1 Introduction

When AI agents have been presented with well-specified targets in mathematics, engineering, and science [37, 38], they have been effective at hill-climbing to achieve their objective. The key challenge for inference optimization at Together AI is that our traffic shape, i.e. our target to optimize, is constantly evolving. We explore the problem of *continually* compiling and optimizing our inference stack using AI agents.

Traffic primarily evolves due to three reasons. The first reason is behavioral trends. For instance, Lab A's model is more heavily utilized for agent-shaped workloads in one country compared to the rest of the world, which uses it for chat-shaped workloads. The second reason is burstiness, where a customer suddenly increases its usage or changes its use case. The third reason is shifts in usage patterns as users learn more about a model's capabilities. For Lab B's model, traffic gradually moved towards longer context, from 10K to 18K, and cache hit jumped by 8 percentage points in the 2-week period following the model's release.

*Equal contribution.
Correspondence to {rabhyankar, dylanlim, simran}@together.ai.

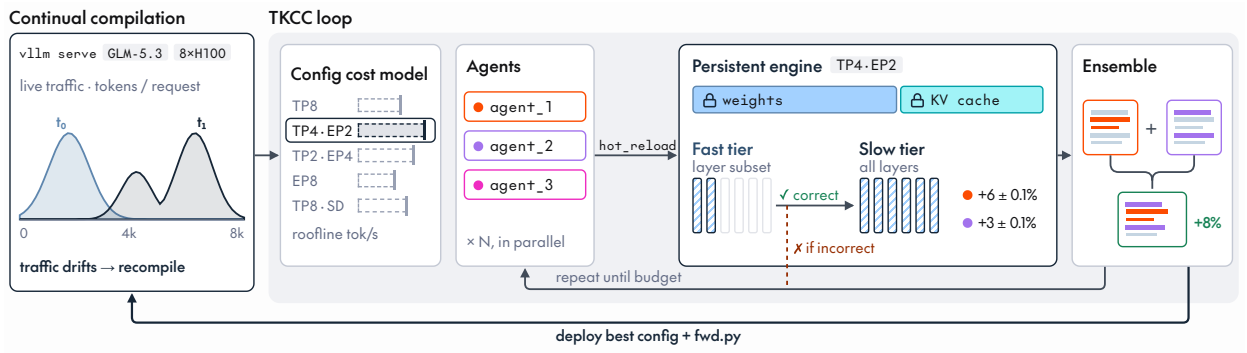


Figure 1: **Together Compiler architecture.** As input, TKCC takes in a server launch command, traffic profile, and hardware configuration. It uses a cost model to select the starting parallelism configuration for optimization, spawns parallel agents that are constantly communicating, and measures generated optimizations with a persistent engine. The final source code is directly integrated into the serving engine. As the traffic profile shifts over time, TKCC continually adapts.

There are many existing recent agentic optimization systems, but they either tackle narrow tasks (e.g., hyperparameter search [7] or one-off kernel benchmarking [17, 27]) or their design decisions have not been thoroughly examined across production traffic patterns [19, 28, 53]. When stress-testing existing systems on production traffic, we find that they are unable to achieve strong performance because they face a tradeoff between insufficient optimization throughput and insufficient measurement fidelity:

- **Low fidelity, high throughput.** Many agent systems optimize individual AI kernels on standalone benchmarks using random input data [26, 39]. However, this hurts measurement fidelity since the execution behavior of modern AI architectures is sensitive to the input data distribution (e.g., mixture-of-experts, sparse top-k attention) [13, 35, 59]. In Section 3, we show that measurement fidelity can impact the ideal optimization strategy for mixture-of-experts (MoE) kernels.
- **High fidelity, low throughput.** Other recent systems apply agents to end-to-end inference engine optimization, where the engine loads real model weights and measures speedups on real input data [28, 53]. This uses a high fidelity data distribution but suffers from painfully slow experiment iteration times due to engine launch overheads that take minutes (Section 4.2). All this idle GPU time is especially wasteful given the rising cost of AI compute [18].

Ideally, we could enable high-throughput iteration while preserving measurement fidelity. We introduce Together Compiler (TKCC), the first *truly continual* agent-driven loop for AI inference optimization that produces fast forward passes and integrates them directly into the serving engine (Figure 1). TKCC uses agents to search for optimizations and balances the throughput-fidelity tradeoff to unlock speedups. We highlight three key contributions from deploying TKCC in production:

1. **Tiered Persistent Engine.** Agents need high-fidelity measurements, but obtaining them requires expensive inference engine start up. To address this tradeoff between measurement throughput and fidelity, we introduce a new paradigm for iterating on inference optimization: the Tiered Persistent Engine. Our finding is that engine startup can be decoupled from forward pass computation, and the forward pass can be *hot-reloaded* with minimal overhead. TKCC initializes the production engine once and establishes two measurement tiers: a fast tier for high-throughput iteration and a slow tier for high-fidelity signal. The Tiered Persistent Engine allows TKCC to process 252× more agent experiments than existing inference optimization systems (Section 4.2).
2. **Cost model for parallelism selection.** As inference architectures and traffic evolve, selecting the optimal parallelism configuration requires balancing prediction fidelity against search cost. Two sources of uncertainty complicate this process: data-dependent execution (e.g., sparse KV accesses, MoE expert imbalance, and variable request lengths) and the maturity of existing software implementations, which

can obscure hardware performance potential. We propose a cost model that ranks the top-k parallelism configurations using analytical estimates and tiered measurements, ranging from theoretical hardware capabilities to measured compute, memory, and communication performance. This enables TKCC to identify promising configurations before investing in expensive agentic optimization (Section 4.3).

3. **Bitter lessons [51] of performance optimization.** We have been studying frameworks for accelerating AI inference for the past 2 years, starting from creating ThunderKittens [49] and Megakernels [47, 48] to building an in-house compiler. Much of our investigation predates tools like Claude Code, `/goal` or `auto mode`, or high-quality compaction. Earlier versions of TKCC deployed in production utilized rule-based lowering, built custom intermediate representations, and explicitly managed agent context. We continued to ablate these decisions as coding harnesses improved and have observed that agents without abstractions can achieve a higher performance ceiling (Section 4.4).

We evaluate Together Compiler on a variety of workloads against state-of-the-art baselines and make three primary claims, detailed in Section 5. First, Together Compiler outperforms highly-optimized megakernel baselines (e.g., ThunderKittens [49] and Mirage Persistent Kernel [8]) on their workhorse workload: up to 1.7× higher tokens per second (TPS) on Llama 1B (2,100 TPS). Second, Together Compiler adapts to different models and engine configurations, outperforming the best open performance on the SemiAnalysis InferenceX benchmark [44] across frontier models (Moonshot Kimi K3 [35], Z.ai GLM 5.3 [59], MiniMax M3.1 Flash [34], DeepSeek v4.1 Flash [13]) for diverse hardware platforms by up to 2.6× higher TPS. This was achieved in under 24 hours using at most 2 nodes for each model. We also set the state-of-the-art performance on low-latency batch size 1 inference, achieving 1,111 TPS on B200 M3.1 Flash and 1,205 TPS on GB300 DeepSeek v4.1 Flash. Third, TKCC generalizes to speedup in domains beyond language modeling. Together Compiler serves trillions of tokens per day and continually compiles our production fleet.

2 Background

We provide background on the following work: emerging model architectures, inference engine deployments, agentic kernel generation, and agent-driven optimization loops.

Emerging model architectures. Recent frontier open models – Z.ai's GLM 5.3 [19, 59], DeepSeek v4.1 Flash [13], MiniMax M3.1 Flash [34] – have started using two architectural patterns: MoE and sparse attention. MoE models dynamically route each query token to a subset of the model weights [16, 45], as opposed to dense transformers that apply the full model weights to every token [52]. MoE models thus maintains a large parameter space during training with higher inference efficiency [10, 11, 16]. Sparse attention dynamically selects and attends to a subset of the sequence [9, 12, 30, 58], as opposed to non-sparse attention that attends to all tokens [52]. Sparse attention improves inference efficiency at higher context lengths [12, 58].

Inference deployments. An inference engine schedules requests to be executed by the model's forward pass code. Requests can differ in their input sequence length (ISL) and output sequence length (OSL). Inference engines can choose to batch requests together [22] for higher throughput. Frontier models are served across several GPUs with different modes of parallelism [46], which result in a different layout for weights and KV cache data. TensorRT-LLM [1], SGLang [61], and vLLM [22] are examples of engines that differ in batching, cache layout, and model implementations.

Agentic kernel generation. Coding agents are rapidly improving at kernel generation [39, 57] and now beat expert baselines [31]. Agents can author kernels in raw CUDA/PTX, a domain-specific language (DSL), or even a custom intermediate representation (IR) designed specifically for agents, such as TIRx and CAKE [4, 56]. Systems built around agents (GEAK, Astra, STARK, KernelAgent, AutoComp, AccelOpt) [15, 21, 33, 53, 54, 60] typically employ the following strategy for choosing which kernels to optimize: they rank kernels with a cost model based on roofline or wall-clock estimates and then iteratively measure and refine one kernel at a time. We find that explicit strategies can negatively bias agent search (Section 4.4), while individual kernel optimization may not translate to end-to-end speedup (Section 3).

Agent-driven loops for inference optimization. Optimization loops involve agents proposing solutions, collecting measurements, and integrating feedback [37, 38]. Recent systems have built such loops for inference optimization [14, 59]. Apple's RSI-Inference Bench [28] evaluates different coding agents on a variety of inference optimization tasks in vLLM and SGLang. We find that these systems face low agent experiment throughput and GPU utilization because they launch the entire inference engine for each measurement of a generated optimization. We explore this in the following section (Section 3).

3 Motivation for Continual Compilation

Together AI serves many trillions of tokens per day across thousands of customers, so our services are constantly experiencing variable traffic shapes. The following are the primary dimensions of variation:

- **Behavioral trends.** Traffic shape is impacted by repeated patterns. Some countries use Lab A's model for agentic traffic, while other countries use the same model for chat. This results in the ISL shifting from $\approx$ 97k during one country's working hours to an ISL of $\approx$ 70k during off-hours.
- **Burstiness.** Customers frequently shift their requests per second or use case, impacting cache hit rate, ISL, and OSL. For example, within a period of 3 days, Customer A's P90 ISL grew from 18k to 33k tokens, OSL grew from 1k to 1.9k, and cache hit rate fell 4 percentage points. For Customer B, the average OSL increased from 700 to 1.7k, and cache hit rate fell by 7 percentage points within the span of one week.
- **Gradual shift.** Model labs roll out new checkpoints to our team over the course of their training runs, which can induce changes in the traffic shape. For example, Lab A rolled out a new checkpoint that generated more tokens during its thinking phase, resulting in 2$\times$ longer outputs. For Lab B's model, traffic gradually moved towards longer context, from 10K to 18K, and cache hit rate jumped by 8% over a 2-week period as customers shifted from evaluating the model to adopting the model.

Such changes in traffic requires continual optimization to maximize the GPU utilization of our fleet.

Throughput and fidelity tradeoffs. Recently, coding agents have provided significant value in accelerating performance optimization. Maximizing speedup requires high experimentation throughput and high-fidelity measurements:

- **Low fidelity, high throughput.** Existing agent systems from companies like Alibaba [17] and Cursor [27] iterate on standalone kernels in offline benchmark harnesses outside the inference engine (e.g., KernelBench [39]), often using random data as input. It is fast for agents to quickly try many optimizations using these harnesses, but random inputs provide low-fidelity measurements.
  - Randomly initialized data activates 13-38% more experts than real traffic in Figure 13a. On DeepSeek v4.1 Flash, we observe that Cursor's production-grade MoE megakernel system (Mixture-of-Kittens) [50] incurs a 6-15% slowdown when measured on real traffic compared to balanced load across experts [50] (Figure 13).
  - We also observe a similar problem with the indexer operation in sparse attention. An agent-authored top-k kernel was written assuming random data for its inputs, but this approach scaled quadratically on real data, which caused a 1.3$\times$ slowdown end-to-end in the engine (Figure 13b).
- **Higher fidelity, low throughput.** Other AI systems from companies like AMD [53], Apple [28], and Decagon [14], focus on end-to-end inference optimization. However, we find that they are unable to keep up with the velocity of traffic shifts. This is because their agents' experiments are bottlenecked by over ten minutes of inference engine startup time (Table 1).

Next, we introduce Together Compiler, the first *truly continual* agentic loop for AI inference optimization, powered by high-throughput and high-fidelity design decisions to maximize speedup.

Phase	Time (s)	Share (%)
Model loading	265	34.8
KV-cache sizing	73	9.6
Autotuning	192	25.2
CUDA-graph capture, decode	34	4.5
CUDA-graph capture, prefill	192	25.2
Total time to serving	**761**	**100.0**

(a) Engine cold-startup time

Metric	Slow tier	Fast tier
Layers served	60	12
Startup to first job (min)	11	3
Hot reload (s)	45	17
Job completion time, median (s)	64	28
Measurement variance (%)	0.14	0.05
Crashes / 1,000 jobs	0	3.9

(b) Tiered Persistent Engine statistics

Table 1: MiniMax-M3.1-Flash on TensorRT-LLM. We break down the startup time of an inference engine and showcase differences between tiers in the persistent engine.

4 Method

In this section, we describe Together Compiler's design and our lessons from developing the system. First, we propose a novel Tiered Persistent Engine architecture that provides high-fidelity and high-throughput measurement feedback to agents. Second, we propose a cost model that ranks the top-k parallelism configurations and selects the starting point for optimization. Third, we share insights from the previous two years of studying frameworks for accelerating AI inference and find that the bitter lesson [51] holds true for performance optimization.

4.1 System overview

Figure 1 shows the system architecture. TKCC takes as input the traffic shape, model, and hardware. TKCC's goal is to use agents to generate a series of optimizations for the inference engine and validate them by collecting measurements. TKCC spins up a persistent engine that loads the model once and accepts any agent-generated forward pass as a "candidate." TKCC uses a custom cost model to select the parallelism strategy for the persistent engine. The persistent engine *hot-reloads* a candidate and runs it through different measurement tiers that are tailored towards a traffic profile. To maximize GPU utilization, TKCC uses multiple parallel agents that are constantly communicating with each other to generate many diverse candidates. TKCC perpetually loops until a given budget is exhausted, at which point we run a full suite of verification tests (e.g., for KV cache contamination and quality), ensuring that generated candidate optimizations can be served directly in production.

4.2 High-throughput Tiered Persistent Engine

To apply TKCC at production scale, minimizing GPU idle time is critical. We propose to achieve high GPU utilization during optimization by constructing a persistent engine with distinct measurement tiers. This design leads to high agent experiment throughput and measurement fidelity.

Persistent engine with hot reloading. We observe that engine launch and teardown can be decoupled from model forward pass execution. Based on this observation, we build a persistent engine that can run different forward pass variants without restarting the engine. This is a new paradigm for agentic inference optimization; TKCC enables 252× higher throughput for agents and 11× lower idling time per agent compared to a system that restarts the engine for each measurement.

Figure 3 shows our implementation of "hot reloading." The engine is initialized once with an additional endpoint that accepts a candidate forward pass. When an agent submits its candidate as a "job" to the endpoint, the server replaces the model's forward function pointer and re-captures CUDA/HIP graphs. To ensure GPUs are fully utilized, TKCC spawns parallel agents so that one agent can think and make tool calls while other agents are receiving measurement feedback.

Two further optimizations to improve throughput are (1) Prefix cache reuse: rather than using fresh requests for each measurement, TKCC reuses a fixed set of requests. Since the prompts remain cached across

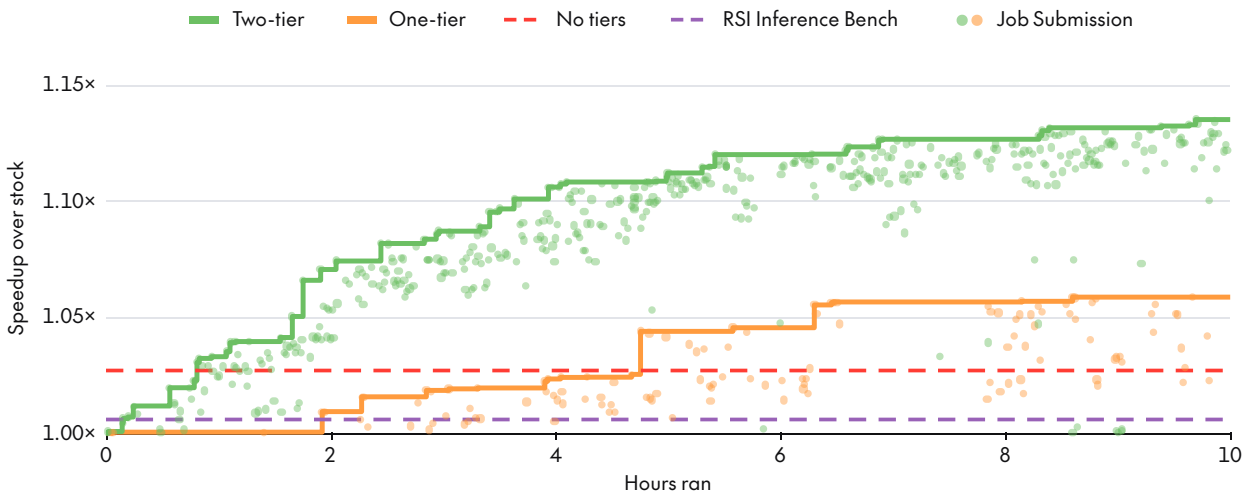


Figure 2: TKCC's two-tiered measurement system outperforms a single slow tier, no measurement tiers, and Apple's RSI inference bench harness on B200 GLM 5.3 optimization.

measurements, measurement latency reduces by an additional 2.6× on Kimi K3. (2) Binary reuse: many kernels remain the same from one submission to the next, so we cache kernel binaries to prevent needless recompilation.

Given the high-throughput measurement framework, agents must use the experimentation capacity effectively. We find that maximizing agent communication is critical for avoiding redundant tasks, improving optimization speed by 2.3× (Figure 5).

Tiered measurement. TKCC instantiates multiple persistent engines as distinct measurement tiers: a high-throughput tier and a high-fidelity tier. Agents submit to the faster tier to weed out slower kernels that could cause illegal memory accesses (IMAs) or hangs to avoid expensive engine restarts (Table 1a). Slower tiers contain more layers (Table 1b) and provide higher-fidelity measurements for data-dependent architectures. Agents see each tier's queue depth and job completion times and can load-balance among themselves. The fast tier also traces kernels (e.g., with CUPTI on NVIDIA) to provide agents with kernel-level reports. In Figure 2, we show the two-tiered measurement system evaluates 8.1× more candidates than a single slow tier (1,354 vs. 168) and finds a solution that is 7.2% faster.

We find that running the same forward pass back-to-back in the engine can vary measurements by 4-6%, which is problematic because a valid optimization can boost performance by even as little as 1%. To reduce variance, we (1) Split prefill and decode: with continuous batching, decode timings vary based on whether it is mixed with prefill requests. We decouple prefill from decode measurements, reducing variance in back-to-back runs from 6.6% to 2.7%. (2) Tune measured layers: We find that the number of measured layers affects measurement variance. Before optimization starts, we tune the minimal number of layers needed for representative measurements for that particular workload.

4.3 Cost model for configuration search

We find that changes in traffic shape may also need changes to the inference engine's parallelism configuration. Therefore, we introduce a cost model to identify the top-k promising configurations before investing in expensive agent optimization.

Existing systems have developed sophisticated methods for selecting inference configurations under SLA requirements [41, 55, 62, 63]. For continual configuration selection, we face a tradeoff between prediction fidelity and search cost: measuring every configuration is expensive and reflects the immaturity of existing software implementations, while simple analytical models can miss data-dependent execution effects such as MoE imbalance, multi-token prediction (MTP) acceptance lengths, and uneven request lengths in a batch.

```python
# Agent writes optimized forward pass in "tkcc_forward.py"
def forward(model, input_ids, positions):
    return optimized_forward(model, input_ids, positions)

# Agent submits job to server
post_rpc(engine, "hot_reload", file="tkcc_forward.py")
result = measure(engine)
post_rpc(engine, "teardown")

# Engine updates the forward function pointer
def hot_reload(file):
    write("/tmp/forward.py", file)
    candidate = exec_file("/tmp/forward.py")
    baseline = model.forward
    model.forward = candidate.forward       # patch
    recapture_cuda_graphs()
def teardown():
    restore(model, baseline)   # unload modules
    recapture_cuda_graphs()
```

Figure 3: Hot reloading function snippet. (1) The agent writes `tkcc_forward.py`. (2) The agent sends it to the server over RPC. (3) Each rank writes the file to disk, runs `exec`, updates the model's forward pointer, recaptures CUDA graphs, and unloads kernel modules upon teardown.

We address these challenges with a cost model that combines analytical estimates with tiered hardware performance measurements.

Cost modeling. We construct the search space for possible configurations, prune the search space, and collect high-fidelity measurements. Then, we use our model to provide a throughput estimate for the configuration at a fixed interactivity SLA.

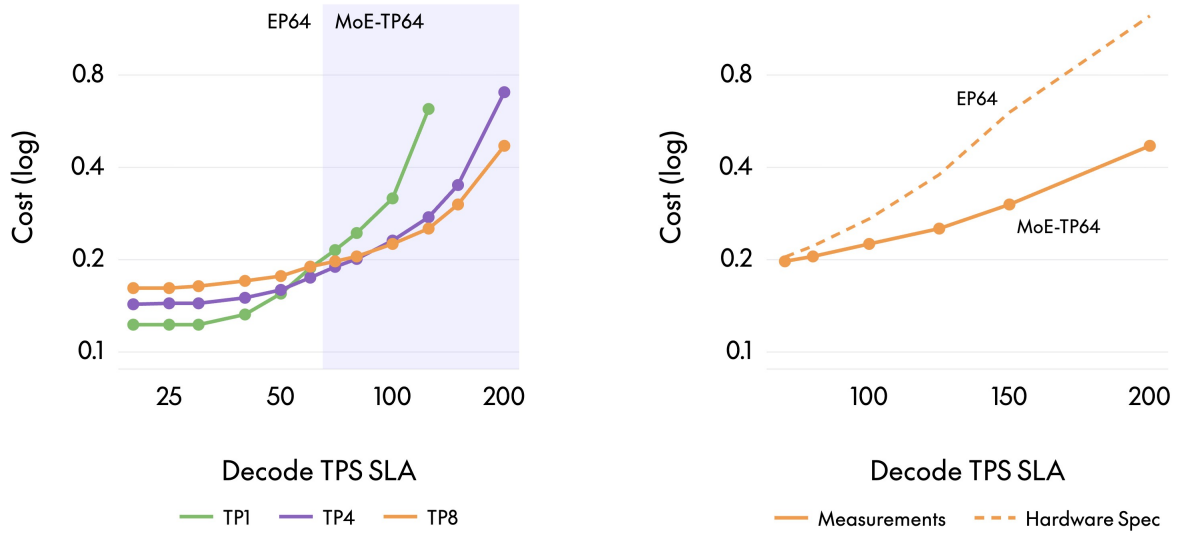
1. **Search space.** The search space considers replica sizes from one to a fixed maximum number of chips (e.g., 4096). We collect all combinations across: **MoE**: Expert Parallel (EP), Tensor Parallel (TP); **Dense**: Tensor Parallel; **Attention**: Tensor Parallel, Data Parallel (DP), Context Parallel (CP).

2. **Pruning.** We prune configurations based on a few factors. First, we eliminate configurations that cannot fit the model weights or evenly divide any of the model's tensor dimensions (e.g., the number of attention heads should be divisible by the number of chips). Second, we eliminate configurations if they cannot meet a fixed prefill time-to-first-token (TTFT) and decode TPS SLAs (1.5s TTFT, 40 TPS). For example, layouts that are spread over hundreds of nodes are often eliminated due to high communication overhead. The remaining rules are described in Appendix D.

3. **Tiered measurements.** We account for two sources of uncertainty when estimating configuration performance: the gap between theoretical hardware capabilities and current software performance, and variability in production traffic.

   (a) First, configurations should not be penalized solely because their current implementations are poorly optimized. We therefore distinguish between three levels of performance estimates: theoretical hardware specifications, measured compute/memory/collective performance, and measured end-to-end forward-pass performance. Comparing these estimates helps distinguish hardware limitations from software implementation overheads (Figure 4b).

   (b) Second, several components of modern inference are data-distribution dependent (e.g., MTP acceptance rates [29] and MoE expert activation imbalance [36]) and workload dependent (e.g., prompt unevenness across requests in a batch). We measure these distributions in production, since uneven expert routing and request lengths can create stragglers that determine the execution time of an entire parallel replica.



(a) EP wins at higher concurrencies while higher TP wins at lower concurrencies.

(b) Hardware specs favor EP since it ignores per-call latency for collectives.

Figure 4: **Cost Model Crossover Points.** As decode TPS increases, the best configuration uses TP instead of EP for MoE layers and higher degrees of TP for the attention and dense weights.

4. **Model.** Given a model, a chip, and a set of parallelism configuration candidates, the cost model estimates the forward-pass time (T_{pass}) of the prefill and decode phases per unique model layer. Each layer's time is the larger of its compute time (T_c^L, from FLOPs and peak throughput) and its memory time (D^L/BW, from bytes read and memory bandwidth), plus its collective time (T_{comm}^L, from bytes moved and interconnect bandwidth and latency). The model then finds the largest batch (C) that fits in high-bandwidth memory (HBM) alongside weights and KV cache (C_{memory}) while still meeting the phase's latency target (C_{speed}), giving chip-seconds per request for that phase (chip-s_{prefill}, chip-s_{decode}). Prefill and decode chip-seconds are summed and inverted into a single metric: tokens per minute per chip (TPM).

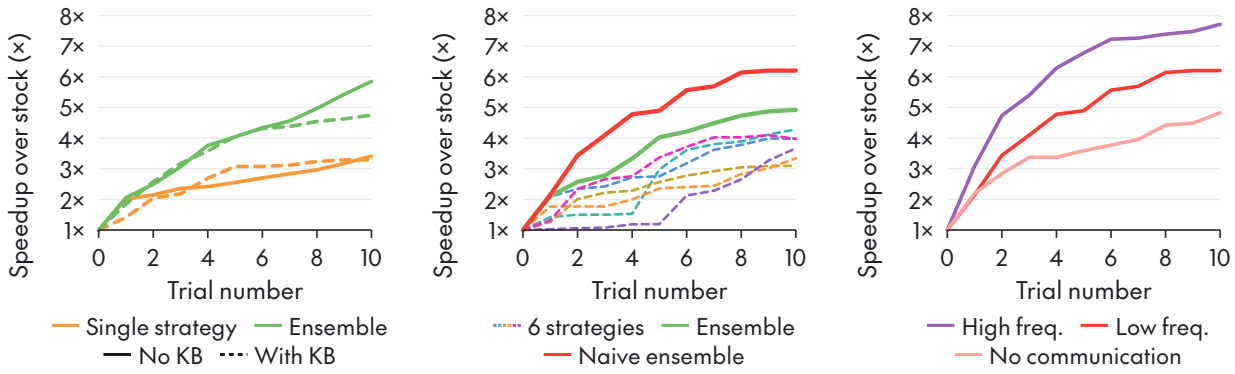
$$T_{\text{pass}} = \sum_L \left[\max\left(T_c^L,\ \frac{D^L}{BW} \right) + T_{\text{comm}}^L \right]$$

$$\tau_{\text{pass}} = \frac{N\, T_{\text{pass}}\, n_{\text{pass}}}{C}$$

$$\text{TPM} = \frac{60\,(\text{ISL} + \text{OSL})}{\tau_P + \tau_D}$$

Using the cost model. We illustrate how the cost model selects different configuration as model architectures, latency requirements, and usage patterns evolve. *Architecture:* For models with fewer experts, such as GLM 5.3, smaller TP configurations are favored because additional parallelism introduces communication overhead without sufficient computational benefit. As expert counts increase, as in Kimi K3, wide EP configurations spanning 32 or 64 GPUs become more efficient. *Latency requirements:* For Kimi K3, wide EP is favored in throughput-oriented regimes, where higher concurrency amortizes communication costs. As the per-user TPS requirement increases, the maximum feasible concurrency decreases, favoring higher TP degrees for both attention and MoE (Figure 4a). *Usage patterns:* As the MTP acceptance length increases, each decoding step generates more tokens, relaxing the latency constraint for a fixed TPS target. This allows for more replicas to support higher concurrencies, shifting the optimal configuration toward 64-GPU wide EP rather than 32-GPU configurations.

The cost model was able to improve our production deployment for Kimi K3 by finding a configuration



(a) Inclusion of a knowledge base can incur 26% worse performance.

(b) Expert-tuned ensembling is 1.3× slower than naive strategies.

(c) High-frequency agent communication is 4.7× faster.

Figure 5: Ablating techniques for agent strategies during search on MiniMax-M3 layer 3, B200 TP4, batch size 1 with TensorRT-LLM.

that boosted TPS by up to 24% and TPM by up to 18%. In Figure 8, we use this cost model to select parallelism configurations that lead to speedups over InferenceX parallelism configs to improve per-user interactivity metrics.

4.4 Lessons from building the Together Compiler

Our primary takeaway is that broad AI exploration can now increasingly outperform prescriptive intermediate representations, curated optimization strategies, and explicit agent context management.

System versions. We have studied frameworks for AI inference acceleration for the past two years, putting three distinct iterations of the design into production for customer use:

1. **Megakernels [47, 48].** We fuse all model operations into a single kernel to minimize launch overheads and tail effects. Specialized warps perform instruction scheduling on the device side.

2. **Torch Compile [3] augmentation.** Given the difficulty of authoring Megakernels at scale, we turned to augmenting Torch Compile (aka Inductor). Inductor traces PyTorch code to produce an intermediate representation called the FX Graph. From there, we lowered all captured kernels to PTX, inserted optimizations like Programmatic Dependent Launch (PDL), and replaced with optimized kernels. Then, we recompiled the resultant graph into a single binary. Since the FX Graph cannot represent kernels launched from C++, we leveraged low-level tracing (e.g., CUPTI for NVIDIA, rocProf for AMD [42]) to augment the graph with the missing kernel launch information.

3. **Expert-managed agent context.** We provided expert-curated knowledge bases and optimization strategies to seed each agent's optimization approach. We asked agents to log their research findings in a custom log data structure.

As coding harness capabilities have improved (Claude Code, `/goal`, `auto mode`, better compaction), we found ourselves *removing intermediate representations* to unlock higher performance ceilings. TKCC now directly provides the agent with the inference engine's source code and no explicit context management structures or curated strategies.

Traditional IRs have limited expressivity. We hypothesized that IRs could help agents understand complex inference codebases, which contain multiple codepaths and kernel launch mechanisms. When directly provided with the source code, we do observe that agents spend $\approx 20\%$ of their time reading engine files to understand the relevant code paths and kernel calls within the engine code. However, we also find that

unrestricted source code access unlocks a higher performance ceiling. Even our heavily endowed augmented Inductor IR was not expressive enough to permit certain optimizations. For example, using the source code representation to optimize Kimi K3, agents could rewrite the MLA metadata planning kernel, so long sequences could be split across multiple thread blocks during attention, which provided a $1.7 - 2.5\times$ speedup on the MLA kernel itself. Neither the default, nor the augmented FX graph would be able to mutate the host-side metadata.

Surprising findings on expert-curated context for agents. A prior version of our system, as well as most agentic performance optimization systems [53], utilize a knowledge hub, require agents to use research and experimentation logs, or prescribe the order in which the agent should tackle optimizations. For instance, systems often rank kernels by estimating its roofline and potential wall-clock speedup. Over time, in light of improvements in coding agent abilities, we ablated more of these choices and found surprising results:

- *Knowledge hub.* Originally, our production system used a knowledge hub seeded by experts. Upon ablating, we found that this approach can lead to 25–31% lower speedup than agent performance without a knowledge hub after 12 hours of optimization (Figure 5a).
- *Expert strategies.* Our system also seeded each parallel agent with a different expert-written strategy (e.g. fusion focused, communication-computation overlap, Megakernels). Upon ablating, we found that simply giving each agent a simple prompt for optimization ("make this forward pass as fast as possible") outperforms the expert prompts (Figure 5b).
- *Research and logging.* We find that providing human-engineered data structures for the agents to log and track their research also impedes throughput. Agents spend up to 43% of the CLI/tool calls per trial on research tracking and data structure inspection, time that could instead be spent on optimization.

Additional details on the expert-curated context are provided in Appendix C. Together, these experiments point towards a bitter lesson [51] in agentic performance optimization. However, while we find that prescriptive IRs can constrain agent optimization, structured representations can be valuable when verifying the correctness of compiled programs. Ideally, future work could design an IR for verifying agent-generated kernels, without restricting the optimization space.

5 Results

We evaluate Together Compiler on a variety of workloads against the strongest available open-source baselines. We find that TKCC:

1. **Outperforms hand-tuned Megakernels.** Together Compiler outperforms highly-optimized Megakernels (ThunderKittens [47] and Mirage Persistent Kernel [8]) on their workhorse Llama 1B workload by up to $1.7\times$ (TPS) on batch size 1. We evaluate on H100 and B200 GPUs on output sequence lengths up to 64K tokens.
2. **Tops engine leaderboards.** Together Compiler adapts to different models, engine configurations, and hardware platforms, outperforming the best open performance on SemiAnalysis' InferenceX benchmark [44] across frontier models (Moonshot Kimi K3 [35], Z.ai GLM 5.3 [59], DeepSeek v4.1 Flash [13], and MiniMax M3.1 Flash [34]) by up to $2.6\times$ (TPS).

   TKCC also sets the state-of-the-art performance at the low-latency batch size 1 regime, generating 1111 TPS on B200 MiniMax 3.1 Flash and 1205 TPS on GB300 DeepSeek v4.1 Flash.
3. **Generalizes across codebases and domains.** Third, Together Compiler generalizes beyond frontier inference, providing speedups of $1.5 - 6.4\times$, outperforming Claude Code on a variety of models spanning weather, biology, time series forecasting, and partial differential equations (PDE) simulation.

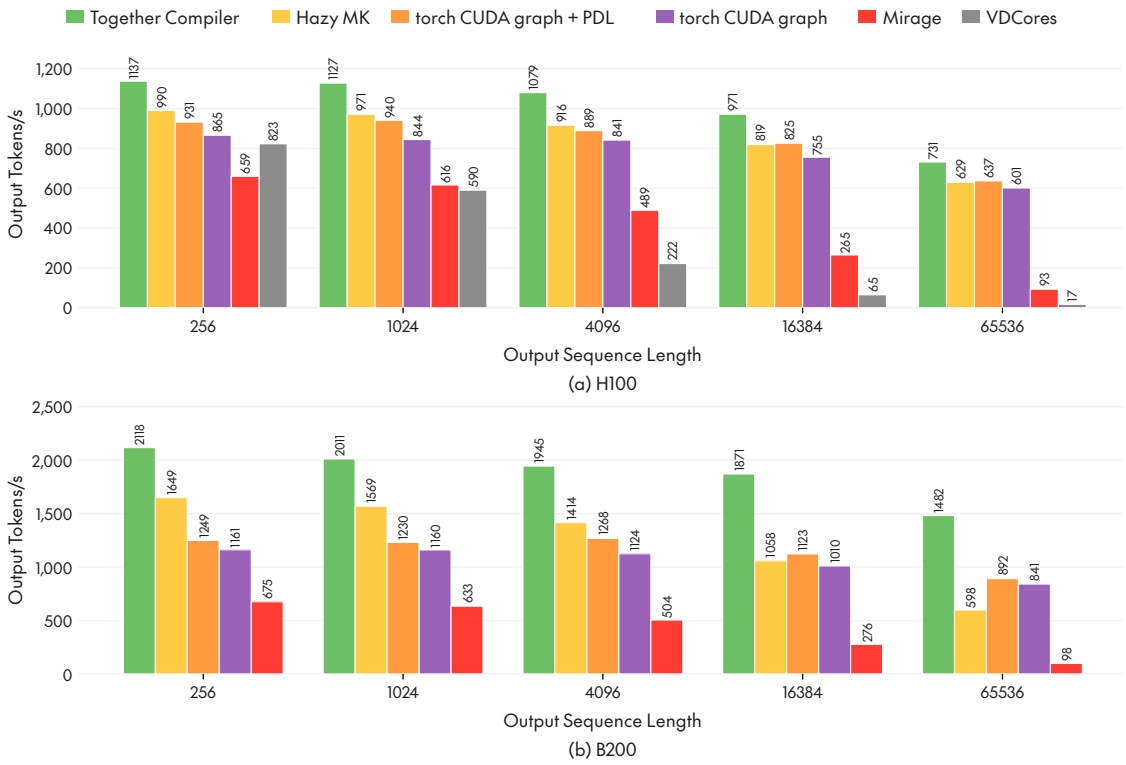


Figure 6: TKCC outperforms existing systems on Llama-3.2-1B-Instruct, Batch Size 1, ISL 32

5.1 Optimizing traditional workhorse workloads

Figure 6 shows that TKCC achieves state-of-the-art performance on batch size 1 Llama 1B inference running on both H100 and B200s and across a range of output sequence lengths. This was achieved without the typical megakernel abstractions. Upon investigation, we find gains from the fine-grained memory paging system to be small (worth only 3.1% of the full forward pass) and that driver-level primitives such as PDL, CUDA graphs, and L2 prefetching are able to recover the benefits of a single persistent kernel.

5.2 Topping inference leaderboards

We evaluate the improvements that TKCC yields over the top AgentX configuration per concurrency, model, and hardware (as of October 27, 2026).

Protocol. For each setting, we use the same parallelism strategy that was used by leading configuration to isolate the effect of the agentic optimization system. For optimization compute, TKCC uses 2× the replica size per model compilation; this supports our tiered persistent engine. We evaluate on B200, B300, GB300, MI355X, and Trainium2, demonstrating that the system generalizes across hardware platforms. We compile just once per parallelism configuration and use the same resulting forward pass for each concurrency that uses the same strategy. For AMD and NVIDIA, we use the AgentX traffic profile as the workload. We evaluate Trainium2 using a synthetic multi-turn dataset that uses 70k token ISL, shorter contexts than AgentX, to focus on accelerator performance instead of CPU offloading overheads (details in Appendix A).

Results. In Figure 7, we show that TKCC provides up to 2.6× improvement in TPS across settings in just a 24-hour compilation window per setting. This improvement stems from TKCC's high-throughput engine, which improves agent experiment throughput by 252× and reduces idling time per agent by 11×. Notably, TKCC sets the state-of-the-art across different engines (vLLM, SGLang) and hardware platforms (B200, B300, MI355X, Trainium2). In Figure 8, we identified points where our cost model suggests a different

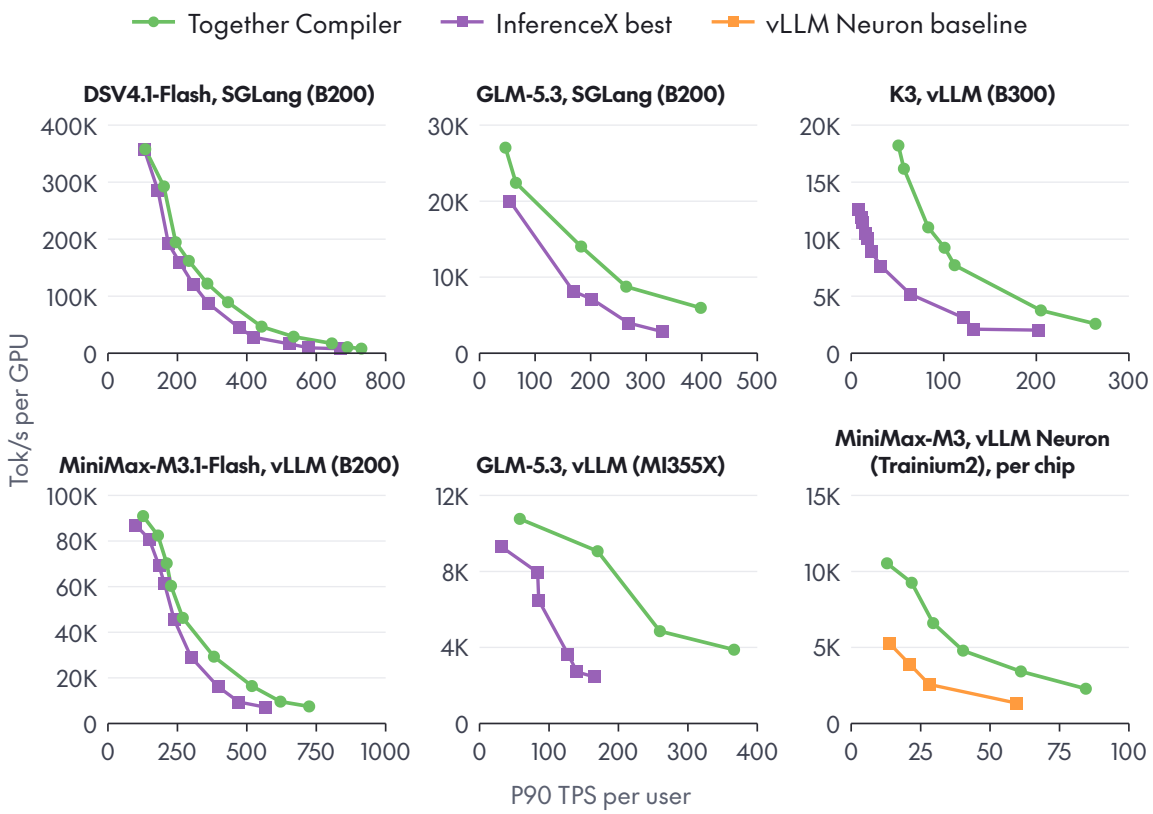


Figure 7: TKCC improves highly-optimized open-source baselines in under 24 hours of compilation, using at most 2 nodes per model. For these runs, we re-purpose the parallelism strategy proposed by the prior best InferenceX engine baseline as a starting point for our runs. The workloads and images used for these experiments are in Appendix A.

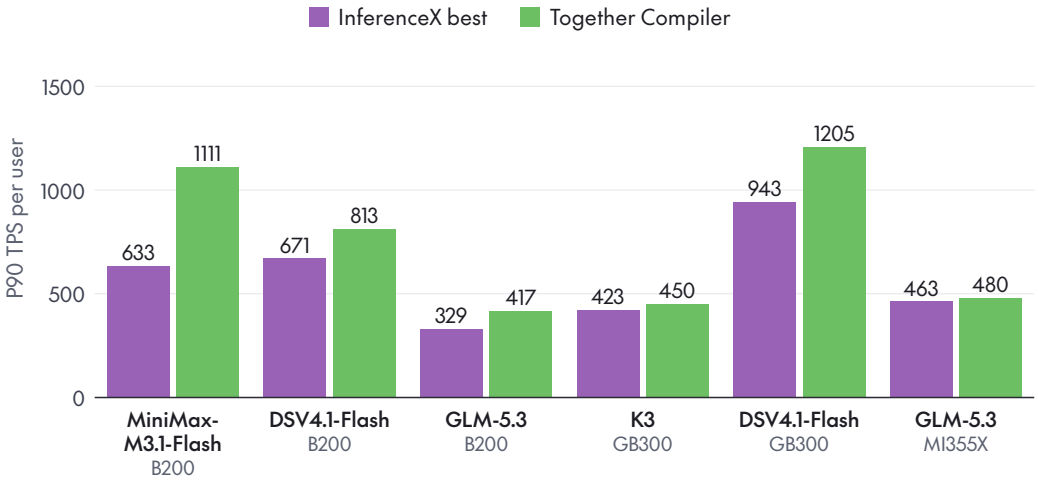


Figure 8: For these runs, we use our cost model to select the parallelism configurations that TKCC's optimization loop focuses on, enabling state-of-the-art TPS in the low concurrency regime in under 24 hours of optimization.

Model	Parameters	Evaluation	**TKCC**	**Claude Code**
AIFS-ENS 2.0	230M	36-case forecast eval	3.5×	Invalid
Aurora 1.5	1.3B	ERA5: 54 cases	3.7×	3.0×
Boltz-2 (Anthropic)	521M	7-size residue eval	1.5×	1.2×
Evo 2 (Anthropic)	7B	16,000-token generation	1.6×	1.6×
Chronos-2	120M	Chronos: 27 datasets	2.6×	2.2×
Panda-72M	72M	Chaotic ODEs	1.9×	1.9×
Walrus	1.3B	The Well: radiative layer	4.3×	3.2×
Poseidon-T	21M	NS-PwC	6.4×	4.4×

Table 2: TKCC and Claude Code (`/goal`) speedups over baseline implementations on the respective evaluation datasets for each model. On AIFS-ENS 2.0, Claude Code fails to find a correct solution.

parallelism strategy than the baseline image at concurrency 1. For MiniMax M3.1 Flash, the top-performing B200 image on InferenceX follows a TP4 configuration. However, at batch size 1, decode is memory-bound. TKCC suggests a TP8 configuration, which halves each GPU's weight traffic. With TP8, TKCC reaches 1,111 TPS, 1.8× higher than the best published result (633). This configuration may not be intuitive. Running the same InferenceX image under TP8 achieves similar performance to the chosen TP4 result.

Continual compilation as workloads shift. TKCC is able to identify different optimizations across concurrency regimes, adapting to bursty traffic behavior. On Kimi K3, optimizations such as a persistent MoE down-projection boosts performance by 2.8-5% across concurrencies 22-78 but is 6.6% slower at concurrency 8. TKCC recognizes this and selectively applies optimizations based on the active batch size.

While we optimize open-source baselines to help readers contextualize our results, we have also used TKCC to optimize for production traffic across our internal engine configurations. Notably, when replaying our production traffic, we found that not all InferenceX baselines scale. Measured on our internal benchmarks, the leading InferenceX MiniMax M3.1 Flash image hits a capacity ceiling at low requests per second per replica, and the tail TTFT degrades 10-40× while 17% of requests fail.

5.3 Generalizing across codebases and domains

We evaluate TKCC on domains and codebases beyond language model inference, specifically models for weather, biology, time series forecasting, and PDE simulation. We find that TKCC's design generalizes to these domains, producing faster models that translate to gains in simulation accuracy at test-time.

Workloads We optimize the AIFS-ENS 2.0 [24] and Aurora 1.5 [5] weather models; the Boltz-2 [40] and Evo 2 [6] biology models (on top of Anthropic's and Evo 2's baselines [43]); the Chronos-2 [2] and Panda-72M [23] time series foundation models; and Walrus [32] and Poseidon-T [20] PDE foundation models (2D only).

Results. TKCC speedups over the baseline forward passes range from 1.5× to 6.4× (Table 2), while maintaining bit exactness to the original forward pass. We compare TKCC to the baseline of using the Claude Code `/goal` command in Appendix B and Figure 11). TKCC achieves performance improvements at a faster rate than Claude Code. Speedups can unlock model accuracy gains for tasks like weather modeling that require taking many samples to reduce statistical noise or models that benefit from higher-resolution processing. For instance, AIFS and Aurora rely on ensembling multiple perturbed samples to produce forecasts for metrics like mean sea-level pressure and air temperature. The optimizations generated by TKCC allows users to take more samples and reduce statistical noise (Table 3).

Model	Run	Samples	Temperature RMSE ↓	CRPS ↓
AIFS	Baseline	4	0.97 K	0.50
	TKCC	**13**	**0.94 K**	**0.46**
Aurora	Baseline	4	0.60 K	0.25
	TKCC	**14**	**0.55 K**	**0.21**

Table 3: TKCC enables more sampling in a fixed wall-clock time than the baseline implementation, corresponding to reductions in Root Mean Squared Error (RMSE) and Continuous Ranked Probability Score (CRPS) for temperature measurements.

6 Conclusion

The era of static inference is over. With evolving model architectures and customer use cases, we observe that inference has taken on many forms—it has now become "polymorphic." The same optimizations no longer translate across the board, and each workload demands specialized deployments. With Together Compiler, we took agent-drive optimization from a prototype to a scalable production system for inference optimization. Our work shows that a successful agentic optimization loop can be built around three guiding principles: (1) Balance the measurement throughput-fidelity tradeoff; we accomplish this through a tiered persistent engine. (2) Establish the upper bound; we achieve this through a cost model that can reason through software immaturity. (3) Minimize human intervention; we deleted our old abstractions (custom IRs, knowledge bases, expert strategies) and left the agents in charge. The result is Together Compiler: an efficient system that boosts performance over highly-optimized baselines on the latest model releases by up to 2.6× in under 24 hours using just 2 nodes per model, while generalizing across hardware platforms and domains. For agent-drive optimization to be useful, it must be *continual*.

7 Acknowledgments

We thank Ce Zhang, Chris Ré, Neel Guha, Stuart Sul, Ali Yassine, Alex Angus, Austin Silveria, Gordon Gustafson, Kevin Cui, Michael Granado, Sonny Li, Xinyi Li, Peter Wu, Sanjana Srivastava, Federico Bianchi, Pablo Cases, Susan Lee, and Nikitha Suryadevara for helpful feedback and discussions during this work.

References

[1] Tensorrt-llm. URL https://github.com/NVIDIA/TensorRT-LLM.

[2] Abdul Fatir Ansari, Oleksandr Shchur, Jaris Küken, Andreas Auer, Boran Han, Pedro Mercado, Syama Sundar Rangapuram, Huibin Shen, Lorenzo Stella, Xiyuan Zhang, Mononito Goswami, Shubham Kapoor, Danielle C. Maddix, Pablo Guerron, Tony Hu, Junming Yin, Nick Erickson, Prateek Mutalik Desai, Hao Wang, Huzefa Rangwala, George Karypis, Yuyang Wang, and Michael Bohlke-Schneider. Chronos-2: From univariate to universal forecasting, 2025. URL https://arxiv.org/abs/2510.15821.

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024.

[4] Apache TVM Community. Tirx: An open compiler stack for evolving frontier ml kernels, June 2026. URL https://tvm.apache.org/2026/06/22/tirx.

[5] Cristian Bodnar, Wessel P. Bruinsma, Ana Lucic, Megan Stanley, Anna Allen, Johannes Brandstetter, Patrick Garvan, Maik Riechert, Jonathan A. Weyn, Haiyu Dong, Jayesh K. Gupta, Kit Thambiratnam, Alexander T. Archibald, Chun-Chieh Wu, Elizabeth Heider, Max Welling, Richard E. Turner, and Paris Perdikaris. A foundation model for the Earth system. *Nature*, 641(8065):1180–1187, May 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09005-y.

[6] Garyk Brixi, Matthew G. Durrant, Jerome Ku, Mohsen Naghipourfar, Michael Poli, Gwanggyu Sun, Greg Brockman, Daniel Chang, Alison Fanton, Gabriel A. Gonzalez, Samuel H. King, David B. Li, Aditi T. Merchant, Eric Nguyen, Chiara Ricci-Tam, David W. Romero, Jonathan C. Schmok, Ali Taghibakhshi, Anton Vorontsov, Brandon Yang, Myra Deng, Liv Gorton, Nam Nguyen, Nicholas K. Wang, Michael T. Pearce, Elana Simon, Etowah Adams, Zachary J. Amador, Euan A. Ashley, Stephen A. Baccus, Haoyu Dai, Steven Dillmann, Stefano Ermon, Daniel Guo, Michael H. Herschl, Rajesh Ilango, Ken Janik, Amy X. Lu, Reshma Mehta, Mohammad R. K. Mofrad, Madelena Y. Ng, Jaspreet Pannu, Christopher Ré, John St. John, Jeremy Sullivan, Joseph Tey, Ben Viggiano, Kevin Zhu, Greg Zynda, Daniel Balsam, Patrick Collison, Anthony B. Costa, Tina Hernandez-Boussard, Eric Ho, Ming-Yu Liu, Thomas McGrath, Kimberly Powell, Sudarshan Pinglay, Dave P. Burke, Hani Goodarzi, Patrick D. Hsu, and Brian L. Hie. Genome modelling and design across all domains of life with Evo 2. *Nature*, 652(8112): 1349–1361, April 2026. ISSN 1476-4687. doi: 10.1038/s41586-026-10176-5.

[7] Tony Chen, Leon Li, Jason Mancuso, Charles Frye, and Rahul Chalamala. Autoscaling autoresearch: Give your agents elastic GPUs on Modal. Modal Blog, April 2026. URL https://modal.com/blog/autoscaling-autoresearch. Accessed: 2026-10-30.

[8] Xinhao Cheng, Zhihao Zhang, Yu Zhou, Jianan Ji, Jinchen Jiang, Zepeng Zhao, Ziruo Xiao, Zihao Ye, Yingyi Huang, Ruihang Lai, Hongyi Jin, Bohan Hou, Mengdi Wu, Yixin Dong, Anthony Yip, Zihao Ye, Songting Wang, Wenqin Yang, Xupeng Miao, Tianqi Chen, and Zhihao Jia. Mpk: A compiler and runtime for mega-kernelizing tensor programs. Seattle, WA, July 2026. USENIX Association.

[9] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[10] Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. DeepSeekMoE: Towards ultimate expert specialization in mixture-of-experts language models, 2024. URL https://arxiv.org/abs/2401.06066.

[11] DeepSeek-AI. Deepseek-v3 technical report, 2024. URL https://arxiv.org/abs/2412.19437.

[12] DeepSeek-AI. Deepseek-v3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

[13] DeepSeek-AI. Deepseek-v4.1-flash: Pushing the limits of KV cache compression, 2026. URL https://arxiv.org/abs/2609.19969.

[14] Pavi Dhiman and Cyrus Asgari. Building an Inference Stack That Improves Itself. https://x.com/DecagonAI/status/2115511110120120437, 2026. Decagon, article posted on X, October 28, 2026. Accessed: 2026-10-30.

[15] Juncheng Dong, Yang Yang, Tao Liu, Yang Wang, Feng Qi, Vahid Tarokh, Kaushik Rangadurai, and Shuang Yang. Stark: Strategic team of agents for refining kernels, 2025. URL https://arxiv.org/abs/2510.16996.

[16] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022. URL https://arxiv.org/abs/2101.03961.

[17] Tianyu Feng, Haoxuan Yu, Tianyuan Wu, Lingyun Yang, Daocheng Ying, Yuxiao Wang, Ruibo Fan, Yinghao Yu, Guodong Yang, Liping Zhang, and Wei Wang. Krex: Concurrent kernel benchmarking on shared gpus via region-granular exclusivity, 2026. URL https://arxiv.org/abs/2609.30057.

[18] Frank Flight. Tokenomics 2.0. Citadel Securities, Global Macro Strategy, October 2026. URL https://www.citadelsecurities.com/news-and-insights/global-macro-strategy/tokenomics-2-0/. Accessed: 2026-10-30.

[19] GLM-5 Team. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

[20] Maximilian Herde, Bogdan Raonić, Tobias Rohner, Roger Käppeli, Roberto Molinaro, Emmanuel de Bézenac, and Siddhartha Mishra. Poseidon: Efficient foundation models for pdes, 2024.

[21] Charles Hong, Sahil Bhatia, Alvin Cheung, and Yakun Sophia Shao. Autocomp: A powerful and portable code optimizer for tensor accelerators, 2025. URL https://arxiv.org/abs/2505.18574.

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

[23] Jeffrey B. Lai, Anthony Bao, and William Gilpin. Panda: A pretrained forecast model for chaotic dynamics. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=DgnsohAUMn.

[24] Simon Lang, Martin Leutbecher, and Pedro Maciel. A multi-scale loss formulation for learning a probabilistic model with proper score optimisation, 2025. URL https://arxiv.org/abs/2506.10868.

[25] Brian Li, Faraz Shahsavan, and Pankaj Gupta. Agentic kernels in production. Baseten Blog, August 2026. URL https://www.baseten.co/blog/agentic-kernels-in-production/.

[26] Edward Lin, Sahil Modi, Siva Kumar Sastry Hari, Qijing Huang, Zhifan Ye, Nestor Qin, Fengzhe Zhou, Yuan Zhang, Jingquan Wang, Sana Damani, Dheeraj Peri, Ouye Xie, Aditya Kane, Moshe Maor, Michael Behar, Triston Cao, Rishabh Mehta, Vartika Singh, Vikram Sharma Mailthody, Terry Chen, Zihao Ye, Hanfeng Chen, Tianqi Chen, Vinod Grover, Wei Chen, Wei Liu, Eric Chung, Luis Ceze, Roger Bringmann, Cyril Zeller, Michael Lightstone, Christos Kozyrakis, and Humphrey Shi. Sol-execbench: Speed-of-light benchmarking for real-world gpu kernels against hardware limits, 2026. URL https://arxiv.org/abs/2603.19173.

[27] Wilson Lin, Sahil Modi, Yuan Zhang, and Edward Lin. Speeding up GPU kernels by 38% with a multi-agent system. Cursor Blog, April 2026. URL https://cursor.com/blog/multi-agent-kernels. Accessed: 2026-10-30.

[28] Junteng Liu, Richard He Bai, Huangjie Zheng, Yizhe Zhang, and Ruixiang Zhang. RSI-Inference: A long-horizon benchmark for self-improving inference systems. https://github.com/apple-aiml-research/ml-rsi-inference-bench, 2026. GitHub repository.

[29] Xiaoxuan Liu, Jiaxiang Yu, Jongseok Park, Ion Stoica, and Alvin Cheung. Speculative decoding: Performance or illusion?, 2026. URL https://arxiv.org/abs/2601.11580.

[30] Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, Zhiqi Huang, Huan Yuan, Suting Xu, Xinran Xu, Guokun Lai, Yanru Chen, Huabin Zheng, Junjie Yan, Jianlin Su, Yuxin Wu, Neo Y. Zhang, Zhilin Yang, Xinyu Zhou, Mingxing Zhang, and Jiezhong Qiu. MoBA: Mixture of block attention for long-context LLMs, 2025. URL https://arxiv.org/abs/2502.13189.

[31] Mao Luo, Hongbin Li, Feng Lin, Hanling Yi, and Zhe Huang. Agentic kernel optimization: Generating state-of-the-art gpu kernels without hand-written cuda, 2026. URL https://arxiv.org/abs/2608.14560.

[32] Michael McCabe, Payel Mukhopadhyay, Tanya Marwah, Bruno Regaldo-Saint Blancard, Francois Rozet, Cristiana Diaconu, Lucas Meyer, Kaze W. K. Wong, Hadi Sotoudeh, Alberto Bietti, Irina Espejo, Rio Fear, Siavash Golkar, Tom Hehir, Keiya Hirashima, Geraud Krawezik, Francois Lanusse, Rudy Morel, Ruben Ohana, Liam Parker, Mariel Pettee, Jeff Shen, Kyunghyun Cho, Miles Cranmer, and Shirley Ho. Walrus: A cross-domain foundation model for continuum dynamics, 2025. URL https://arxiv.org/abs/2511.15684.

[33] Meta PyTorch Team. Kernelagent: Hardware-guided gpu kernel optimization via multi-agent orchestration. PyTorch Blog, March 2026. URL https://pytorch.org/blog/kernelagent-hardware-guided-gpu-kernel-optimization-via-multi-agent-orchestration/.

[34] Minimax. Minimax-m3. Technical report, MiniMax AI, 2026. URL https://github.com/MiniMax-AI/MiniMax-M3.

[35] Moonshot. Kimi k3: Open frontier intelligence. Technical report, Moonshot AI, 2026. URL https://github.com/MoonshotAI/Kimi-K3/blob/main/k3_tech_report.pdf.

[36] Niklas Muennighoff, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Jacob Morrison, Sewon Min, Weijia Shi, Pete Walsh, Oyvind Tafjord, Nathan Lambert, Yuling Gu, Shane Arora, Akshita Bhagia, Dustin Schwenk, David Wadden, Alexander Wettig, Binyuan Hui, Tim Dettmers, Douwe Kiela, Ali Farhadi, Noah A. Smith, Pang Wei Koh, Amanpreet Singh, and Hannaneh Hajishirzi. OLMoE: Open mixture-of-experts language models, 2024. URL https://arxiv.org/abs/2409.02060.

[37] OpenAI. Mathematical manuscripts and supporting proof artifacts. GitHub repository, 2026. URL https://github.com/openai/math. Accessed: 2026-10-30.

[38] OpenAI. Finite time blowup for navier–stokes. OpenAI Mathematical Research Manuscript, 2026. URL https://cdn.openai.com/pdf/32d9f210-8b73-45e0-91bc-82a30aef8a9a/navier-stokes.pdf. Accessed: 2026-10-30.

[39] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels?, 2025. URL https://arxiv.org/abs/2502.10517.

[40] Saro Passaro, Gabriele Corso, Jeremy Wohlwend, Mateo Reveiz, Stephan Thaler, Vignesh Ram Somnath, Noah Getz, Tally Portnoi, Julien Roy, Hannes Stark, David Kwabi-Addo, Dominique Beaini, Tommi Jaakkola, and Regina Barzilay. Boltz-2: Towards accurate and efficient binding affinity prediction. *bioRxiv : the preprint server for biology*, 2025. doi: 10.1101/2025.06.14.659707.

[41] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In D. Song, M. Carbin, and T. Chen (eds.), *Proceedings of Machine Learning and Systems*, volume 5, pp. 606–624, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/file/c4be71ab8d24cdfb45e3d06dbfca2780-Paper-mlsys2023.pdf.

[42] Cole Ramos and Keith Lowery. Kerncap: Automated kernel extraction and isolation for amd gpus, 2026. URL https://arxiv.org/abs/2605.03208.

[43] Claude Science, Richard Shuai, Rohil Badkundri, Vincent Fan, Kilian Fatras, Lukas Jarosch, and Amir Shanehsazzadeh. Accelerating open-source biomolecular models with Claude. 2026.

[44] SemiAnalysis. InferenceX: Agentic inference benchmarks. https://inferencex.semianalysis.com/inference, 2026. Continuously updated inference benchmark dashboard. Accessed 2026-10-26.

[45] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017. URL https://arxiv.org/abs/1701.06538.

[46] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2019. URL https://arxiv.org/abs/1909.08053.

[47] Benjamin Spector, Jordan Juravsky, Stuart Sul, Owen Dugan, Dylan Lim, Dan Fu, Simran Arora, and Chris Ré. Look ma, no bubbles! designing a low-latency megakernel for llama-1b. Technical report, Stanford University, 2025. URL https://hazyresearch.stanford.edu/blog/2025-05-27-no-bubbles.

[48] Benjamin Spector, Jordan Juravsky, Stuart Sul, Dylan Lim, Owen Dugan, Simran Arora, and Chris Ré. We bought the whole gpu, so we're damn well going to use the whole gpu. Technical report, Stanford University, 2025. URL https://hazyresearch.stanford.edu/blog/2025-09-28-tp-llama-main.

[49] Benjamin F Spector, Simran Arora, Aaryan Singhal, Daniel Y Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels. *arXiv preprint arXiv:2410.20399*, 2024.

[50] Stuart H. Sul, Nash Brown, Henry Wildermuth, William Lin, Federico Cassano, and Christopher Ré. Mixture-of-kittens: MoE megakernel for NVL72s, 2026. URL https://arxiv.org/abs/2609.36070.

[51] Richard S. Sutton. The bitter lesson. Incomplete Ideas, March 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html. Accessed: 2026-10-30.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *31st Conference on Neural Information Processing Systems (NIPS 2017)*, 2017.

[53] Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. Geak: Introducing triton kernel ai agent & evaluation benchmarks, 2025. URL https://arxiv.org/abs/2507.23194.

[54] Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for gpu kernel performance optimization, 2025. URL https://arxiv.org/abs/2509.07506.

[55] Tianhao Xu, Yiming Liu, Xianglong Lu, Yijia Zhao, Xuting Zhou, Aichen Feng, Yiyi Chen, Yi Shen, Qin Zhou, Xumeng Chen, Ilya Sherstyuk, Haorui Li, Rishi Thakkar, Ben Hamm, Yuanzhe Li, Xue Huang, Wenpeng Wu, Anish Shanbhag, Harry Kim, Chuan Chen, and Junjie Lai. AIConfigurator: Lightning-fast configuration optimization for multi-framework LLM serving, 2026. URL https://arxiv.org/abs/2601.06288.

[56] Zihao Ye, Yingyi Huang, Hongyi Jin, Bohan Hou, Junru Shao, Zhongming Yu, Jinqi Chen, Meghan Cowan, Shiyi Cao, Shanli Xing, Hanfeng Chen, Vinod Grover, Tianqi Chen, and Luis Ceze. Cake: Compiler-agent co-design for frontier kernel evolution, 2026. URL https://arxiv.org/abs/2608.12629.

[57] Yang Yu, Peiyu Zang, Chi Hsu Tsai, Haiming Wu, Yixin Shen, Jialing Zhang, Haoyu Wang, Zhiyou Xiao, Jingze Shi, Yuyu Luo, Wentao Zhang, Chunlei Men, Guang Liu, and Yonghua Lin. Towards automated kernel generation in the era of llms, 2026. URL https://arxiv.org/abs/2601.15727.

[58] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention, 2025. URL https://arxiv.org/abs/2502.11089.

[59] Z.ai. Glm-5.3. Technical report, Z.ai, 2026. URL https://huggingface.co/zai-org/GLM-5.3.

[60] Genghan Zhang, Shaowei Zhu, Anjiang Wei, Zhenyu Song, Allen Nie, Zhen Jia, Nandita Vijaykumar, Yida Wang, and Kunle Olukotun. Accelopt: A self-improving llm agentic system for ai accelerator kernel optimization, 2026. URL https://arxiv.org/abs/2511.15915.

[61] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, 2024.

[62] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

[63] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. MegaScale-Infer: Serving mixture-of-experts at scale with disaggregated expert parallelism, 2025. URL https://arxiv.org/abs/2504.02263.

The appendix is organized as follows:

1. Appendix A provides further information on gathered experiments.
2. Appendix B describes the results on scientific ML models in more detail.
3. Appendix C provides additional analysis and ablations for results described in the main paper.
4. Appendix D details the cost model and parallelism search space.

A Experiment protocol

This section provides more detail on how we ran the baseline systems and how we produced our main results.

A.1 Baseline system protocols

We discuss the protocol for running experiments on existing agentic performance optimization systems. For these experiments, we focus on the throughput of the TKCC optimizer and fix the baseline parallelism configuration. The coding harness for all systems was Claude Code with Opus 5.5, reasoning high. The edit surface is kernel and forward pass changes without allowing any quantization. The base images are the same as TKCC and listed in Appendix A (GLM 5.3 SGLang on B200 and GLM 5.3 vLLM on MI355X). The evaluation workload was the same AgentX workload, and the signal given to the agent was decode TPS.

RSI-Inference Bench. We ran the default harness used by Apple's RSI-Inference Bench [28] (upstream commit `1a38072`) given a 10-hour budget and 8-hour verifier timeout. The base image is the same as the GLM 5.3 image used by TKCC (listed in Appendix A). We provide the RSI-Inference agent with the same edit surface as TKCC (i.e., the model's forward pass without any quantization allowed). We also use the same AgentX workload for evaluation and measure on output TPS. The result is provided in Figure 2.

A.2 Megakernel baseline protocols

We sweep output sequence length at ISL 32 and batch size 1. We compare to the following baselines on H100 and B200:

- We compare to megakernels across this repository: https://github.com/HazyResearch/Megakernels
- We compare to Mirage using this repository: https://github.com/mirage-project/mirage
- We compare to VDCores using this repository: https://github.com/vdcores/vdcores

A.3 InferenceX engine protocols

We ran TKCC on four different model architectures across five hardware platforms, starting from their best configuration at the time of publishing. Optimization loops ran for 24 hours of continuous optimization and ran on 2× the replica size (one for the fast persistent engine and one for the slower persistent engine). Here we describe the datasets and protocols used for each setting.

AgentX experiments. For the NVIDIA and AMD experiments, we evaluate end-to-end serving on SemiAnalysis InferenceX's AgentX benchmark and keep its methodology unchanged. That includes its pinned load generator (agentx-harness/AIPerf 1.0.12), benchmark command, public trace dataset (393 agentic-coding traces, seed 42), measurement window and scoring. We report InferenceX's metrics:

- **Throughput per GPU:** input tokens (cached tokens counted in full) plus output tokens, divided by wall-clock time and by the number of serving GPUs.
- **Per-user interactivity:** 1/P90 inter-token latency, and 1/median inter-token latency.

For Figure 7 and Figure 8, the baseline performance is sourced from the best InferenceX numbers at the time of writing. To check our protocol, we rerun it on InferenceX's published engine configurations and locally confirmed that it reproduces the published throughput exactly, and P90 interactivity to within about 2%. We include the P50 results in Figure 9 and Figure 10.

Trainium Evaluation. We optimize for a similar multi-turn synthetic dataset when evaluating Trainium.

- Prompts are built from public data: about 50% Python code (codeparrot-clean), about 30% Wikipedia prose (wikitext-103), and about 20% generated JSON/YAML/Markdown. They are tokenized with M3's tokenizer, seed 20261007.
- 65,792-token context with a 3,456 token system prompt shared across all users. Each request sees roughly 95% cache hit rate. Output sequence length is exactly 700 tokens. Sampling parameters are temperature 1.0, top-p 0.95. Each user sends its next turn as soon as the previous turn finishes.

Configurations. We report the engine configuration and images for each hardware platform below:

					Precision		Parallelism		
Model	Engine	Hardware	Chips	Conc.	Base	Experts	DP	Attention	MoE
DeepSeekv4.1-Flash	SGLang	B200	—	1–256	FP4	FP4	{1, 4}	{TP4, TP4 + DPA}	{TP4, EP4}
GLM-5.3	SGLang	B200	—	1–48	FP4	FP4	{1, 8}	{TP4, DPA}	TP4
Kimi K3	vLLM	B300	—	1–70	FP4	FP4	—	TP, DCP8	TP
MiniMax-M3.1-Flash	vLLM	B200	—	1–56	FP4	FP4	—	TP4	TP4
GLM-5.3	vLLM	MI355X	—	1–16	—	—	—	—	—
MiniMax-M3	vLLM Neuron	Trainium2	{4, 8, 16}	1–128	MXFP4/ FP8	{MXFP4, FP8}	{1, 2, 4}	{TP16, TP16 + CP4}	{EP2×TP8, EP4×TP8, EP8×TP8}

Table 4: Engines, hardware, precision, and parallelism configurations used to seed TKCC.

The baseline inference images used to start each compilation are as follows:

- **DeepSeekv4.1-Flash (SGLang, B200):** `lmsysorg/sglang:nightly-dev-cu13-20261021-c1c287a4@sha256:b0473834763b2753e069cb96c1acaf235ba7fd632ccd9bbeb44da809c437fdeb`
- **GLM-5.3 (SGLang, B200):** `lmsysorg/sglang:v0.5.22-cu130@sha256:e3a4f06001a29542fc4a8638536d9ffc5eb9c9942b08603c09b19a56eae4a1ae`
- **Kimi K3 (vLLM, B300):** `vllm/vllm-openai:nightly-dev-x86_64-cu13-3696c77`
- **Kimi K3 (vLLM, GB300):** `vllm/vllm-openai:nightly-e4a5b5537344b103ed10de7764378ef09604ef3d@sha256:6b90105076216e861cd30fae8828efef9263a0399137b894289f3b6a685bf4ff`
- **MiniMax-M3.1-Flash (vLLM, B200):** `vllm/vllm-openai:minimax-m31-flash@sha256:86f582dae25ea92802353727164ae4eb03b37b18baf9119b805ffaec2a7ef07c`
- **GLM-5.3 (vLLM, MI355X):** `vllm/vllm-openai-rocm:nightly-rocm100-29468dde8b515031dc6d4d9d06bf0a2fa0442098`
- **Minimax-M3 (vLLM-neuron, Trainium2):** PyTorch `2.11.0`, vLLM version `0.24.0`, vLLM-neuron upstream release `upstreamrelease-0.24.0.1.1.0`, neuron-cc compiler commit `2.27.5334.0+f702b353`, libtorch-neuronx-lite `2.11.0.1.0.1284+f49d8626`, aws-neuronx-runtime-lib `2.34.10.0`

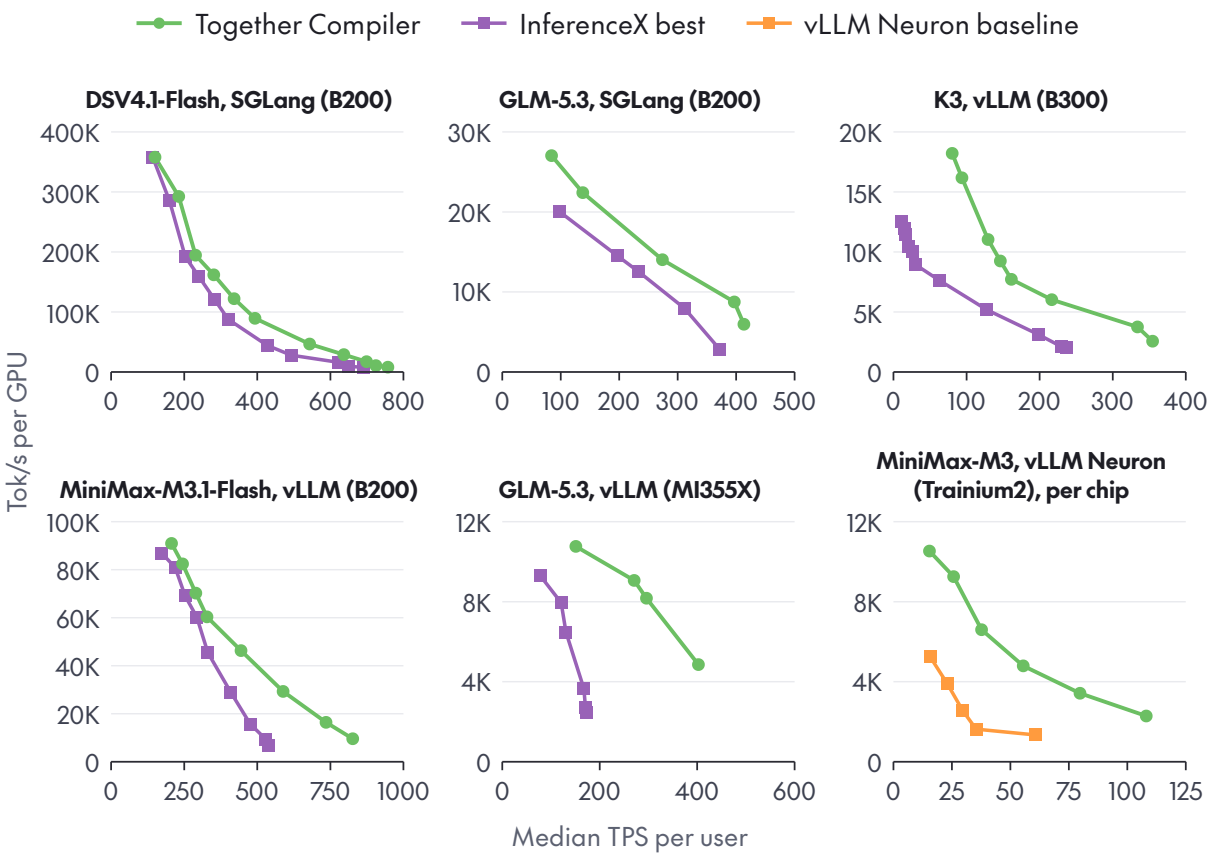


Figure 9: TKCC improves median TPS across production workloads.

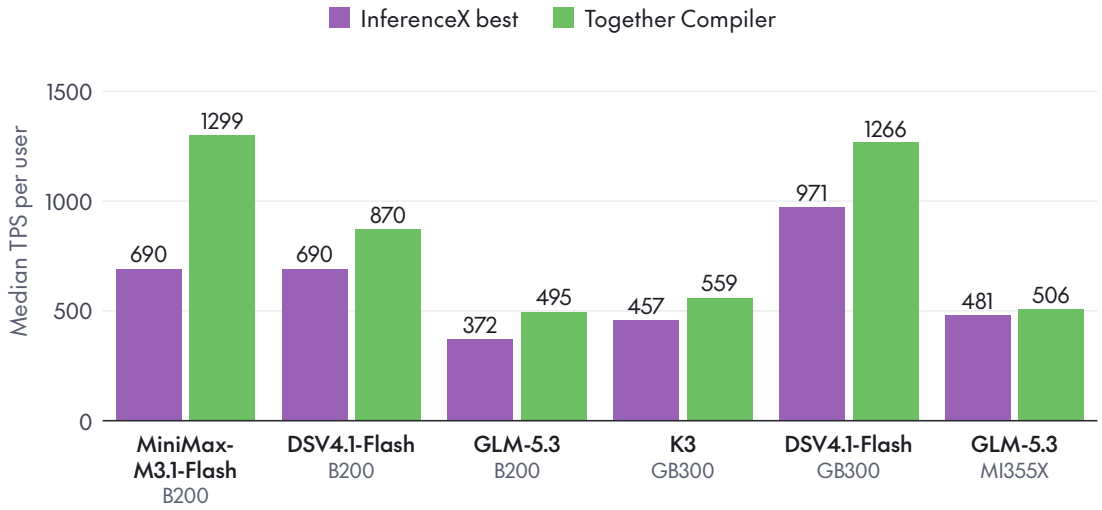


Figure 10: TKCC achieves over 1,299 TPS and 1,266 Median TPS on MiniMax-M3.1-Flash and DeepSeek-v4.1-Flash respectively.

B Diverse Workloads

We provide additional details on the set of diverse workloads we compare to a naive Claude Code baseline. For each, we adapted the model to standardize the weight loading, warmup, and candidate evaluation. Additionally, since multiple models relied on injected noise in their forward passes, we required bit exactness in outputs as well as in RNG states and other critical metadata.

Model Overviews

- **Aurora 1.5 Ensemble** is a weather foundation model that takes in atmospheric states, surface states, and forecasts future weather. Repository: https://github.com/microsoft/aurora/blob/b628d7ce93a8537e4844e41fc8aa2d44f522c674/aurora/model/aurora.py.
- **AIFS-ENS 2.0** is a probabilistic weather model that takes in two global weather states six hours apart, along with static and time-dependent forcing fields. It predicts the state six hours after the latest input. Repository: https://github.com/ecmwf/anemoi-core/blob/models-0.11.2/models/src/anemoi/models/models/ens_encoder_processor_decoder.py.
- **Boltz-2** is a biomolecular model that predicts 3D structures of proteins, nucleic acids, and small molecules. Repository: https://github.com/anthropics/uplifting-biomolecular-modeling/tree/f4f62fa6592ae4938d49b1757bea0cfeff9f468e/boltz2/opt.
- **Evo 2** is an autoregressive model that predicts nucleotides in DNA sequences. It follows a StripedHyena2 backbone. Repository: https://github.com/anthropics/uplifting-biomolecular-modeling/tree/f4f62fa6592ae4938d49b1757bea0cfeff9f468e/evo2/opt.
- **Chronos-2** is a T5 based, encoder only, time series foundation model that takes in a multivariate time series as input and outputs their forecasts. Repository: https://github.com/amazon-science/chronos-forecasting/blob/1224fafa91e1ecd6674f5204066b159e40b5e4a1/src/chronos/chronos2/model.py.
- **Panda-72M** time series foundation model is a PatchTST based, encoder-only transformer model. Repository: https://github.com/abao1999/panda/blob/c229e7c8c49cbe294458c44160248bb17856a715/panda/patchtst/patchtst.py.
- **Walrus** is a PDE foundation model that simulates the dynamics of fluid-like physical systems. Repository: https://github.com/PolymathicAI/walrus/blob/73f2cf6b4c5871c8f5434802725b55666c66b5ab/walrus/models/isotropic_model.py.
- **Poseidon-T**, unlike Walrus, takes in a single input state and an additional time step, and outputs the state after that amount of time. Repository: https://github.com/camlab-ethz/poseidon/blob/b8fa28f59bd7f7673323f28d11a12c6f3a215c61/scOT/model.py.

Evals Each model was tested for bit-exactness on an H100. The full list of evals can be found in Table 5.

TKCC vs Claude Code Our Claude Code baseline used very similar context to the TKCC setup, where we gave it the same baseline references along with detailed descriptions of our evals and correctness requirements. We didn't, however, give it access to a persistent engine, a standard model adapter, and correctness tests–all of which we expected Claude to implement itself if it found them useful. Our choice of baseline was deliberately chosen so as to provide insights into what advantages the TKCC design provided over pure prompting on Claude Code.

Model	Evaluation coverage	Reference	TKCC	Claude Code
AIFS-ENS 2.0	36 forecast cases: three initialization times, four members, two rollout states; 24 single-member and 12 paired-member cases. Matched Baseline-noise protocol.	Baseline	Passed	Failed
Aurora 1.5 Ensemble	54 ERA5 forecast cases: six initializations, lead times of 1–6 hours; 36 cases at batch size 1 and 18 at batch size 2. Matched per-member RNG.	Baseline	Passed	Passed
Boltz-2	Seven residue sizes, 200–1,400, using constructed barnase/barstar inputs with empty MSAs; 10 recycles, 200 diffusion steps, and 5 samples.	Anthropic exact	Passed	Passed
Evo 2	Evo 2-7B generation from a 3,000-base rCRS prompt: 16,000-token warmup and measured generation in each of three matched repetitions.	Anthropic exact	Passed	Passed
Chronos-2	Forward panel from 27 dataset configurations: 357 strata representing 5,973 original calls.	Baseline	Passed	Passed
Panda-72M	Chaotic-ODE forward panel: 154 strata representing 4,628 original calls.	Baseline	Passed	Passed
Walrus	The Well's `turbulent_radiative_layer_2D`: representative inputs for two batch-size buckets, weighted by 1,184 original calls.	Baseline	Passed	Passed
Poseidon-T	NS-PwC forward panel. Additional TKCC full-evaluator checks cover 8,640 samples, 240 trajectories, 135 batches, returned loss, and 39 non-timing metrics.	Baseline	Passed	Passed

Table 5: Verified bit-exact agreement with the designated reference on the tested workloads. Boltz-2 and Evo 2 use Anthropic's shipped exact baselines rather than unmodified Baseline.

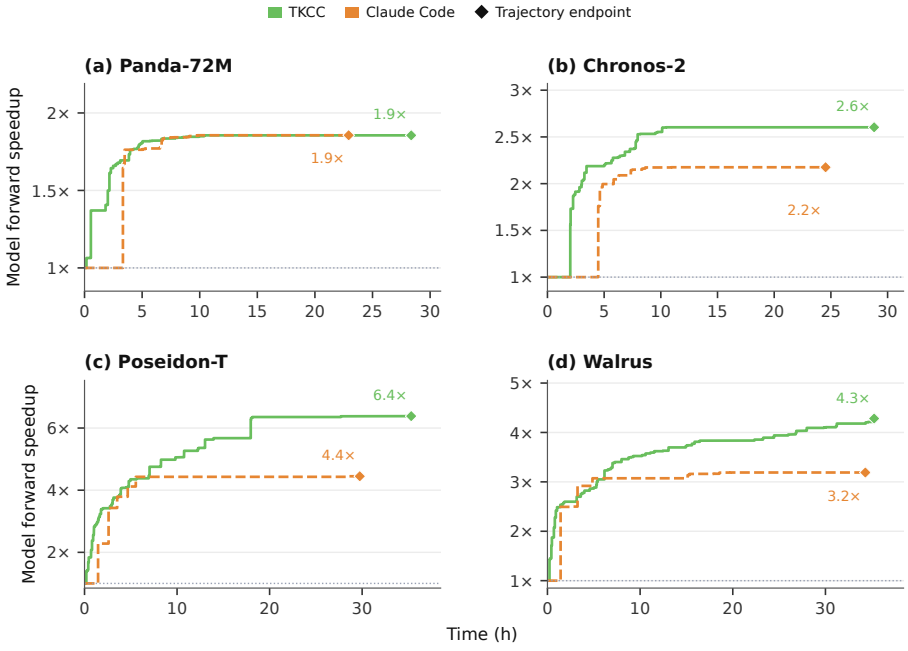


Figure 11: Best model-forward speedup found versus time for TKCC and Claude Code, with gaps between sessions removed, and final speedup of optimization marked by solid diamond. The significant time it took for Claude Code to obtain a candidate optimization is explained by the fact that it had to develop its own tooling/adapter for the model based off the context given to it by the codebase. TKCC and Claude Code performances are very similar for Panda, but we see a clear advantage in using TKCC over Claude Code for the other three models in terms of time and performance.

C Additional experiments and analysis

C.1 Measurement fidelity for data-dependent architectural components

Modern AI architectures use data-dependent execution paths governed by sparse top-k attentions and routed mixture-of-experts. Here we include examples of kernels generated based on random data distributions that may struggle to generalize to real data distributions during inference.

Sparse top-k attention. Below we include an example of an Opus 5.5 agent-authored sparse top-k attention kernel for GLM 5.3 on AMD MI355X. The kernel provided a speedup when benchmarked on random input data, but faces high run times when benchmarked on real data.

```
// tk_topk5: exact top-K (K = 2048) of one row of indexer scores (excerpt, simplified).
// Source: tkcc champion kernel topk5.hip (GLM-5.3 layers 1-2, MI350X).

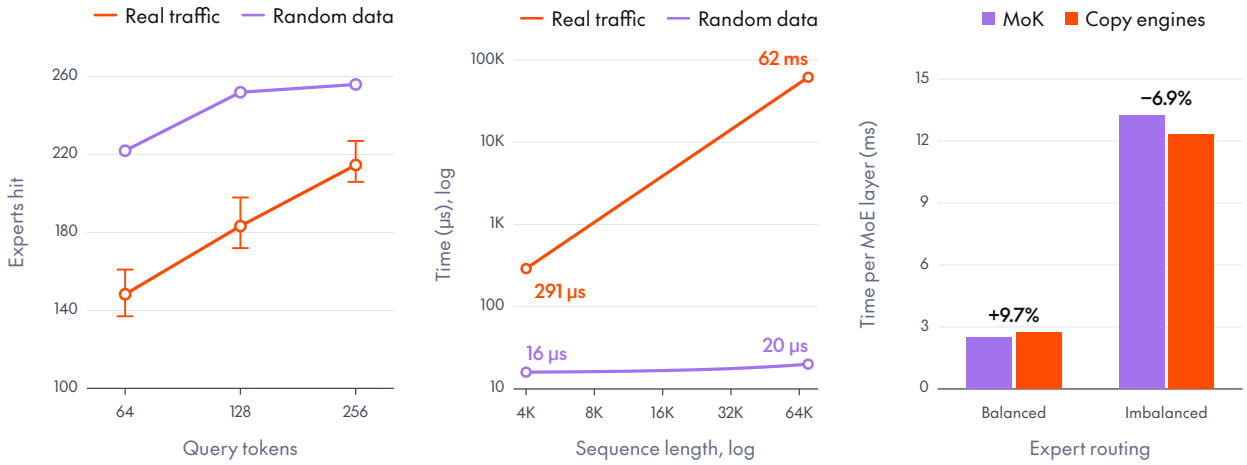
// 1) Bucket scores into 2048 histogram bins; find the threshold bin T holding the K-th largest score.
const uint32_t T  = cut[0];        // threshold bin
const int nAbove  = cut[1];        // scores in bins above T: all kept
const int cntT    = cut[2];        // scores tied in bin T (ReLU zeros: up to the whole context)
const int need    = K - nAbove;    // how many of bin T to keep

// 2) Rank every candidate in bin T against every other candidate: O(cntT^2).
if (cntT <= CAPG) {                // CAPG = 32768 staged candidates
  for (int c = c0 + w; c < c0 + tc; c += NT / 64) {         // each candidate (one per wave)
    const uint32_t kt = ckey[c];  const int it = cidx[c];
    int rank = 0;
    for (int j = lane; j < cntT; j += 64)                   // vs. all cntT candidates
      rank += (ckey[j] > kt) | ((ckey[j] == kt) & (cidx[j] < it));
    for (int o = 32; o > 0; o >>= 1) rank += __shfl_xor(rank, o);
    if (lane == 0 && rank < need) out[nAbove + rank] = it;
  }
} else {
  // 3) Overflow fallback: rescan the whole row per candidate: O(cntT * len).
  for (int i = beg + tid; i < end; i += NT) {
    if (hbin(lp[i]) != T) continue;
    const uint32_t kt = fkey(lp[i]);
    int rank = 0;
    for (int j = 0; j < len; ++j)                           // entire row (len = context length)
      if (hbin(lp[j]) == T)
        rank += (fkey(lp[j]) > kt) | ((fkey(lp[j]) == kt) & (j < i));
    if (rank < need) out[nAbove + rank] = i;
  }
}
```

Figure 12: A top-k kernel written by the agent that is fast on random data but slow on real data. The kernel scores every earlier token in the context and must return the indices of the k highest-scoring tokens. It groups the scores into histogram bins and keeps every bin above the one that contains the k-th largest score. Within that bin, it ranks each score by comparing it against every other score in the bin, breaking ties by lower token index, so the run time grows quadratically with the bin's size. Random test inputs spread across many bins, so the kernel took 20μs. Real scores contain many exact zeros from ReLU, which all fall into one bin, so the kernel took up to 62 ms at 70K context.

Mixture-of-experts. Below we include our analysis of Cursor's production mixture-of-experts (MoE) megakernel, which is used to train their models [50]. The reference repository at the time of this writing uses random inputs in the kernel benchmark harness. We observe that on random inputs, the proposed megakernel works well on GB300, but when there is more expert imbalance and large batch sizes, it is more efficient to use the copy engine for dispatch communication (Figure 13).

https://github.com/cursor/mixture-of-kittens



(a) Random data activates more experts than real traffic.

(b) Top-k kernel written assuming random data has quadratic slowdown.

(c) Copy engines beat MoK under imbalanced routing.

Figure 13: Copy engine outperforms a production-grade MoE megakernel (MoK [50]) in imbalanced routing.

C.2 Agent context management bitter lessons

In Section 4.4, we describe our ablations for context management. Here we include additional details on the expert-curated knowledge base and optimization strategies, and we provide analysis into how the agent behavior changes with versus without this expert-curated context. Authors of the expert context were involved in the authoring of FlashAttention, ThunderKittens, HipKittens, MegaKernels, and many production-grade kernels at Together AI.

Notably, most existing agentic optimization systems for end-to-end inference optimization use a knowledge bases [25, 53, 56], prescribe specific cost models or strategies for agents to follow during the optimization process (e.g., rank by end-to-end expected gain, roofline model, wall-clock speed) [4, 25, 53], or instruct the agents to use specific data structures for logging research and experimentation [14].

We first describe the contents of our knowledge base, strategies, and data structures and then analyze why agents may have struggled to use that context effectively.

C.2.1 Contents of the context shaping

Expert-curated knowledge base. In Figure 5 (Left) we ablate the use of a knowledge base. Surprisingly, we find that the knowledge base can sometimes hurts performance. We include the following artifacts in our knowledge base:

- **Fused-block recipes:** Based on prior production optimizations for each model, hardware architecture, batch size and context length, we include a schedule of the custom kernels that make up one transformer layer. Each recipe comes with a decision log explaining which operations were fused or kept separate, and why. We include per-session histories of tuning attempts, measurements and outcomes that led to each recipe.
- **Production kernels:** Kernel source code organized by operation (GEMM, grouped/MoE GEMM, attention, normalization, RoPE/QKV, quantization, top-k, collectives, routing), with one variant per architecture (NVIDIA Hopper/Blackwell, AMD CDNA3/CDNA4). Each kernel has shape metadata, measured performance and notes on design choices.
- **Reference kernels:** Example kernels collected from open-source kernel libraries, recorded as pinned pointers to an upstream commit and path, plus an index of GEMM shape configurations.

- **Kernel techniques:** Short write-ups of reusable optimization and correctness techniques in categories such as compute (MMA tiling, tensor-core instructions), memory (TMA, swizzling, caching), scheduling, occupancy, launch overhead, numerics, correctness under CUDA graphs, and benchmarking methodology.
- **Fusion patterns:** Descriptions of common fusion regions across operations, e.g. attention/FFN prologues, GEMM epilogues, MoE dispatch/finalize, and GEMM or attention fused with collective communication.
- **Research papers:** Distilled summaries of kernel and systems papers, with the full text attached.
- **Vendor documentation:** Local copies of the CUDA programming guide, PTX ISA and HIP/ROCm reference docs.
- **Third-party code:** Pinned copies of open-source kernel DSLs, kernel libraries and inference engines, with survey notes on each.

The knowledge base uses a graph structure. Every entry is a Markdown file with YAML metadata (architecture, data type, operation kind, bottleneck type) and typed links to related entries. Agents navigate from one entry to the next rather than querying a database. We found that with the inclusion of a knowledge base, agents often wasted time finding relevant kernels and ideas they could reuse, whereas without a knowledge base similar performing kernels or strategies were achieved independently in a shorter period of time (Figure 5a).

Expert-curated optimization strategies. In Figure 5b we ablate ensembling naïve Claude Code agents that are all instructed to make the forward pass as fast as possible against ensembling agents that are each assigned a distinct optimization objective. These distinct objectives include:

- **Quick wins:** Rank candidate changes with a written cost model, (share of step $\times$ confidence) / time to a measured result. Work from routing and stream changes, to drop-in kernels, to structural rewrites, banking each win before the next and re-ranking after every new best result.
- **Fusion:** Take ownership of the dominant kernels. For instance, agents replace closed-source GEMMs that cannot be fused with agent-authored open-source kernels. Agents then fuse across boundaries that a graph compiler like Inductor cannot reach (since closed-source kernels are opaque to them). This includes fusing GEMM prologues and epilogues, and the next layer's norm folded into the current layer's residual add.
- **Reduction:** Cut the decode step's launch count using only code already held (authored kernels and shipped fused ops). Target the runs of short copy, cast, norm and quantize launches between GEMMs, then tune each remaining launch per batch size.
- **Communication/compute overlap:** Build a ledger of the exposed time of every collective (all-reduce, MoE all-to-all), own each collective and its neighbouring GEMM, and measure each collective's floor on the machine. Hide collectives using producer-epilogue sends, per-tile signalling, split-grid persistent kernels, reduce-scatter + norm + all-gather fusion, and cross-layer pipelining.
- **Megakernel:** Grow a single persistent kernel step by step, one correct submission at a time: first the non-attention/non-MoE "shell" of a layer, then across layer boundaries, then absorbing attention and MoE (including in-kernel cross-GPU dispatch), and finally the whole decode step in one launch. Emphasis is on deadlock-safe grid synchronization and CUDA-graph safety.
- **Cost Model:** Treat benchmark submissions as the scarce resource. First measure the instrument's resolvable effect size E, and build a wall-time ledger that includes idle time and unexplained ("dark") time. Then try to refute each candidate before spending a submission on it: does the row exist, is it on the critical path, does the mechanism match? Submit only candidates whose estimated ceiling exceeds $2E$. Notably, this strategy is representative of the approach used in many existing agentic inference systems [25, 53]. As a reduced number of strategies can be validated in engine at a time, a system must decide which optimizations to prioritize.

We find that prescribing agents to fixed strategies often led to wasted agent cycles. Fixed strategies cannot represent a dynamically changing environment, and constraining a prescribed order ahead of time forces agents to explore less optimal paths. On AMD GLM 5.3, we saw the megakernel agent spend 24-hours, only to come up with 0.22% faster performance. In Figure 5b, we see that ensembling naive agents outperforms distinct expert strategies (6.2× vs. 4.9× after 10 trials), and that a single naive agent beats every prescribed strategy alone (4.8× vs. 3.1–4.5×).

Data-structure for logging research and experimentation progress. Figure 14 shows examples of the graph data structure that we proposed to help agents keep track of experiment threads.

We persisted data using a knowledge graph that managed an agent's context at the end of each trial boundary. Nodes are typed: *profile* nodes anchor measurements, *finding* nodes record observations derived from profiles or accumulated history, *experiment* nodes represent falsifiable hypotheses that close only when a measurement gate proves or disproves them, and *candidate* nodes are agent-generated submissions for new optimizations. Edges represent lineage through the graph, as optimizations stack on top of each other over time. To query information from the graph, we provided the agent with a command line interface (CLI), ensuring that future trials can easily parse through previous iteration's findings.

When comparing end-to-end compiler runs that start from the same baseline inference engine on GLM 5.3 AMD and GLM 5.3 NVIDIA, we find that agents spend 8% of time on bookkeeping, i.e. saving and looking up their past results in the graph through the CLI (∼12.6 seconds per call, since each call starts a new Python process), compared to $< 0.5\%$ of time on bookkeeping when we run without our logging data structure and agents instead keep their notes in a plain Markdown file.

C.3 Analysis of Together Compiler agent traces

This section provides additional analysis into how agents spend time within the system. We discuss the positive actions that agents take within the system and the opportunities that we observe for further improvement.

Throughput determines the ceiling of agent optimization. Together Compiler improves throughput by 252× over the baseline RSI-Inference system that repeatedly relaunches the inference engine to test each new optimization idea. However, we find that *throughput is still the primary outstanding bottleneck* to unlock further improvements in agent optimized inference systems. The TKCC measurement gates operate on the order of seconds, but still, we find that $> 50\%$ of agent time can be spent waiting on feedback from the gates, waiting behind experiments queued by other agents, or agents re-sending the same code to re-check results when the code produced marginal improvements or slowdowns (Table 6).

Model (platform)	Waiting for gate (%)	Thinking and writing code (%)	Other (%)
GLM-5.3 (AMD MI350X)	57.0	36.9	6.1
GLM-5.3 (NVIDIA B200)	55.0	35.7	9.3
MiniMax M3.1 Flash (NVIDIA B200)	52.7	40.4	6.9

Table 6: How Together Compiler agents spent their time. "Other" covers reading and editing files, compiling kernels, profiling and bookkeeping.

Optimizations discovered by hardware platform. Here we discuss the key optimizations agents implemented. Some patterns appeared across hardware platforms and others were hardware-specific. Common optimizations across platforms included:

- **Fusing communication into compute.** For example, GLM 5.3 on AMD runs across 8 GPUs, so every layer has several all-reduces, where the GPUs sum their partial results. The agents merged those into the kernels around them:
  - One kernel does the all-reduce, the residual add, the RMSNorm and the FP8 quantization. The matrix multiply before it writes its output straight into the all-reduce's shared buffer, which removes a copy.

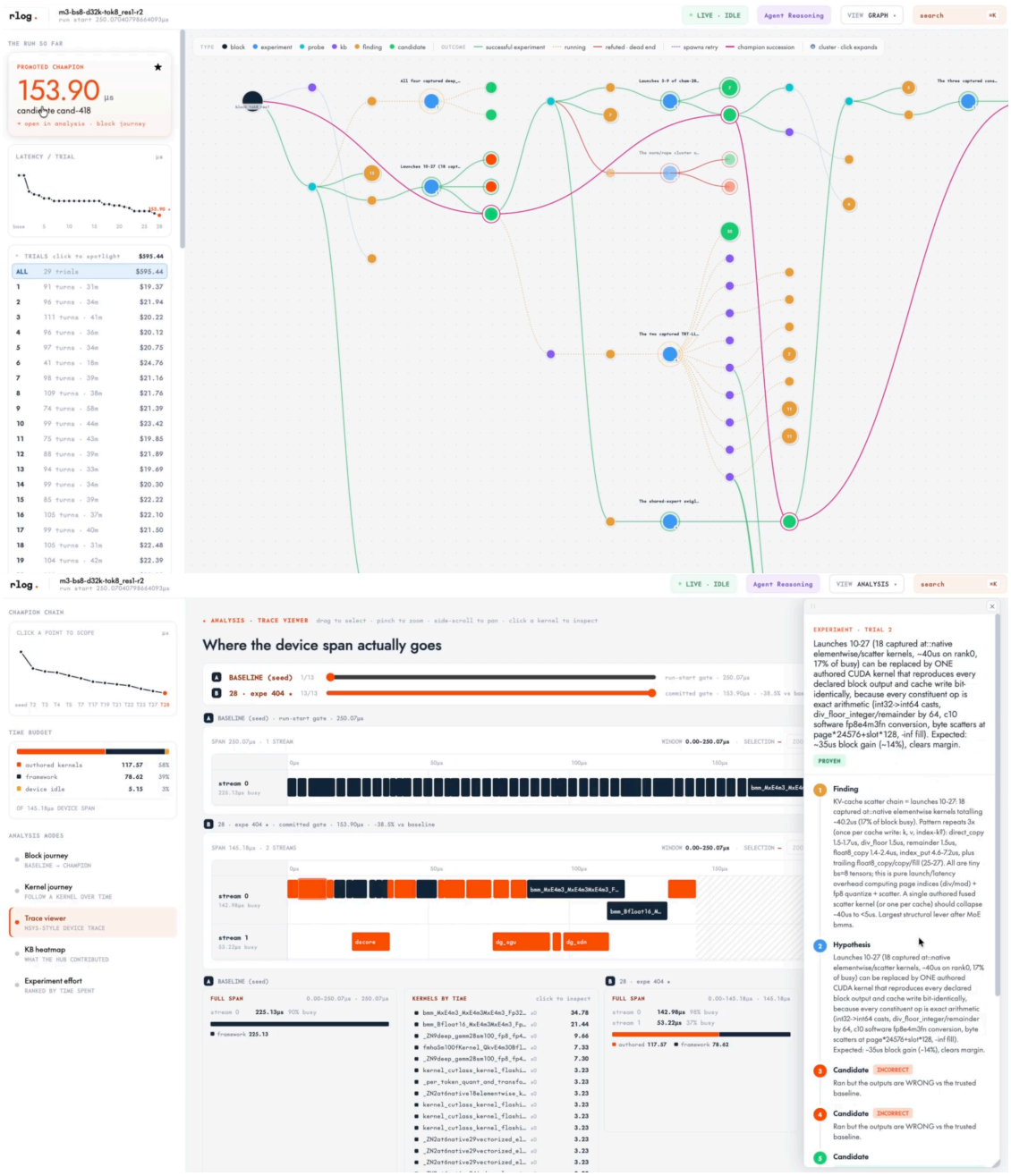


Figure 14: Examples of the experiment graph data structure that existed in earlier versions of TOGETHER COMPILER. **Above**, we show nodes of the parallel agent experiment graph where blue dots represent new agent hypotheses, red dots indicate optimizations that led to slower performance, and green dots indicate optimizations that led to faster performance. The left panel shows a chart of the model's layer timing across optimization trials; the table below the chart shows the dollars spent per trial. **Below**, we show the trace viewer for the baseline kernel sequence and the TKCC-generated kernel sequence. On the right panel, we can see the hypotheses, findings, and experiments that the agents run.

  - The MoE down-projection is fused with the next all-reduce.
  - The attention output projection is fused with its all-reduce and with part of the expert router.

  On NVIDIA, agents in all four models wrote their own Lamport-style all-reduce, in which each GPU polls the data buffer for a sentinel value instead of waiting on a separate ready flag, and fused it with the residual add and RMSNorm (and, on MiniMax M3.1 Flash, the quantization). On one GLM 5.3 optimization run, Lamport and NVLS all-gather/reduce-scatter kernels took the score from 1.00× (stock) to 1.26× in the first 1.5 hours, and on MiniMax M3.1 Flash the fused two-shot all-reduce ran about 2× faster than NCCL at decode.

- **Deleting dead or wasted operations.** For example, with GLM 5.3 on AMD, agents found that three copy/memset operations in the captured graph did nothing. On NVIDIA, four DeepSeek V4.1 agents independently found that the stock FlashMLA decode kernel padded 16 attention heads to 64, doing four times the needed work. Removing the padding was the largest decode-attention gain in any DeepSeek V4.1 optimization run.

Common areas where agents invested a lot of time, but struggled to realize any performance improvements include:

- **Modifying the attention kernel.** Agents spent significant effort on fusing neighboring kernels (e.g., norm, RoPE, output projection) into attention, or on changing how the work is split across GPU processors (e.g., split-KV). This applied both to the agents working on NVIDIA and AMD kernels.

- **Megakernel formation.** On GPUs, merging a layer, or long chains of dependent kernels, into one persistent kernel did not pay off. On AMD GLM 5.3 the result was only 0.22% faster than the existing best version, and on NVIDIA it gave no lasting gain. Each dependent step still needed a grid-wide barrier, which cost 1.5–3.5 μs inside a kernel versus 0.5–1 μs at a kernel boundary with PDL. Trainium2 was the exception: its final builds run each decoder layer's attention and MoE as a single fused decode kernel.

- **Overlapping communication with computation.** On AMD, running RCCL all-reduces at the same time as GEMMs lost on every concurrency. On NVIDIA, related ideas (overlapping two batches, pushing the output projection into the all-reduce, split-K GEMMs) were each 3-4× slower across MiniMax M3.1 Flash runs.

Some of the hardware-specific optimizations and failure modes are as follows:

- **NVIDIA:** On NVIDIA B200, B300 and GB300 GPUs, we optimized GLM 5.3, DeepSeek V4.1 Flash, Kimi K3 and MiniMax M3.1 Flash.

  - **Programmatic dependent launch (PDL).** The next kernel launches early and loads its weights while the previous one finishes, waiting only where it needs the previous output. On DeepSeek V4.1, this cut the fixed cost of each decode step by 9.7%.
  - **Reductions inside the NVLink switch.** NVIDIA's GPU-to-GPU switch can sum data itself (NVLS, `multimem`). This contributed to GLM 5.3's early step from 1.00× to 1.26× and to a Kimi K3 attention block that ran 1.68× faster than stock. For the very small transfers in decode, though, it was slower than the agents' Lamport all-reduce. AMD MI350X GPUs connect to each other directly, without such a switch.
  - **Blackwell low-precision tensor-core instructions.** The one agent-written matrix multiply that beat the vendor kernel used Blackwell's `tcgen05` instructions with NVFP4: a GLM 5.3 MoE down-projection that ran in 23.7 μs versus 34.1 μs for the vendor kernel.
  - **Thread-block clusters.** Groups of thread blocks that run together and share fast local memory. Switching to an existing cluster-based attention kernel from the vendor library was the largest single improvement in one GLM 5.3 optimization run.

- **Tuning vendor libraries instead of replacing them.** Changing the tile sizes of DeepGEMM's MoE kernel cut its time from 273 to 210 μs on DeepSeek V4.1. In contrast, agent-written replacements for vendor matrix multiply, attention and MoE kernels almost always lost, because the vendor kernels already ran at 78–100% of memory bandwidth.
- **GPU resets.** Agent code with illegal memory accesses put GPUs into a state that required a full reset (driver error Xid 154): 9 times during GLM 5.3, which led to 2 machines being removed from the cluster.

- **AMD:** On AMD, we optimized GLM 5.3 TP8 using FP8 precision weights.
  - **ROCm scheduling tricks.** When a captured HIP graph splits into parallel streams, ROCm's scheduler decides which branch stays on the main stream. Agents found that recording the main computation first at each split keeps the critical path on the main stream.
  - **Reusing AITER's all-reduce buffers.** AITER, AMD's kernel library, does its multi-GPU sums through special shared GPU-to-GPU buffers that bypass the cache. Agents had the preceding matrix multiply write its output directly into that buffer, which removes a copy and a kernel launch.
  - **Filling gaps in AMD's tuning tables.** For several of GLM 5.3's matrix shapes, Composable Kernel (CK) had no tuned settings, so the code fell back to slow defaults. Agents added split-K variants for those shapes.
  - **Compute-unit partitioning.** Attempts to reserve some compute units for communication while the rest ran computation (CU masks) were ignored on AMD.
  - **Per-chip timers are unusable for timing.** MI350X/MI355X is built from 8 compute dies (XCDs), and each has its own clock. Agents found that reading the GPU timer inside kernels gave meaningless per-operation numbers. Thus, attempts to schedule work by die ("XCD remapping") came out as ties.

- **Trainium2:** On Trainium2, we optimized MiniMax-M3 serving long multi-turn conversations.
  - **Fusing whole decoder layers into one kernel.** Custom NKI kernels run each decoder layer's attention and MoE as a single fused decode kernel. This is the opposite of what we saw on GPUs, where megakernels gave little or no gain. Other fused kernels cover the sparse-attention and MoE steps, the speculative-decoding verify and sampling step, and the KV-cache writes.
  - **Choosing the parallel layout for each load level.** The split of work across chips (tensor, expert, context and data parallelism) was tuned separately for each number of concurrent users using our contributed cost model. For example, splitting the KV cache across chips helped from about 4 users up but was slower at 1–2 users.
  - **Placing experts by measured routing.** At larger chip counts, experts are assigned to chips based on a measured profile of which experts tokens actually use, which balances the load across chips.
  - **Choosing what to compile as one program.** At low load, the whole decode step is compiled as a single program; at high load, one layer is compiled and reused across layers.

Closed model safety checks. Several times, closed source models could stop optimizing altogether due to safety concerns. For instance, after two pieces of agent code crashed GPUs, agents were banned from editing the model's recorded sequence of GPU steps directly. That stopped the crashes, but some of the improvements that had worked relied on exactly that technique, so they can't be redone.

D Parallelism cost model

One possible solution to the parallelism search problem is a full grid-search, where we enumerate a large number of possible parallelism schemes and run each scheme on GPU against a production-approximating workload. Theoretically, this approach would be of very high fidelity, given we actually collect timing information directly on the deployment for each parallelism configuration. However, grid-search has some notable downsides:

1. Results are dependent on software implementation for each parallelism strategy
2. Time consuming, requiring engine startup and a benchmarking run for each strategy

Therefore, we propose a cost-model-based approach to identify good parallelism candidates.

Notation Symbols used throughout:

N, $N_{\max}$	chips per replica; max replica upper bound (4096)
N_{dom}	chips in one scale-up domain
$TP_{\text{attn}}, CP_{\text{act}}, CP_{\text{kv}}, DP$	attention tensor, context (activations), context (KV cache), data parallel
TP_{dense}	dense tensor parallel
$EP_{\text{MoE}}, TP_{\text{MoE}}$	MoE expert parallel, MoE tensor parallel
ISL, OSL	input and output tokens per request
L	layer index
T_*, S_*, F_*	a time, a byte count, a FLOP count

Admissible schemes First we decide on the search space of admissible parallelism schemes. A scheme is admissible if all of the following rules hold:

1. Replica size
   (a) $N \leq N_{\max}$ (4096)
2. MoE
   (a) Expert Parallel: EP_{MoE} divides E, the number of routed experts, for every MoE block. EP_{MoE} is not constrained by the scale-up-domain size.
   (b) Tensor Parallel: TP_{MoE} divides every expert's width where $N \leq N_{dom}$
3. Dense
   (a) Tensor Parallel: TP_{dense} divides N
4. Attention
   (a) Tensor Parallel: TP_{attn} divides every attention head and projection.
   (b) Context Parallel (Activations): $TP_{\text{attn}} \cdot CP_{\text{act}}$ divides TP_{dense}
   (c) Context Parallel (KV Cache): CP_{kv} divides TP_{attn} sharing one KV head
   (d) Data Parallel: $TP_{\text{attn}} \cdot CP_{\text{act}}$ divides N, giving $DP = N/(TP_{\text{attn}} \cdot CP_{\text{act}})$

Cost model From there, we produced a cost model designed to estimate performance of a given model on a given chip and parallelism scheme. For each phase, the model computes the forward pass time from each model layer given parameters regarding compute FLOPs, HBM bandwidth, HBM capacity, and interconnect bandwidth, and it determines maximum concurrency allowed by memory capacity and latency target. From there, we can derive chip seconds per phase, which is then combined into one single throughput value: tokens/min per chip.

Forward pass time To calculate the forward pass time T_{pass}, let's define

- T_c^L, the time spent doing matrix compute at layer L
- D^L, the number of bytes — both weights and KV cache — loaded from HBM at layer L
- BW_{HBM}, the HBM memory bandwidth of the GPU
- $T_{\text{comm}}^{L,i}$, the time spent in GPU-to-GPU communication collective i at layer L

Within a layer, compute and memory reads are assumed to overlap perfectly and max is taken per layer. Communication is assumed to be sequential with computation. For prefill, we assume it is compute bound and drop the memory load component.

$$T_{\text{pass}}^{\text{decode}} = \sum_L \left[\max\left(T_c^L,\ \frac{D^L}{BW_{\text{HBM}}} \right) + \sum_i T_{\text{comm}}^{L,i} \right], \tag{1}$$

$$T_{\text{pass}}^{\text{prefill}} = \sum_L \left[T_c^L + \sum_i T_{\text{comm}}^{L,i} \right]. \tag{2}$$

Concurrency The concurrency C is determined by the largest number of requests that satisfy two independent limits, given some fixed tokens/sec per user:

1. Memory Limit C_{memory}: largest number of requests' KV cache that fit in HBM
2. Speed Limit C_{speed}: largest number of requests whose pass still meets the latency target set by fixed tokens/sec per user

$$C = \min\left(C_{\text{memory}},\ C_{\text{speed}} \right) \tag{3}$$

Disaggregated prefill and decode Prefill and decode are modeled as disaggregated, where each phase runs on separate hardware with its own sharding scheme. For each phase, we evaluate every admissible scheme and keep top k lowest chip-seconds per request. Among schemes within ϵ of the best, we select based on fewest chips.

$$\text{chip-s}_{\text{decode}} = \frac{N \cdot T_{\text{pass}}^{\text{decode}} \cdot \text{OSL}}{C}, \tag{4}$$

$$\text{chip-s}_{\text{prefill}} = \frac{N \cdot T_{\text{pass}}^{\text{prefill}}}{C}. \tag{5}$$

Throughput Finally, these are combined into a single measure of throughput, tokens/min per chip (TPM).

$$\text{TPM} = \frac{(\text{ISL} + \text{OSL}) \times 60}{\text{chip-s}_{\text{decode}} + \text{chip-s}_{\text{prefill}}}. \tag{6}$$

D.1 Attention

Attention is a key part of the model where there is a lot of variance from model to model. To properly account for all the variations, we characterize an attention layer with two variables: the family of an attention layer (based on the attention algorithm) and the key restrictions (decides which keys a query sees).

Families

1. **Softmax attention** scores each query against a set of keys ($QK^\top$) and sums their values (PV).
   (a) *GQA*: H query heads share H_{kv} key/value heads, with head widths d_{qk} (query/key) and d_v (value). H_{kv} changes what is cached and read, not the FLOPs: every query head still scores every key it sees.
   (b) *MLA*: the cache holds one latent of width d_c plus a shared RoPE key of width d_r. Whether the up-projection is folded into the query (*absorbed*) is set per phase: absorbed scores over $d_c + d_r$ and sums over d_c; unabsorbed scores over $d_{nope} + d_r$ and sums over d_v, where d_{nope} is the per-head key width without RoPE.
2. **Linear attention** (KDA) keeps a fixed $d_k \times d_v$ state per head instead of keys, so its work per token and its cache do not grow with context, and key restrictions do not apply.

Key restrictions A softmax layer may combine any of the following.

1. *Window* W: a query sees at most the last W tokens.
2. *Compression* r: stores one entry per r consecutive kv cache tokens instead of one per token.
3. *Sparse selection* s: a scoring function ranks the kv cache entries and the query attends to the top s of them (may be compressed or a block of tokens). The scorer is either a dedicated lightweight indexer or the attention scores from a coarser branch.

A window combined with a kv cache (compressed and/or sparse) is *additive*: the query attends to the union of its window and its selected kv cache entries in a single softmax or *gated*: each branch is attended separately and the outputs are mixed by learned gates.

Query–key pairs All softmax FLOPs follow from the number of query–key pairs. For a sequence with q new tokens after c cached ones, under a causal mask and a cap of m keys per query where m is set by the layer's key restrictions. The number of new tokens that see fewer than m keys is

$$u = \max\big(0,\ \min(q,\ m - c - 1)\big), \tag{7}$$

These u tokens see $c+1, \ldots, c+u$ keys and the remaining $q-u$ see exactly m. This gives us P, the total number of query key pairs for one sequence in this pass:

$$P(q, c; m) = u\,c + \frac{u(u+1)}{2} + (q-u)\,m. \tag{8}$$

Let W be the window size, r the compression ratio (tokens per kv cache entry; $r = 1$ without compression), and s the number of kv cache entries kept by sparse selection. Since s entries cover $s\,r$ tokens, the kv cache portion is counted in tokens with cap $s\,r$ and divided by r to give entries, a layer's query-pairs can be calculated:

$$P^L = \underbrace{P(q, c; W)}_{\text{window}} + \frac{1}{r}\,\underbrace{P(q, c;\ s\,r)}_{\text{long range}}, \tag{9}$$

dropping the window term without a window, the kv cache term for a sliding-window layer, and the cap $s\,r$ without sparse selection. Every attention step is then modeled as a matmul, per layer:

$$F^L_{QK} = 2\,P^L H\,d_{qk}, \qquad F^L_{PV} = 2\,P^L H\,d_v, \tag{10}$$

$$F^L_{idx} = 2\,\tfrac{1}{r}\,P(q, c;\ N_{cand}\,r)\,H_{idx}\,d_{idx}, \tag{11}$$

where H_{idx} and d_{idx} are the indexer's heads and key width, N_{cand} is the number of entries it scores, and the indexer's pairs ignore the window and the top-s cap.

KV cache We model two aspects of the KV cache:

1. Capacity: how many bytes one request *stores*, which sets the memory limit on concurrency C_{memory}
2. Reads: how many bytes one decode step *loads*, which is part of D^L

These differ for sparse layers, so we compute them separately. For a context of n_{ctx} tokens, let's define

- $n_{win} = \min(n_{ctx},\ W)$, the tokens held in the window
- $n_{kv} = \lceil n_{ctx}/r \rceil$, the entries of the kv cache
- w_{kv}, the values stored per entry: $H_{kv}(d_{qk} + d_v)$ for GQA ($H_{kv}\, d_{qk}$ with shared KV) and $d_c + d_r$ for MLA
- d_{idx}, the width of one index key, stored per kv cache entry for the sparse indexer
- b_{kv}, the bytes per element of the KV cache

Stored A layer stores its window and its kv cache, plus an index key per kv cache entry if it is sparse:

$$S^L_{\text{stored}} = b_{kv} \cdot \big(n_{win} + n_{kv}\big) \cdot w_{kv} + b_{kv} \cdot n_{kv} \cdot d_{idx}. \tag{12}$$

Some layers reuse an earlier layer's kv cache or index keys instead of keeping their own so they store only what they own.

Read per decode step A decode step reads the window and only the kv cache entries the query attends to, which is the top s for a sparse layer:

$$S^L_{\text{read}} = b_{kv} \cdot \big(n_{win} + \min(n_{kv},\ s)\big) \cdot w_{kv}. \tag{13}$$

A layer that runs the indexer also reads the index key of every entry it scores (all of them, or a shortlist of N_{cand}), adding $b_{kv} \cdot \min(n_{lv},\ N_{cand}) \cdot d_{idx}$. Sparse layers therefore read far less than they store.

D.2 Mixture of Experts

Similarly to attention, the MoE layer can be naively modeled fairly simply, but more complex modeling can account for effects which heavily impact the set of proposed sharding schemes. Some definitions that allow us to represent an MoE layer:

- B, the total number of tokens that engage in an MoE layer, before the top-k
- E, the number of total routed experts
- k, the number of experts each token routes to
- EP_{MoE}, the number of EP ranks, indexed by j
- $E_{\text{rank}} = E/EP_{\text{MoE}}$, the number of experts per rank
- L_{MoE}, the number of MoE layers in the model
- π_e, the probability a token picks expert e, such that $\sum_e \pi_e = k$
- S_{exp}, the total bytes of the routed experts per layer
- d_{lat}, an optional hidden dimension size used by latent MoE implementations
- F_{pair}, the FLOPs for one token through one routed expert

Modeling Expert Imbalance Due to our interest in producing both a theoretical model and a model based on empirical measurements, we ultimately decide to use π_e as input distribution for our modeling. That is, we enable both selecting a specific distribution to use as a theoretical model of the probability π_e and measuring that distribution directly from a model run. For the theoretical model, we by default use a Zipf distribution $\pi_{(i)} = \min(1, c/i^a)$ for the i-th most popular expert, with c set such that $\sum_i \pi_{(i)} = k$, $a = 0$ is uniform, and larger a is more skewed. For the empirical model we measure π_e per layer as the share of tokens routed to expert e. The expected number of a token's k picks which are routed to rank j is

$$\mu_j = \sum_{e \in j} \pi_e$$

Then, the proportion of all $B \cdot k$ (token, expert) pairs which are assigned to rank j is

$$f_j = \mu_j / k$$

Treating tokens as routing independently, and a token's k picks as independent of one another, the number of (token, expert) pairs which fall on rank j is

$$A_j \sim \text{Binomial}(B \cdot k,\ f_j)$$

If Z_j is the number of experts on rank j which are routed to by at least one token, then

$$\mathbb{E}[Z_j] = \sum_{e \in j} \left(1 - (1 - \pi_e)^B\right)$$

We take the approximation that each expert is routed to independently with probability $R_e = 1-(1-\pi_e)^B$, so Z_j is a sum of Bernoulli(R_e) over the experts on rank j. Because a layer waits for its slowest rank, we take the expected maximum over ranks, treating ranks as independent:

$$\mathbb{E}\big[\max_j X_j\big] = \sum_{x \geq 0} \Big(1 - \prod_j F_j(x)\Big),$$

where F_j is the CDF of A_j or Z_j. The routed experts' compute and memory time per layer under EP are then

$$t_{\text{comp}} = \frac{F_{\text{pair}} \cdot \mathbb{E}[\max_j A_j]}{\text{peak FLOP/s}}, \qquad t_{\text{mem}} = \frac{(S_{\text{exp}}/E) \cdot \mathbb{E}[\max_j Z_j]}{\text{HBM BW}},$$

which enter the per-layer $\max(t_{\text{mem}}, t_{\text{comp}})$ alongside the MoE layer's shared and router weights.

Expert Imbalance: MoE TP Under MoE TP, there is no expert imbalance, because experts and tokens are sharded across the full TP domain.

D.3 Collectives and network

Every parallelism scheme except pure data parallelism requires chips to exchange activations within each layer. We model each exchange as one of four standard collectives over the n chips that share the work, and compute its time $T^{L,i}_{\text{comm}}$ from the bytes S each chip moves. Collectives are never overlapped with compute or memory reads: they are added to the layer's time.

Which collectives each scheme needs Per layer:

Parallelism	Collective	Over
Attention TP	all-reduce of the output projection's partial sums	$n = TP_{\text{attn}}$
KV CP	all-gather queries, reduce-scatter partial outputs	$n = CP_{\text{kv}}$
Activation CP	cheaper of: all-gather the KV, or all-gather queries + reduce-scatter outputs	$n = CP_{\text{act}}$
Dense TP	all-gather the attention groups' activations, then all-reduce	$n = TP_{\text{dense}}$
Expert TP	all-reduce of partial expert outputs	$n = N$
Expert parallel	all-to-all dispatch, then all-to-all combine	$n = N$

Inside one scale-up domain The chips connected by the scale-up fabric form a domain of N_{dom} chips. A collective inside it is timed from the most specific data available for the chip:

1. *Measured curves*: time versus message size from our collective micros interpolated log-log between measured sizes. Below the smallest measured size the time is constant, we determine this to be the call's fixed cost.
2. *Library fit*: a latency α and scale up fabric bandwidth BW_{su} fitted per collective,

$$T_{\text{comm}}^{L,i} = \alpha + S \cdot \phi(n)/BW_{su}, \tag{14}$$

   with nccl-tests' bus factors $\phi(n) = 2(n-1)/n$ for all-reduce and $(n-1)/n$ for the others.
3. *Link model* (spec sheet only): the bytes S_{link} crossing the busiest link, of bandwidth $BW_{link}(n)$, plus a latency α_{hop} per hop over $h(n)$ hops,

$$T_{\text{comm}}^{L,i} = h(n) \cdot \alpha_{hop} \; + \; \frac{S_{link}}{BW_{link}(n)}, \tag{15}$$

   where the topology sets S_{link}, BW_{link} and the hop count h:

   - *Switch*: one hop at the chip's full bandwidth. A reduce-scatter sends the whole vector and the switch reduces it; an all-gather receives $(n-1)/n$ of it.
   - *Full mesh*: a direct link to every other chip, each $1/(N_{dom}-1)$ of the chip's bandwidth, so a collective over $n < N_{dom}$ chips uses only $n-1$ of them.
   - *Torus*: two links per axis, $h = \sum_{axis} \lfloor K_{axis}/2 \rfloor$ hops for the slice's most cube-like shape with axis lengths K_{axis}, and the bytes are carried once per hop.

Across domains: the expert all-to-all When expert parallelism spans more than one domain, each chip sends part of its tokens, S_{NIC} bytes, over its NIC (bandwidth BW_{NIC}) and the rest over the scale-up links, taking T_{inside}. The two run at the same time, so the slower one sets the time:

$$T_{a2a} = \max\Big(T_{\text{inside}}, \; \frac{S_{NIC}}{BW_{NIC}}\Big), \tag{16}$$

or their sum on PCIe cards, where one link carries both. Two dispatch styles are modeled, one per phase:

- *Decode* (low-latency kernels): each (token, expert) pair is sent directly to its chip.
- *Prefill* (throughput kernels): a token crosses the NIC once per destination node and is then forwarded over that node's links; the combine sums expert outputs before sending them back.

D.4 Customizability

Every input to the model is a separate, configurable parameter, from theoretical spec-sheet values to measured micro-benchmarks. Each can be swapped independently to trade simplicity for realism, and the best scheme and its cost are recomputed with every change.

Theoretical and measured specs Each chip is evaluated under four spec sources, from the vendor's spec sheet to measured values:

1. *theoretical*: the vendor spec sheet
2. *own micro*: our micro-benchmarks
3. *vendor tool*: vendor benchmarks (nvbandwidth, BabelStream, perftest, ...)
4. *vendor library*: vendor libraries (cuBLASLt, hipBLASLt, nccl-tests, ...)

Comparing each tier allows us to identify peak achievable performance that is not hindered by software immaturity.