CONJUGATE COMPILATION: SELF-IMPROVING AGENTIC CO-COMPILATION FOR SPECULATIVE DECODING

Anonymous authors
Paper under double-blind review

ABSTRACT

As LLM adoption surges, scaling model serving requires not only adding more GPUs but also unlocking the substantial capacity stranded in those already deployed. Speculative decoding brings two models into the serving process: a small draft model proposes tokens that a larger target model verifies. Yet compiling these models individually leaves their shared execution outside the compiler's optimization boundary. We therefore ask whether co-locating the draft and target models on the same GPU and co-compiling their execution can reclaim this stranded capacity. We introduce CONJUGATE COMPILATION, which reframes speculative decoding as a cross-model compilation problem and jointly optimizes the draft and target models within their shared execution rather than treating each in isolation. One of our key insights is that cross-model compilation should be informed by the measured performance of the co-located shared execution it produces: an optimization that improves a kernel in isolation need not improve the models when they execute together. To expand cross-model opportunities, we devise operand layouts that interleave blocks from both models within common tiles, co-locating GEMM and non-GEMM work while preserving their mathematics. We realize CONJUGATE COMPILATION through a self-improving agentic search system in which an LLM edits cross-model optimization plans using the compiler and GPU execution as its environment. Each proposed plan is compiled and evaluated through its co-located shared execution, whose measured score, together with kernel-level verdicts and compiler feedback, informs subsequent proposals, while Monte Carlo tree search accumulates rewards to guide exploration. The system also expands its search space and tuning budget when progress stalls and retains tuned kernels and verdicts for reuse across runs. Across six single-GPU speculative-decoding configurations on NVIDIA B300 and RTX 5090 GPUs, CONJUGATE COMPILATION achieves a geometric-mean per-round speedup of $1.37\times$ over conventional execution. Interleaved kernels raise this speedup to $1.45\times$, reaching $1.59\times$ in the best case. CONJUGATE COMPILATION also improves GPU utilization on average by $1.46\times$ on RTX 5090 and $1.50\times$ on B300. These results demonstrate the potential of cross-model conjugate compilation guided by measurements of the co-located shared execution.

1 INTRODUCTION

As demand for LLM serving grows, expanding GPU infrastructure is only one way to increase serving capacity (International Energy Agency, 2025). Another is to recover the substantial capacity that remains unused within existing deployments. Figure 1 shows one week of hourly GPU compute utilization measurements across enterprise users, illustrating how low average utilization persists despite variation across users and time. Across 6,000 measurements from 350 users collected between May and August 2026, average GPU compute utilization is 17.6%, with 94% of users averaging below 50% (Systalyze, 2026). These observations motivate us to examine how execution-informed compilation can turn more of the GPUs already deployed into useful capacity for model serving.

Turning this capacity into useful work depends in part on compilation, which determines how a model's operations and data are organized for GPU execution (Li et al., 2020). The common practice is to compile and optimize each model individually (Lu et al., 2025). Yet a single model's operations may provide too little parallel work to fill the GPU or leave compute resources underused while

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

waiting for data (Ding et al., 2021; Pope et al., 2023). This limitation motivates us to expand the compiler's optimization boundary: rather than optimize each model in isolation, we bring multiple models into concurrent execution on the same GPU and jointly optimize them informed by their co-located shared execution.

Speculative decoding provides a setting for exploring this opportunity because two models already cooperate to generate each response. A small draft model proposes several tokens, and a larger target model verifies them in a single forward pass, reducing the number of target-model passes needed for generation (Leviathan et al., 2023). This cooperation, however, does not by itself coordinate how their operations share GPU resources, nor does it bring their co-located shared execution within the scope of compiler optimization (Liu et al., 2025; Zhang et al., 2026). We therefore pose a central research question: without changing either model's underlying computations, can we coordinate and co-compile the draft and target models to reclaim the capacity left unused when they are optimized individually?

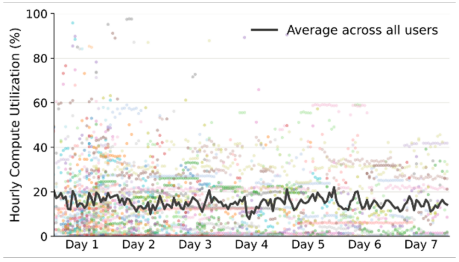

Figure 1: Hourly GPU compute utilization across enterprise users measured with Systalyze Utilyze from May to August 2026. The black line shows the hourly average across users; average utilization over the full dataset is 17.6%.

Speculative decoding provides a concrete setting for exploring this opportunity because two models already cooperate to generate each response. A small draft model proposes several tokens, and a larger target model verifies them in a single forward pass, reducing the number of target-model passes needed for generation (Leviathan et al., 2023). Prior approaches primarily exploit this structure as a scheduling, placement, or resource-allocation opportunity, either distributing draft and target execution across multiple GPUs or within a single GPU, while leaving their shared execution outside the compiler's optimization boundary (Liu et al., 2025; Zhang et al., 2026; Chen et al., 2026). We instead pose a central research question: what becomes possible when we co-locate them on the same GPU and let their shared execution directly inform cross-model compilation.

We introduce CONJUGATE COMPILATION, which reframes speculative decoding as a cross-model compilation problem by bringing the draft and target models into a common optimization boundary on the same GPU. Their co-located shared execution uses capacity left idle by either model, but also changes the conditions under which their operations execute. An optimization that improves a kernel in isolation may increase contention, disrupt beneficial overlap, or otherwise fail to improve the models when they execute together (Elvinger et al., 2025). One of our key insights is therefore that cross-model compilation should be informed by the measured performance of the co-located shared execution it produces. CONJUGATE COMPILATION uses this joint execution feedback to jointly optimize the draft and target models rather than optimizing each model independently.

Optimizing within this co-located shared execution requires a search that can respond to how the draft and target models actually behave together. We therefore contribute a self-improving agentic system that couples an LLM with Monte Carlo tree search (MCTS) (Tang et al., 2025), using the compiler and GPU execution as its environment. The LLM proposes edits to cross-model compilation plans, which the compiler checks, builds, and tunes before we measure the performance of the resulting co-located shared execution. We use MCTS as the agent's search medium, organizing explored plans and propagating their measured rewards to guide further exploration, while returning the complete execution score, kernel-level verdicts, and compiler feedback to the LLM to inform subsequent proposals. This coupling allows the agent to improve its proposals from accumulated execution evidence and directs exploration toward plans that improve the models together. When progress stalls, our system expands its search space and tuning budget, retaining tuned kernels and verdicts for reuse across runs.

To give the agentic system choices beyond coordinating separate kernels, we contribute an operand memory layout transformation that brings work from both models into a common kernel. By interleaving operand blocks within common tiles, we co-locate their work while preserving which operands participate in each computation and where its results belong. The interleaved kernels sup-

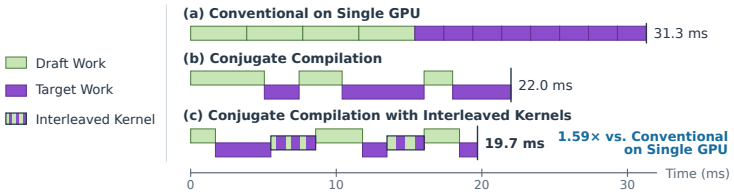

Figure 2: Illustration of (a) conventional execution of speculative decoding, (b) with CONJUGATE COMPILATION, and (c) with CONJUGATE COMPILATION plus Interleaved Kernels. Llama-3.2-1B (draft, $\gamma = 4$) and Llama-3.1-8B (target).

port GEMM and non-GEMM operation groups, including RMSNorm (Zhang & Sennrich, 2019) and residual addition (He et al., 2016), without changing their mathematics. The agent explores which operation groups to interleave and how much work to draw from each model, while the compiler optimizes these transformations as part of the broader cross-model compilation plan. Because an interleaved kernel that improves in isolation need not improve the models together, CONJUGATE COMPILATION evaluates these choices using the measured performance of the complete co-located shared execution.

We also contribute an operand memory layout transformation that brings work from both models into a common kernel. By interleaving operand blocks within common tiles, we co-locate their work while preserving which operands participate in each computation and where its results belong. The interleaved kernels support GEMM and non-GEMM operation groups, including RMSNorm (Zhang & Sennrich, 2019) and residual addition (He et al., 2016), without changing their mathematics. The agent explores which operation groups to interleave and how much work to draw from each model, while CONJUGATE COMPILATION optimizes these choices using the measured performance of their co-located shared execution.

We evaluate CONJUGATE COMPILATION across six single-GPU speculative-decoding configurations on NVIDIA RTX 5090 and B300 GPUs. Compared with compiling and optimizing each model in isolation, CONJUGATE COMPILATION achieves a geometric-mean per-round speedup of $1.37\times$. Adding selectively interleaved kernels raises this speedup to $1.45\times$, reaching $1.59\times$ in the best case. CONJUGATE COMPILATION also improves GPU utilization by $1.46\times$ on RTX 5090 and $1.50\times$ on B300, on average, respectively. Together, these results show that cross-model conjugate compilation, informed by the measured performance of co-located shared execution, can recover stranded capacity by jointly optimizing multiple models on a single GPU.

2 CONJUGATE COMPILATION: CO-OPTIMIZING SHARED EXECUTION

2.1 BRINGING DRAFTING AND VERIFICATION INTO SHARED EXECUTION

Figure 2(a) shows conventional execution of speculative decoding on a single GPU. The green blocks denote draft-model segments and the purple blocks denote target-model segments, with horizontal position representing execution time. The γ draft steps complete before target verification begins, so the green and purple work does not overlap. If one draft step takes T_D and target verification takes T_V, their execution takes

$$T_{\text{conventional}} = \gamma T_D + T_V. \tag{1}$$

This ordering reflects a true dependency within each batch: target verification cannot begin until that batch's draft tokens are available. The dependency, however, does not extend across batches. While the target verifies the proposals for batch b, the draft model can generate proposals for batch $b + 1$.

We use a dependency-preserving multi-stream co-scheduler to expose this concurrency on the same GPU. Figure 2(b) shows the resulting co-scheduled execution. Where purple target and green draft blocks overlap along the time axis, segments from the two models execute concurrently; gaps indicate intervals in which only one stream is executing. We divide both models into ordered segments, ranging from layers and attention/MLP halves down to individual projection stages, and submit the target and draft segments to separate CUDA streams. Segments within each stream execute in order, while segments on different streams may execute concurrently unless a cross-stream dependency orders them. Our co-scheduler inserts CUDA events (NVIDIA, a) only where such coordination is required, preserving the dependencies within each model while exposing legal overlap.

Formally, let $G_\sigma = (V, E_\sigma)$ denote the co-scheduled round, where V contains the draft and target segments and E_σ contains their intra-model dependencies together with any cross-stream edges σ. Execution preserves every edge:

$$t_{\text{start}}(v) \geq t_{\text{finish}}(u), \qquad \forall (u, v) \in E_\sigma. \tag{2}$$

For co-scheduling alone, $\sigma = \emptyset$; cross-model compilation decisions may later introduce cross-stream edges when their placement requires them. We capture both streams and their dependencies in a single CUDA graph and replay the complete co-scheduled round as one unit, removing host launch overhead while preserving the dependency structure. This co-scheduled round realizes the co-located shared execution that CONJUGATE COMPILATION optimizes. Its measured behavior provides the execution feedback for the cross-model compilation decisions introduced next.

2.2 CROSS-MODEL CONJUGATE COMPILATION

Once the draft and target models are co-located, compiler decisions interact through their shared execution. CONJUGATE COMPILATION therefore jointly optimizes cross-model compilation plans and judges them in the execution they change. This optimization applies even when the two models retain separate kernels on separate CUDA streams. Their kernel configurations can still be selected according to their effect on the complete shared execution rather than their isolated latency. We represent these choices through a *joint optimization plan* θ, which captures the cross-model compiler decisions realized within the shared execution.

Realizing a joint optimization plan involves four components: the *conjugated operation groups* selected for joint execution, the *interleaving proportions* that determine how much work each operation contributes to this joint execution, the *kernel configurations* used to execute the resulting work, and the *scheduling arrangement* that places it into the co-scheduled execution.

Cross-model conjugated operation groups. The agent represents the cross-model transformations in a plan as

$$\mathcal{G}(\theta) = \{g_1, \ldots, g_n\}, \tag{3}$$

where each conjugated operation group

$$g = \left(f, \{o_k\}_{k=1}^{K_g}\right), \qquad 2 \leq K_g \leq K_{\max}, \tag{4}$$

contains K_g independent operations from the draft and target models that share a kernel family f. These families include general matrix multiplication (GEMM) as well as row operations such as RMSNorm, residual addition, and softmax. The grouped operations may differ in role and shape, while operations outside the selected groups retain separate kernels.

Interleaving proportions. In our implementation, a selected group can be realized as a cross-model *interleaved kernel* that executes work from multiple operations in one kernel launch. The striped blocks in Figure 2(c) show where selected draft and target operations are replaced by such kernels, while the remaining purple and green blocks continue to execute as separate kernels. For GEMM, each member is decomposed into output-tile and reduction-slice units, and the fraction of work assigned to member k follows its share of the group's arithmetic work:

$$\pi_k = \frac{F_k}{\sum_{r=1}^{K_g} F_r}, \tag{5}$$

where F_k is the arithmetic work of operation o_k. The interleaving proportion is therefore derived from the selected group rather than searched independently. A descriptor table identifies each unit's member, tile coordinates, reduction slice, and operand addresses, allowing operations with unrelated shapes and separately stored operands to execute together. Mixing their work within one launch can fill partial waves and other execution capacity that the individual operations would leave underutilized. Each unit nevertheless reads and accumulates only its own operation's data, preserving every member's computation. The same principle extends to non-GEMM operations such as RMSNorm and residual addition. Interleaving is therefore one cross-model transformation available to CONJUGATE COMPILATION rather than the definition of the joint optimization plan itself.

Kernel configurations. For both separate and conjugated kernels, CONJUGATE COMPILATION exposes multiple backend kernel configurations through TVM. Rather than selecting configurations

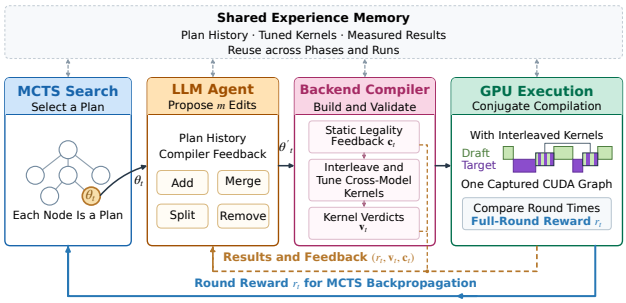

Figure 3: Conjugate compilation as a self-improving agentic loop.

solely by isolated kernel latency, CONJUGATE COMPILATION evaluates candidate configurations within the co-located shared execution and uses their effect on complete execution performance to guide selection. For interleaved GEMMs, TVM explores the same low-level schedule space available to the corresponding separate kernels, while row-operation kernels are selected from the schedules exposed by their templates. Kernel configuration is therefore part of the execution-informed optimization: even when the draft and target remain separate kernels, their implementations can be chosen according to how they perform together.

Scheduling arrangement. For a given co-scheduled execution, the execution scheme in §2.1 fixes the current segmentation, stream assignment, and dependencies that determine where draft and target work may legally overlap. The conjugated operation groups determine which target segment i and draft segment j must rendezvous for joint execution. These pairings induce a joint map $i \mapsto j$ that must be injective and strictly increasing along both execution chains, preserving their relative order and preventing cycles. A joint node $J(i, j)$ is the scheduling container for this co-compiled work. Selected groups may be realized by interleaved kernels, shown as striped blocks in Figure 2(c), while other operations retain separate kernels whose execution remains part of the same shared optimization context. The target stream joins the corresponding draft work before entering $J(i, j)$, and completion of the joint node releases subsequent draft work through a fork.

2.3 JUDGING A JOINT OPTIMIZATION PLAN IN THE EXECUTION IT CHANGES

Let Θ_{legal} denote the statically legal joint optimization plans defined in §2.2. Within this shared execution, CONJUGATE COMPILATION seeks

$$\theta^\star \in \arg\min_{\theta \in \Theta_{\text{legal}}} T_{\text{round}}(\theta), \tag{6}$$

where $T_{\text{round}}(\theta)$ is the measured latency of the complete shared execution produced by joint optimization plan θ. After the compiler realizes a plan, we additionally verify its numerical outputs against the reference execution before considering its performance.

Crucially, this objective does not decompose into isolated kernel improvements. Let $\tau(o)$ denote the isolated latency of operation o, and $\tau(g)$ the latency of the cross-model implementation of conjugated group g. In general,

$$T_{\text{round}}(\theta) \neq T_{\text{round}}(\emptyset) - \sum_{g \in \mathcal{G}(\theta)} \left(\sum_{o_k \in g} \tau(o_k) - \tau(g) \right). \tag{7}$$

Shared execution may already hide part of an operation's latency, while a decision in the joint optimization plan can change resource contention, overlap, and synchronization elsewhere in the round. A faster conjugated group therefore does not necessarily produce a faster shared execution.

We consequently judge each joint optimization plan by the complete shared execution it produces, including the synchronization and transformation overhead introduced by cross-model compilation. Because the decisions in a joint optimization plan interact, their benefit must be learned from measured shared execution rather than from isolated kernel improvements. §3 describes how our self-improving agentic system searches this execution-dependent space using execution feedback.

3 SELF-IMPROVING AGENTIC SEARCH FOR CONJUGATE COMPILATION

The joint optimization space is combinatorial and expensive to explore: the value of a plan is known only after compilation, and the GPU measures the resulting shared execution. In addition, each evaluation produces rich information—including compiler legality feedback, kernel-level verdicts, and complete shared-execution performance—that can guide subsequent optimization. We therefore build the self-improving agentic loop in Figure 3. MCTS selects a previously explored joint optimization plan to refine, the LLM agent proposes structured refinements to that plan, and the compiler and GPU evaluate the resulting candidates. The measured round reward updates the search tree, while compiler feedback and kernel-level verdicts inform subsequent LLM proposals.

3.1 MCTS-GUIDED PLAN SELECTION

We organize explored joint optimization plans as a Monte Carlo tree search (MCTS), shown on the left of Figure 3. Each node represents a complete joint optimization plan, with the empty plan $\emptyset$, corresponding to the co-scheduling-only baseline, as the root, and each edge represents one refinement proposed by the LLM agent. MCTS determines which explored plan should be refined next, balancing further exploration of promising plans against revisiting less-explored alternatives.

Selection follows the upper confidence bound for trees,

$$\theta_t = \arg \max_{\theta' \in \mathrm{ch}(\theta)} \left[\overline{Q}(\theta') + \beta \sqrt{\frac{\ln(N(\theta)+1)}{N(\theta')}} \right], \tag{8}$$

where $\mathrm{ch}(\theta)$ denotes the children of θ, $N(\cdot)$ counts visits, and $\overline{Q}$ is the normalized mean backed-up reward. Progressive widening limits how quickly new alternatives are introduced, allowing a node to gain a child only while

$$|\,\mathrm{ch}(\theta)| < \lceil \kappa\,(N(\theta)+1)^{\alpha} \rceil\,. \tag{9}$$

Once MCTS selects θ_t, the LLM proposes m candidate refinements (§3.2), which become children of θ_t as progressive widening permits. Because each node is an executable joint optimization plan, we evaluate a child directly in the complete shared execution rather than through simulated rollouts and propagate its measured reward through the tree. Candidates that fail compilation or numerical validation receive the lowest reward, while their failure information remains available to subsequent proposals.

3.2 LLM-GUIDED JOINT OPTIMIZATION PROPOSALS

Given the plan θ_t selected by MCTS, the LLM agent proposes new joint optimization plans by refining its conjugated operation groups. The LLM does not generate GPU kernels directly; instead, it operates on the joint optimization plan abstraction defined in §2.2. The agent has four actions: *add* creates a new conjugated operation group, *merge* combines existing groups, *split* separates members of a group, and *remove* dissolves a group. Each action identifies operations and ranges of target layers and draft steps, allowing one proposal to apply the same transformation across multiple locations in the shared execution.

The agent returns m distinct proposals. Edit types, operation identifiers, ranges, and legal placements are drawn from enumerated choices supplied by the compiler, making each proposal either directly interpretable or rejectable with a specific feedback code. The LLM proposes these refinements from accumulated optimization experience rather than from the selected plan alone. Its prompt includes the selected plan and its recent ancestors, their measured round performance, accumulated kernel-level verdicts, compiler feedback, legal placements, and measurements of representative draft–target segment pairs executed separately, concurrently, and conventionally. By comparing a plan with its ancestors and the feedback they received, the agent can associate changes in measured shared execution with the decisions that produced them and use that evidence to guide subsequent proposals. The LLM and MCTS thus serve complementary roles: the LLM converts compiler and execution evidence into informed edits, while MCTS determines which explored plans should be refined and which alternatives should be revisited.

3.3 Learning from Compiler and GPU Feedback

The compiler and GPU form the agent's environment. Evaluating a candidate plan θ'_t returns

$$\mathcal{E}(\theta'_t) = (r_t,\, \mathbf{v}_t,\, \mathbf{c}_t)\,, \tag{10}$$

where r_t is the measured reward, $\mathbf{v}_t$ contains kernel verdicts, and $\mathbf{c}_t$ contains compiler feedback.

The compiler first checks the static legality constraints from §2.2. A proposal is rejected before compilation if, for example, its conjugated operations are dependent, its joint placement introduces a cycle, a group spans more than one segment of either model, or its required operands exceed the memory budget. Each rejection is returned to the agent through a specific code in $\mathbf{c}_t$.

For every new conjugated group that passes these checks, the compiler builds and tunes its cross-model kernel and applies a kernel-gain gate before inserting it into the round. The candidate kernel and the separately compiled kernels of its members are replayed as captured CUDA graphs in alternating timing blocks, with weights rotated beyond the L2 cache between measurements. A group g is admitted only if

$$v(g) = \frac{\sum_k \tau(o_k)}{\widehat{\tau}(g)} > 1 + \max(\delta, z\widehat{s})\,, \tag{11}$$

where $\widehat{\tau}(g)$ includes all work launched by the group, δ is a minimum gain margin, and $\widehat{s}$ is the standard error of the paired timing relative to $\widehat{\tau}(g)$. Every result, whether the group passes or fails, enters $\mathbf{v}_t$ so the agent can learn which conjugations produce useful local improvements.

Passing the gate establishes only that a conjugated group is locally promising. The resulting joint optimization plan is therefore compiled into the complete co-scheduled round and checked against the reference execution for every draft step. We require relative agreement within 10^{-2} and permit an argmax difference only when the reference logits themselves form a near tie; failing positions or joint sets are quarantined and reported to the agent.

We then measure each candidate joint optimization plan on the complete co-scheduled round, using co-scheduling without co-compilation as the anchor. Each timing block replays the anchor, candidate, placement control, candidate, and anchor in that order. The placement control preserves the candidate's join and fork dependencies but executes the original separately compiled kernels. Comparing the anchor with this control measures the effect of the scheduling arrangement; comparing the control with the candidate measures the additional effect of the compiler transformations.

We compute the candidate's full-round speedup over the anchor as the median of the paired anchor-to-candidate latency ratios. To express the search reward relative to baseline, we multiply this speedup by the anchor's speedup over the faster of two conventional baselines: the compiled models executed back-to-back, or sequentially within one captured CUDA graph. This scaling factor is fixed within a configuration, so it does not change how MCTS ranks candidate plans.

The measured full-round reward updates the MCTS search tree, while kernel-level verdicts and compiler feedback enter the LLM's accumulated optimization experience to guide subsequent proposals. Tuned kernels, kernel verdicts, compiled joint nodes, rejection outcomes, and search checkpoints persist across escalation phases and resumed runs. The system therefore improves through accumulated compiler and GPU experience, without modifying the weights of the LLM agent. Appendix A details search-space expansion and final plan acceptance.

4 Evaluation

4.1 Methodology

Benchmarks. We evaluate four target-draft pairs on NVIDIA B300 GPU: Llama-3.1-8B/Llama-3.2-1B, Qwen3-14B/Qwen3-0.6B, Qwen3-32B/Qwen3-Instruct-2507-4B, and Llama-3.1-70B/Llama-3.1-8B (Meta, 2024a;b; Qwen Team, 2025). We evaluate Llama-3.2-3B/Llama-3.2-1B and Qwen3-Instruct-2507-4B/Qwen3-0.6B on NVIDIA RTX 5090 GPU. All experiments use batch size 8 (Qian et al., 2024), speculation length $\gamma \leq 6$ (Leviathan et al., 2023; Qian et al., 2024; Liu et al., 2025), a 1024-token input sequence length (Patel et al., 2024), and FP16 precision (Qian et al., 2024).

Evaluated systems. Conventional represents the current practice for deploying speculative-decoding models, where the draft and target models are compiled independently and executed sequentially (Leviathan et al., 2023; Kwon et al., 2023). Conjugate Compilation co-locates

the draft and target models on the same GPU and jointly optimizes cross-model compilation plans using measurements of their shared execution through the closed-loop search of §3. A dependency-preserving multi-stream schedule realizes legal concurrent execution, including between non-matching operators such as Tensor Core GEMMs and CUDA Core operations. CONJUGATE COMPILATION WITH INTERLEAVED KERNELS additionally replaces selected compatible operation groups with kernels that interleave their work within one launch.

Implementation. We implement CONJUGATE COMPILATION in Python using Apache TVM v0.25 (Chen et al., 2018) to import PyTorch (Paszke et al., 2019) models, lower their computation, and generate CUDA code. Our search forms a closed loop between the agent, compiler, and GPU, where agent proposes cross-model optimization plans, compiler turns them into joint nodes and interleaved kernels that it tunes and builds on the TVM backend, and measurements and compiler feedback inform subsequent proposals. We organize the evaluated plans in a Monte Carlo tree search (MCTS) (Browne et al., 2012), which propagates their measured rewards through the tree to prioritize subsequent exploration. We use OpenAI's gpt-5.6-luna (OpenAI, 2026) as the agent.

Measurements. We run all experiments with CUDA 12.8 on NVIDIA B300 SXM6 and RTX 5090 GPUs. We run experiments 50 times, report the median, and normalize all results to CONVENTIONAL. For GPU utilization we use Nsight Compute (NVIDIA, c) and collect kernel traces with CUPTI (NVIDIA, b).

4.2 EXPERIMENTAL RESULTS

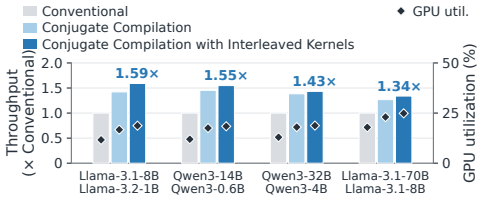

Figure 4: Throughput relative to CONVENTIONAL (left y-axis) and GPU utilization (right y-axis) on NVIDIA B300.

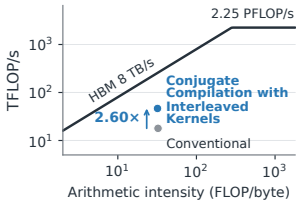

Figure 5: Roofline analysis of Llama down-projection kernel on B300 GPU.

Throughput improvement. Figure 4 shows target-model verification throughput relative to CONVENTIONAL. Across the four benchmarks, CONJUGATE COMPILATION improves throughput by 1.38× on average over CONVENTIONAL. The improvement reflects cross-model compilation informed by shared-execution measurements, enabling joint optimization of the draft and target models rather than optimizing each model independently (see §2.2 and §3). CONJUGATE COMPILATION WITH INTERLEAVED KERNELS further raises throughput to 1.47× that of CONVENTIONAL on average, achieving the highest throughput in every benchmark and reaching 1.59× on Llama-3.1-8B/Llama-3.2-1B. The additional 7% average gain over CONJUGATE COMPILATION is due to co-optimizing selected draft and target operations within a common interleaved kernel, beyond co-scheduling separate kernels (§2.2).

GPU utilization. To understand the throughput improvements, we plot GPU utilization in Figure 4. Across the four benchmarks, CONJUGATE COMPILATION WITH INTERLEAVED KERNELS increases utilization by 50% on average over CONVENTIONAL. Consider Llama-3.1-8B/Llama-3.2-1B, where utilization increases from 11.6% to 18.7%. Co-located shared execution allows ready draft operations to use GPU resources left available during target execution, rather than requiring the two models to execute sequentially. Interleaved kernels further allow the compiler to schedule selected work from both models within a single launch, so work from one operation can fill execution capacity left unused by the other (§2.2). The Llama pair's 61% utilization increase is consistent with more effective use of the GPU during joint execution and supports the 59% increase in throughput.

Roofline analysis. To examine how interleaving improves kernel performance, Figure 5 compares the down projections of Llama-3.1-8B/Llama-3.2-1B under CONVENTIONAL and CONJUGATE COMPILATION WITH INTERLEAVED KERNELS on B300. The two CONVENTIONAL kernels achieve a combined throughput of 17.9 TFLOP/s when executed sequentially, whereas the interleaved kernel achieves 46.5 TFLOP/s, a 2.6× improvement. The interleaved kernel operates closer

to the memory-bandwidth roof, indicating better effective bandwidth use when the two projections are compiled and executed together.

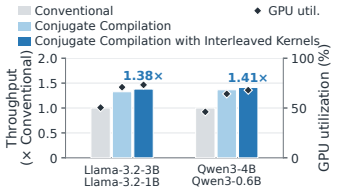

Figure 6: Sensitivity to GPU platform.

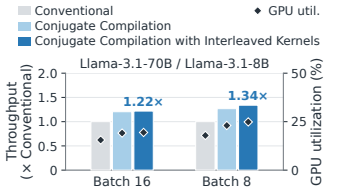

Figure 7: Sensitivity to batch size.

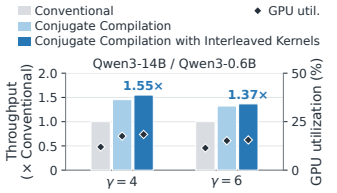

Figure 8: Sensitivity to draft length.

Sensitivity to GPU platform. Figure 6 evaluates Llama-3.2-3B/Llama-3.2-1B and Qwen3-4B/Qwen3-0.6B on NVIDIA RTX 5090 GPU. CONJUGATE COMPILATION improves throughput by $1.33\times$ and $1.37\times$ over CONVENTIONAL, respectively, while CONJUGATE COMPILATION WITH INTERLEAVED KERNELS raises the speedups to $1.38\times$ and $1.41\times$, averaging $1.40\times$ across the two pairs. Unlike the B300 workloads, the RTX 5090 workloads already achieve substantially higher utilization under CONVENTIONAL (50.4% for the Llama pair and 46.2% for the Qwen pair). Nevertheless, cross-model compilation informed by shared-execution measurements remains beneficial at higher baseline utilization, with CONJUGATE COMPILATION WITH INTERLEAVED KERNELS raising utilization to 73.2% and 67.9%, respectively. Co-location and execution-informed compilation provide most of the throughput improvement, while interleaving adds a further 4.1% and 3.3%, extending optimization from separate kernels to common cross-model kernels.

Sensitivity to batch size. Figure 7 evaluates Llama-3.1-70B/Llama-3.1-8B at batch sizes 8 and 16 on B300 GPU. At batch size 8, CONJUGATE COMPILATION and CONJUGATE COMPILATION WITH INTERLEAVED KERNELS improve throughput by $1.27\times$ and $1.34\times$ over CONVENTIONAL, respectively, while at batch size 16, the speedups are $1.20\times$ and $1.22\times$. The larger relative gains at batch size 8 are consistent with each model supplying less work per invocation, leaving more opportunity for co-location and interleaving to improve GPU resource use by combining work across models. At batch size 16, more work is available within each model, allowing batching itself to contribute more to throughput while joint compilation provides a further $1.22\times$ improvement over CONVENTIONAL. Batching and joint compilation therefore offer complementary benefits.

Sensitivity to draft length. Figure 8 evaluates Qwen3-14B/Qwen3-0.6B at batch size 8 with draft lengths $\gamma = 4$ and $\gamma = 6$. At $\gamma = 4$, CONJUGATE COMPILATION and CONJUGATE COMPILATION WITH INTERLEAVED KERNELS achieve $1.45\times$ and $1.55\times$ speedups over CONVENTIONAL, respectively, compared with $1.32\times$ and $1.37\times$ at $\gamma = 6$. Longer drafts add sequential draft steps while the target verifies more tokens in one pass, changing the balance of work available for overlap. The larger gains at $\gamma = 4$ are consistent with a more favorable balance between drafting and verification. At the measured acceptance rate of $\alpha = 0.73$, CONJUGATE COMPILATION WITH INTERLEAVED KERNELS generates 852 tokens/sec at $\gamma = 4$ versus 713 at $\gamma = 6$, favoring $\gamma = 4$ for the workload.

5 RELATED WORK

Parallel speculative decoding systems coordinate drafting and verification through runtime scheduling and placement (Liu et al., 2025; Chen et al., 2026). GPU-sharing and context-aware compilation approaches schedule concurrent computations or optimize kernels for resource constraints (Strati et al., 2024; Liu et al., 2022). LLM-guided optimization typically targets individual kernels or programs (Ouyang et al., 2025; Dong et al., 2026). Conjugate Compilation searches joint draft–target compilation plans using an LLM agent and MCTS, with measurements of the complete co-located shared execution guiding subsequent proposals. Additional related work is in Appendix B.

6 CONCLUSION

We introduced Conjugate Compilation, a framework for jointly optimizing co-located models using the measured performance of their shared execution to guide compilation. Our approach combines a self-improving LLM–MCTS search over cross-model compilation plans with transformations that bring work from multiple models into common kernels, spanning GEMM and non-GEMM operations. Across speculative-decoding workloads, our results show that cross-model Conjugate Compilation can recover GPU capacity left stranded when models are optimized independently.

AI USE STATEMENT

In this work, we used generative AI tools for polishing writing, such as refining the language and improving readability. We have not used generative AI tools for retrieval and discovery, for research ideation or execution, for generating synthetic dataset, for drafting sections of the paper, or for proving mathematical claims. We take responsibility for the final content of this work, including text polished with the aid of generative AI.

REFERENCES

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Aodong Chen, Fei Xu, Li Han, Yuan Dong, Li Chen, Zhi Zhou, and Fangming Liu. Opara: Exploiting operator parallelism for expediting DNN inference on GPUs. *IEEE Transactions on Computers*, 74(01):325–333, January 2025.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

Wenyan Chen, Chengzhi Lu, Yanying Lin, and Dmitrii Ustiugov. FASER: Fine-grained phase management for speculative decoding in dynamic LLM serving. *arXiv*, 2026.

Patrick H. Coppock, Brian Zhang, Eliot H. Solomon, Vasilis Kypriotis, Leon Yang, Bikash Sharma, Dan Schatzberg, Todd C. Mowry, and Dimitrios Skarlatos. LithOS: An operating system for efficient machine learning on GPUs. In *SOSP*, 2025.

Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, and Hugh Leather. Large language models for compiler optimization. *arXiv*, 2023.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. LLM Compiler: Foundation language models for compiler optimization. In *CC*, 2025.

Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. IOS: Inter-operator scheduler for CNN acceleration. In *MLSys*, 2021.

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. In *ICLR*, 2026.

Paul Elvinger, Foteini Strati, Natalie Enright Jerger, and Ana Klimovic. Understanding GPU resource interference one level deeper. In *SoCC*, 2025.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

International Energy Agency. Energy and AI. World Energy Outlook Special Report, https://www.iea.org/reports/energy-and-ai, 2025.

Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel GPU task scheduling for deep learning. In *NeurIPS*, 2020.

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 *SOSP*, 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *ICML*, 2023.

Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

Tianyu Liu, Yun Li, Qitan Lv, Kai Liu, Jianchen Zhu, Winston Hu, and Xiao Sun. PEARL: Parallel speculative decoding with adaptive draft length. In *ICLR*, 2025.

Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *ASPLOS*, 2022.

Jiamin Lu, Jingwei Sun, Yunlong Xu, Peng Sun, and Guangzhong Sun. ConCo: Optimizing compilation of concurrent tensor programs on shared GPU. In *ICS*, 2025.

Lixian Ma, Haoruo Chen, En Shao, Leping Wang, Quan Chen, and Guangming Tan. ElasticRoom: Multi-tenant DNN inference engine via co-design with resource-constrained compilation and strong priority scheduling. In *HPDC*, 2024.

Meta. Introducing Llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/, 2024a.

Meta. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models. https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/, 2024b.

NVIDIA. CUDA programming guide: Asynchronous execution. https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html, a. CUDA Programming Guide v13.4.2, accessed September 2026.

NVIDIA. CUPTI: CUDA profiling tools interface. https://docs.nvidia.com/cupti/12.8/index.html, b. CUPTI 12.8 documentation (v2025.1.0), accessed September 2026.

NVIDIA. NVIDIA Nsight Compute. https://docs.nvidia.com/nsight-compute/, c. Version 2025.1, shipped with CUDA Toolkit 12.8; accessed September 2026.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. https://openai.com/index/gpt-5-6/, 2026.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: can LLMs write efficient GPU kernels? In *ICML*, 2025.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 2019.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *ISCA*, 2024.

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

Haifeng Qian, Sujan Kumar Gonugondla, Sungsoo Ha, Mingyue Shang, Sanjay Krishna Gouda, Ramesh Nallapati, Sudipta Sengupta, Xiaofei Ma, and Anoop Deoras. BASS: Batched attention-optimized speculative sampling. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *ACL Findings*, 2024.

Qwen Team. Qwen3 technical report. *arXiv*, 2025.

Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained GPU sharing for ML applications. In *EuroSys*, 2024.

Systalyze. Utilyze. https://github.com/systalyze/utilyze, 2026.

Annabelle Sujun Tang, Christopher Priebe, Rohan Mahapatra, Lianhui Qin, and Hadi Esmaeilzadeh. REASONING COMPILER: LLM-guided optimizations for efficient model serving. In *NeurIPS*, 2025.

Zhenwei Tang, Arun Verma, Zijian Zhou, Zhaoxuan Wu, Alok Prakash, Daniela Rus, and Bryan Kian Hsiang Low. MineDraft: A framework for batch parallel speculative decoding. In *ICML*, 2026.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *NeurIPS*, 2019.

Shulai Zhang, Quan Chen, Weihao Cui, Han Zhao, Chunyu Xue, Zhen Zheng, Wei Lin, and Minyi Guo. Improving GPU sharing performance through adaptive bubbleless spatial-temporal sharing. In *EuroSys*, 2025.

Ziyi Zhang, Ziheng Jiang, Chengquan Jiang, Menghan Yu, Size Zheng, Haibin Lin, Xin Liu, and Henry Hoffmann. SwiftSpec: Disaggregated speculative decoding and fused kernels for low-latency LLM inference. In *ASPLOS*, 2026.

A Additional Search Details

A.1 Expanding the Search Space

A fixed search space and tuning budget can cause exploration to stall before profitable co-optimizations are found. When measured progress plateaus, it first retunes near-miss kernels with progressively larger budgets, then widens the candidate groups and legal placements, refines the co-schedule down to individual projection stages so joint nodes can contain more precisely selected work, and finally exposes additional kernel-template choices. The agent is informed whenever the available search space expands.

We impose an effort floor before concluding that a kernel family offers no profitable co-optimization. Each kernel family from both models is explored through multiple groupings, and every cross-model GEMM kernel appearing in a final candidate receives at least the combined tuning budget of the kernels it replaces. This prevents differences in compiler effort from determining the comparison.

A.2 Final Plan Acceptance

Search-time measurements guide exploration, but the reported plan is subjected to a stricter final evaluation. After search, we re-time the strongest candidates and the best agent-proposed plans in paired ABBA blocks against the co-scheduling anchor, while re-measuring the conventional and plain two-stream baselines in the same measurement window. A plan is accepted only if its outputs pass the numerical checks in §3.3, the paired 95% confidence interval of its gain excludes zero, and its improvement exceeds a noise threshold. Finally, we remove every range of joint nodes whose leave-one-out gain does not exceed the noise threshold, unless the full plan is itself significantly faster than the reduced plan. This final pruning requires each retained range to contribute measurable value, either independently or through its interaction with the rest of the plan.

B Additional Related Work

B.1 Parallel Speculative Decoding

Prior work on parallel speculative decoding primarily treats draft–target parallelism as a runtime scheduling, placement, or resource-allocation problem rather than a compile-time optimization problem. Most of these systems place the draft and target models on separate GPUs, which improve performance but leave both devices underutilized: PEARL overlaps drafting and verification through adaptive parallel speculative decoding (Liu et al., 2025), MineDraft exploits batch-level parallelism across speculative-decoding rounds (Tang et al., 2026), and SwiftSpec disaggregates draft and target execution across GPU resources (Zhang et al., 2026). FASER instead is a profile-driven runtime system for parallel draft–target execution on a single GPU, using offline profiling to guide how the two models share the device (Chen et al., 2026). These approaches operate after compilation: they schedule, place, or partition already-compiled model computations rather than changing how the draft and target are compiled together. Conjugate Compilation instead treats co-located execution as a compilation problem, searching cross-model transformations and evaluating each candidate using the complete score produced by the co-located shared execution.

B.2 GPU Sharing and Context-Aware Compilation

GPU sharing and inter-operator scheduling systems improve utilization by concurrently executing independent operators or models on the same device (Kwon et al., 2020; Ding et al., 2021; Chen et al., 2025; Strati et al., 2024; Zhang et al., 2025; Coppock et al., 2025). These systems do not target speculative decoding; rather, they address general multi-tenant or inter-operator GPU sharing, primarily deciding which already-compiled computations should execute concurrently, while context-aware compilation approaches optimize individual kernels for a predefined resource budget or interference condition (Liu et al., 2022; Ma et al., 2024; Lu et al., 2025). More fundamentally, these approaches rely on profiling, analytical models, or conventional optimization policies to make their scheduling and compilation decisions; they do not use an execution-driven agentic compiler that learns how to improve co-located execution from the outcomes of its own optimization attempts. Conjugate Compilation instead targets speculative decoding directly and couples an LLM agent with Monte Carlo tree search to iteratively propose cross-model compilation plans, compile and execute them, and use the complete score from the co-located shared execution together with kernel-level verdicts and compiler feedback to guide subsequent proposals. This self-improving loop makes the measured behavior of the co-located execution the feedback signal for compilation, allowing the search to adapt its transformations as it discovers which optimizations help—or hurt—the models when they execute together.

B.3 LLM-Guided Compiler Optimization

Recent work uses LLMs and agents to optimize individual kernels or programs (Ouyang et al., 2025; Dong et al., 2026; Cummins et al., 2025; 2023; Tang et al., 2025). Conjugate Compilation extends agentic optimization across multiple models, using an LLM with MCTS to search joint draft–target compilation plans guided by measurements from their co-located shared execution.