KernelEvolve: Scaling Agentic Kernel Coding for Heterogeneous AI Accelerators at Meta

Gang Liao*[1] Hongsen Qin[1] Ying Wang[1] Alicia Golden[1,2] Michael Kuchnik[1] Yavuz Yetim[1]
Ruichao Xiao[1] Jia Jiunn Ang[1] Chunli Fu[1] Yihan He[1] Samuel Hsia[1] Zewei Jiang[1] Roman Levenstein[1]
Dianshi Li[1] Liyuan Li[1] Ajit Mathews[1] Varna Puvvada[1] Feng Shi[1] Nathan Yan[1] Xiayu Yu[1]
Uladzimir Pashkevich[1] Matt Steiner[1] Carole-Jean Wu*[1] Gaoxiang Liu*[1]
[1]*Meta* [2]*Harvard University*
gangliao@meta.com, carolejeanwu@meta.com, gaoxiang@meta.com

***Abstract*—Making deep learning recommendation model (DLRM) inference fast and efficient is important. However, this presents three key system challenges – model architecture diversity, kernel primitive diversity, and hardware generation and architecture heterogeneity. The combination of the three diversity dimensions leads to a complex optimization space for fast, efficient inference at-scale.**

This paper presents KernelEvolve– an agentic kernel coding framework – to tackle heterogeneity at-scale for DLRM training and inference. KernelEvolve is designed to take kernel specifications as input and automate the process of kernel generation and optimization for recommendation model across heterogeneous hardware architectures. KernelEvolve does so by operating at multiple programming abstractions, from Triton and CuTe DSL to low-level hardware agnostic languages, spanning the full hardware-software optimization stack. The kernel optimization process is described as graph-based search with selection policy, universal operator, fitness function, and termination rule, dynamically adapts to runtime execution context through retrieval-augmented prompt synthesis.

We designed, implemented, and deployed KernelEvolve to optimize a wide variety of production recommendation models across generations of NVIDIA and AMD GPUs, as well as *Meta*'s AI accelerators. We validate KernelEvolve on the publicly-available KernelBench suite, achieving 100% pass rate on all 250 problems across three difficulty levels, and 160 PyTorch ATen operators across three heterogeneous hardware platforms, demonstrating 100% correctness. KernelEvolve reduces development time from weeks to hours and achieves substantial performance improvements over PyTorch baselines across diverse production use cases and for heterogeneous AI systems at-scale. Beyond performance efficiency improvements, KernelEvolve significantly mitigates the programmability barrier for new AI hardware by enabling automated kernel generation for in-house developed AI hardware.

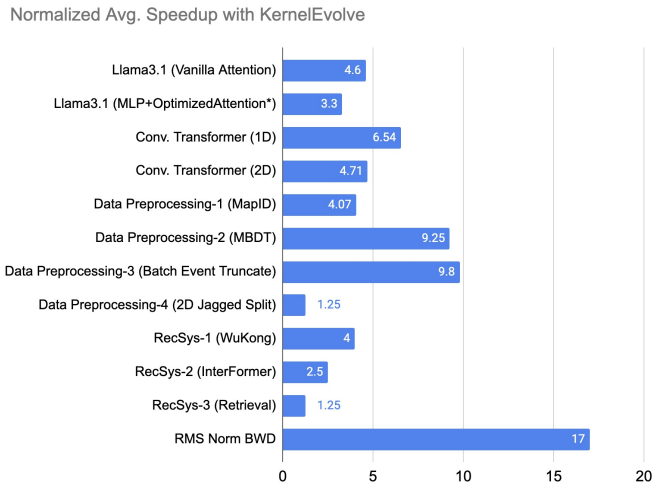

Fig. 1: KernelEvolve achieves 1.25–17x speedups across Meta LLMs and production use cases, spanning convolutional Transformers, data preprocessing operators, and recommendation systems, over heterogeneous AI hardware.

I. Introduction

Deep learning recommender systems form the backbone for digital entertainment, advertisement in e-commerce platforms, and news recommendation. Modern recommendation model inference can demand significant computing resources to meet stringent sub-second latency requirement [29]. In 2019, about 80% of Meta's overall deep learning inference stemmed from recommendation model tasks [17]. As recommendation model architectures evolve from simple Multi-Layer Perception (MLP) with embedding tables [33] to Transformer model architectures [58], the complexity and system resource demands increases as well. Additionally, a large and diverse collection of models are deployed and ensembled in a multi-stage ranking system to address varying product needs.

However, deploying DLRMs efficiently is challenging, as a wide range of AI hardware accelerators, including Meta Training and Inference Accelerators (MTIAs) [9], [12], NVIDIA GPUs, AMD GPUs, are used to improve performance efficiency and cost. In contrast to LLM inference, where input shapes are relatively homogeneous and model architectures are consistent, DLRMs introduce unique diversity characteristics. This diversity presents key system challenges and design optimization opportunities at scale:

- **Model diversity over ranking stages:** Modern recommendation models are composed of multi-stage model architectures. Early-stage models *retrieves* O(10K) can-

*Corresponding authors.

Ranking Stages	Description	Production Models	Total Models	Model Complexity
Early stage ranking	Rank 10K candidates	$\geq$ 150 models	$\geq$ 500 models	0.01-0.1 GFLOPS/request
Late stage ranking†	Rank 100 candidates	$\geq$ 200 models	$\geq$ 1000 models	0.2-10 GFLOPS/request

TABLE I: Recommendation model distribution at Meta. †Transformer-based sequence models require significantly higher computational complexity at ∼80 GFLOPS/request, representing a 10-100× increase over traditional dense ranking models.

didates, using moderate-complexity neural networks to balance computational cost and filtering accuracy. These models employ lightweight scoring functions, and techniques such as nearest neighbor search and efficient embedding operations to prune the search space. In contrast, late-stage models *ranks* 100× less candidates, employing heavyweight neural networks such as Transformer-based architectures with 10-100× higher model complexity of GFLOPS/request.

- **Kernel diversity beyond GEMM:** While dense matrix multiplication (GEMM) operations benefit from mature, highly-optimized libraries (cuBLAS, FBGEMM, DeepGEMM), production recommendation model workloads necessitate broader kernel coverage. Recommendation ranking models can execute over 200 distinct data preprocessing operators as integral components of model inference pipelines, where raw features undergo transformation before feeding subsequent neural network layers.
- **Hardware diversity across hardware vendors and generations:** Our production recommendation model deployment spans across NVIDIA GPUs, AMD GPUs, and Meta's custom MTIA chips. Each platform consists of distinct architectural properties, including varying peak FLOPS capabilities, memory hierarchies, and scale-out interconnect properties. Furthermore, the software stack presents a significant challenge – programming model compatibility presents a major hurdle to adopting these heterogeneous platforms.

The multi-dimensional diversity challenge is expected to exacerbate as new model architectures, serving systems, and hardware solutions, become available.

To respond to the ever-increasing diversity challenge for recommendation model inference at-scale, we present KernelEvolve— an agentic kernel coding framework for hyperscale AI accelerators at Meta. KernelEvolve tackles the programmability challenge faced by ASIC hardware using large language models that specialize in coding tasks. Building upon the Triton programming ecosystem, KernelEvolve automates the generation and optimization of compute kernels of recommendation model inference across heterogeneous hardware architectures by taking a triton kernel specification as input and producing optimized kernels as the code executable for the respective hardware backend.

In addition, we enable low level hardware-specific performance tuning. KernelEvolve leverages agentic AI capabilities to establish an end-to-end kernel generation service — starting with automated kernel synthesis, followed by iterative optimization with performance feedback, cross-platform compilation (NVIDIA, AMD, MTIA), and finally automated correctness verification. To verify correctness, KernelEvolve employs a multi-level profiling infrastructure spanning system-level latency monitoring, kernel-level performance characterization, and intra-kernel operator profiling. This approach fundamentally transforms traditional kernel development from a manual, expertise-dependent process to an automated, scalable service maintaining production-grade performance standards while adapting to the fast evolving model architectures and hardware diversity.

This paper makes the following key contributions:

- We present KernelEvolve, the first production-grade AI-powered kernel optimization system deployed at industrial scale for recommendation model inference workloads. Unlike prior research prototypes, KernelEvolve operates continuously in Meta's production serving infrastructure, generating Triton kernels for hundreds of models serving trillions of inference daily. We demonstrate how agent-based approaches achieve production-grade reliability while exploring optimization strategies infeasible through manual development.
- We demonstrate KernelEvolve's autonomous kernel optimization achieving competitive performance with expert manual implementations while reducing development time from weeks to hours. Through diverse production use cases spanning ads training and serving workloads, KernelEvolve-generated kernels achieve 1.2 to 17 times speedup over the PyTorch baselines, demonstrating that automated synthesis can exceed state-of-the-art compiler-generated code for domain-specific operators. We analyze optimization trajectories across diverse operator types and hardware platforms, identifying patterns in successful kernel generation strategies that inform future automated optimization systems.
- We share insights from operating KernelEvolve in production environments, including failure mode analysis, debugging strategies for incorrect kernel generation, performance validation methodologies ensuring production safety, and organizational integration patterns for adopting automated kernel generation workflows.

Additional result analysis detail is available in the extended version of this paper [24].

II. Recommendation Inference Production Environments at Meta

To better understand the motivation for KernelEvolve, we first analyze insights from our internal production recommendation workloads, highlighting the operator and system stack diversity across hardware platforms in the production fleets.

Serving Paradigm 1: Overall Latency (ms)				
Compute Tier	**p50**	**p75**	**p90**	**p99**
Client → MTIA Tier	39	44	46	61
Serving Paradigm 2: Overall Latency (ms)				
Compute Tier	**p50**	**p75**	**p90**	**p99**
α	58	65	73	97
β	42	48	51	57
γ	4	7	10	16
δ	$\delta = \alpha - \beta - \gamma$ = 10 to 20 ms			

TABLE II: Latency comparison of monolithic vs. disaggregated serving paradigms for a production MTIA model. Paradigm 1 executes preprocessing on client-side, achieving 61ms p99 tail latency with direct client→remote MTIA communication. Paradigm 2 introduces a dedicated CPU tier for preprocessing scalability, incurring additional network hops that increase the p99 tail latency to 97ms. The extra network latency ($\delta = \alpha - \beta - \gamma$) is pure overhead from disaggregated serving when preprocessing operators lack native accelerator implementations. Here, α represents the latency of Client → CPU Tier, β CPU → MTIA Tier, and γ data preprocessing.

A. Meta's Recommendation Models

Traditional recommendation models, such as DLRM [33], combine sparse feature embeddings with dense feature transformations via multi-layer perceptrons to rank and recommend candidates. Recent production models explore sequence generation by introducing Transformer components. For example, compared to DLRM [33], HSTU [58] processes user history through jagged attention mechanisms, resulting in 10-100× per-request complexity increases. It also comes with larger embedding tables, resulting in increased memory capacity and bandwidth requirement [58] .

Other recommendation models in production feature a multi-stage ranking pipeline [16], [17], where each stage is characterized by unique computational characteristics, as outlined in Table I. First, retrieval stages process millions of candidates, leveraging low-complexity models and large batch sizes to narrow down candidates to O(10K)-O(100K) items. Then, early-stage ranking utilizes moderate-complexity models to further refine the space down to O(100) items. Finally, late-stage ranking models make use of heavyweight models, such as, DHEN [61], Wukong [60].

1) Diverse Operator Types and Shapes: The effectiveness of recommendation models in production hinges in part on data preprocessing, where raw features are transformed into model-ready inputs, executing transformations on batched data [23], [62], [63]. While dense matrix multiplication (GEMM) operations benefit from mature, highly-optimized libraries (cuBLAS, FBGEMM, DeepGEMM) [10], [39], production recommendation models exhibit fundamentally different computational characteristics that necessitate a significantly higher kernel coverage. Production recommendation models come with more than two hundred distinct data preprocessing operators, where raw features undergo transformation before being fed into subsequent neural network layers.

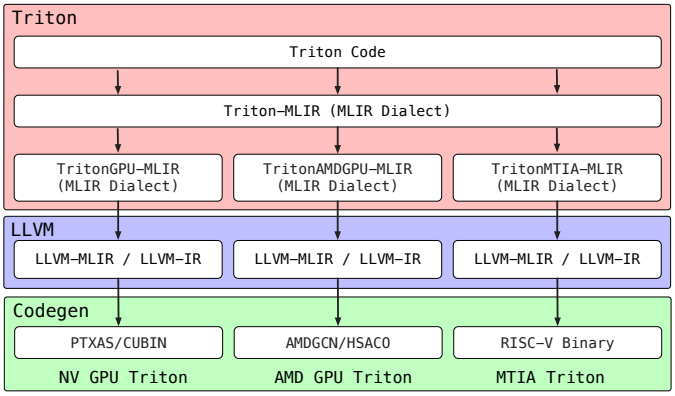

Fig. 2: Triton Multi-Target Compilation Overview.

The data preprocessing pipelines execute three types of operators: (1) dense normalization via statistical transformations and one-hot encodings, (2) sparse processing via top-K selection and hashing, and (3) feature derivation via complex operator compositions (e.g., bucketizing continuous features into categorical bins or n-gram hashing). Feature derivation in particular comprises hundreds of operator branches, creating substantial kernel diversity when compared to models.

While individually exhibiting low arithmetic intensity compared to GEMM operations, kernel availability for these preprocessing operators fundamentally determines *how models are deployed and the resulting system performance*. The lack of optimized preprocessing kernels on AI accelerators creates a binary constraint: without native accelerator implementations, models become ineligible for unified accelerator deployment. As shown in Table II, executing preprocessing on accelerator host CPUs proves infeasible: (1) host CPUs in accelerator servers are underprovisioned compared to dedicated CPU tiers, lacking capacity for preprocessing-intensive workloads; (2) heterogeneous preprocessing demands—varying compute and memory bandwidth requirements—contend with accelerator I/O and system management for limited host resources; (3) allocating additional host CPU capacity negates accelerator consolidation's economic and power efficiency benefits. It is this very diversity and computational significance of preprocessing operators that motivates comprehensive kernel coverage as a first-order architectural requirement.

B. Heterogeneous Deployment Environment

Meta's deployment settings feature a heterogeneous mix of hardware platforms, including NVIDIA and AMD GPUs, and Meta's in-house MTIA accelerators. This not only presents challenges in hardware-specific model inference optimization, but also creates a diversity in the system stack required to utilize these platforms. To address this, Meta is increasingly turning to Triton [46], as it can be leveraged to provide cross-platform support for Meta's heterogeneous deployment scenarios. Figure 2 provides the Triton system overview.

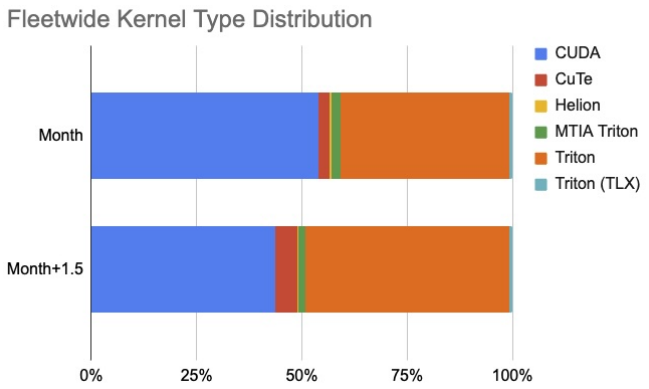

Fig. 3: Triton is overtaking CUDA as the dominant kernel programming model at Meta. This shift toward higher-level DSLs, while maintaining legacy CUDA and introducing new abstraction, creates programming model fragmentation, motivating KernelEvolve's automated synthesis approach.

Figure 3 shows that, over time, Triton is overtaking CUDA as the dominant kernel programming model at Meta, growing to *thousands of kernels* in production already. Other domain specific languages (DSLs) still persist in the system stack, such as CuTe [35] and TLX [31]. Deployment platforms must still support CUDA. This shift toward higher-level DSLs—while maintaining legacy CUDA and supporting new abstraction—creates programming model fragmentation, thereby motivating a new automated kernel synthesis approach.

III. KernelEvolve

We introduce KernelEvolve—an agentic framework that automates the generation and optimization of high-performance compute kernels for production-scale recommendation model serving across heterogeneous hardware architectures. KernelEvolve delivers an autonomous end-to-end kernel generation service that performs synthesis, compilation, numeric verification, profiling, and benchmarking.

A. System Overview.

Figure 4 depicts the KernelEvolve framework. KernelEvolve consumes a natural-language prompt as input, which specifies the target kernel and hardware platform. KernelEvolve then passes the prompt to an prompt synthesizer, which augments the prompt based on contextual information. The framework then leverages this augmented prompt to generate a kernel using one of several LLM models (Llama [11], CWM [5], GPT [14], Claude [8], etc.). KernelEvolve then deploys a series of sub-agents to evaluate accuracy (via TritonBench [32]), and system performance (through hardware-specific profilers). Finally, the system updates its contextual memory with relevant generation information which can then be used for dynamic prompt generation during the next iteration. KernelEvolve repeats this process in an iterative manner as it navigates through a graph of the solution search space, ultimately producing an optimized kernel as output. We describe each system component of KernelEvolve in more detail below.

B. Iterative Graph-Based Search Framework.

KernelEvolve builds on prior work in coding agent design [19], [47] and represents the coding problem as a graph-based search space $G_t = (V_t, E_t)$, where $v_i \in V_t$ is a single kernel solution in the set of all solutions observed at step t, and edge $(v_i, v_j) \in E_t$ represents a transformation from one solution to the next. As shown in Figure 4, this framework allows the agent to effectively explore the search space of possible kernels until an optimal kernel is found.

- **Selection Policy** At each iteration, a selection policy π_{sel} will choose a subset of nodes from the graph for expansion, guided by a heuristic for scoring existing solutions (nodes). Greedy selects the single highest fitness node. Monte Carlo Tree Search (MCTS) balances exploration-exploitation via upper confidence bounds weighted by visit count and evolutionary search uses fitness-proportional selection with crossover/mutation operators.
- **Universal Operator** $\mathcal{O} : \mathcal{S} \times \mathcal{C} \to \mathcal{S}$ is a single transformation function that generates new kernel candidates from existing implementations, where $\mathcal{C}$ represents the contextual information (profiling results, error messages, hardware constraints, historical optimizations) that guides the transformation. Unlike traditional approaches that employ multiple specialized operators with fixed prompting strategies [19], [47], KernelEvolve limits operators to 1) initial *drafting* of starting solutions and 2) the *universal improvement operator*, which dynamically adapts its behavior (debugging or improving) based on runtime context through retrieval-augmented prompt synthesis.
- **Fitness Function** $\mathcal{F} : \mathcal{S} \to \mathbb{R}_{\geq 0}$ estimates the quality of a kernel implementation node $v \in V_t$ through the speedup achieved by the generated Triton kernel relative to the PyTorch compiled reference code [3]. A generated Triton kernel is assigned a fitness equivalent to the speedup-multiple over the baseline, or 0 if the kernel fails correctness check. This fitness measure directly captures the performance optimization objective while ensuring correctness as a hard constraint.
- **Termination Rule** τ halts search when computational budgets (wall-clock time or maximum number of artifacts) are exhausted, progress stalls, or fitness thresholds are achieved. In practice, the search graph is lightweight: typically $<$100 nodes (up to 200–300 for complex kernels), with depth 10–20 levels. Each node stores a complete kernel artifact (source code + profiling results). Node selection is $O(n)$ over the frontier. The computational bottleneck is kernel compilation and profiling ($\sim$minutes per iteration), not graph traversal ($\sim$microseconds).

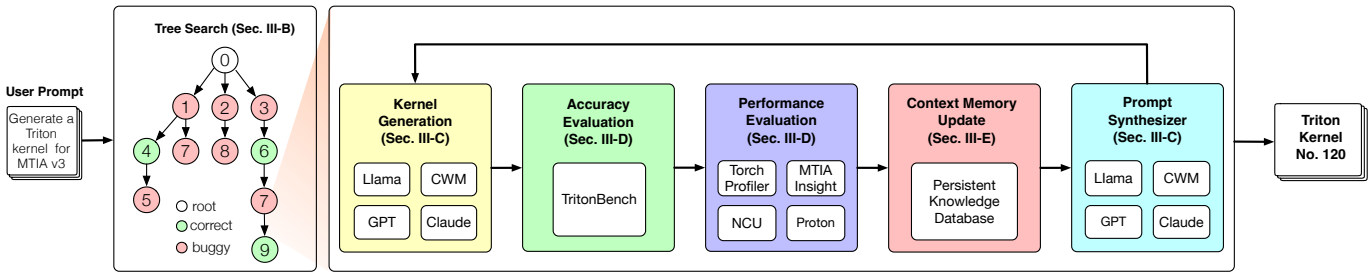

Fig. 4: **KernelEvolve Framework Overview.** KernelEvolve takes in a natural-language prompt and synthesizes an augmented prompt based on user input and contextual information. The framework then generates a kernel using one of several LLM models (Llama, CWM, GPT, Claude, etc.). KernelEvolve then deploys a series of sub-agents to evaluate accuracy (via TritonBench), and system performance (through hardware-specific profilers). Finally, the system updates its contextual memory with relevant generation information which can then be used for dynamic prompt generation during the next iteration. KernelEvolve repeats this process as it navigates through a graph of the search space, ultimately producing an optimized kernel as output.

C. LLM Synthesizer and Kernel Generation

As Figure 4 shows, each iteration over the search graph begins with an LLM synthesizer that generates dynamic prompts for kernel generation – and augments these prompts with context from specialized sub-agents. The sub-agents combine information from several sources including (i) previous kernel implementation and execution history, (ii) LLM-generated analysis reports from prior iterations, (iii) retrieved knowledge bases, and (iv) hardware-specific constraints. More details on contextual memory updates can be found in Section III-F.

The augmented prompts are then processed by either internal backends (CWM [5], Llama [11]) or external models (Claude 4.5 [8], GPT 5 [14]) in order to generate a kernel candidate, utilizing roughly 64K-1M tokens depending on the LLM backend. Each generated artifact comprises three components: A PyTorch baseline, an optimized Triton kernel, and input data generation that is structured to support systematic evaluation and downstream compilation integration.

D. Accuracy and System Performance Evaluation

KernelEvolve employs an agentic evaluation framework, utilizing sub-agents to evaluate candidates across multiple dimensions and harness the resulting signals to inform subsequent kernel generation.

First, kernel candidates are evaluated for accuracy via TritonBench [32], which compares generated kernels against PyTorch reference implementations. Speedup is calculated by quantifying the generated kernel latency over baseline implementations. Next, system-level and efficiency metrics are evaluated through the use of several profiling frameworks. Torch Profiler [30] captures CPU/GPU time, kernel launch overhead, and function execution durations to identify host-device bottlenecks, while Nvidia NCU [36], Triton Proton [64], and MTIA Insight offer more detailed explanations of platform-specific insights. KernelEvolve's evaluation framework spans the computing stack, and utilizes Meta's Multi-Pass profiler (MPP) in order to unify instrumentation, compiler transforms, profiling, and trace synthesis as composable job tasks. Section III-E describes how KernelEvolve automatically generates evaluation harnesses for these tools.

To ensure safe execution, KernelEvolve employs several safeguards: (1) sandboxed execution in isolated containers prevents generated code from affecting systems; (2) correctness validation against reference outputs enforces strict numerical tolerances; (3) mandatory human review gates all kernels before deployment; and (4) search trace logging enables full auditing of the optimization trajectory.

E. Evaluation Harness Generation

Kernel candidates produced by the search framework conform to a standardized interface (`PytorchModel`, `TritonModel`, `get_inputs()`), but cannot be profiled directly—each tool in Section III-D requires its own instrumentation boilerplate. KernelEvolve addresses this with a deterministic code generator that transforms kernel artifacts into platform-specific evaluation scripts, cleanly separating LLM-generated kernel logic from evaluation instrumentation.

Harness Synthesis. Given a kernel artifact, the code generator emits one executable Python script per profiling tool. For TritonBench, it wraps both model variants inside the `BenchmarkOperator` framework with correctness validation enabled. For Torch Profiler, it inserts `torch.profiler.profile()` contexts around kernel invocations. For NCU and Proton, it synthesizes tool-specific instrumentation via Triton's Multi-Pass Profiler (MPP). For MTIA Insight, it configures instrumentation for PE utilization, fixed-function accelerator metrics (DPE/SFU/MLU), cache behavior, and memory bandwidth. Every generated script imports from the standardized artifact, iterates over test cases from `get_inputs()`, and emits structured results consumable by the context memory sub-agent (Section III-F).

Pre-deployed Interpreter Environments. Generated evaluation scripts execute against pre-deployed interpreter environments that bundle complete toolchains—Triton compilers, profiling frameworks, and runtime libraries—via continuous

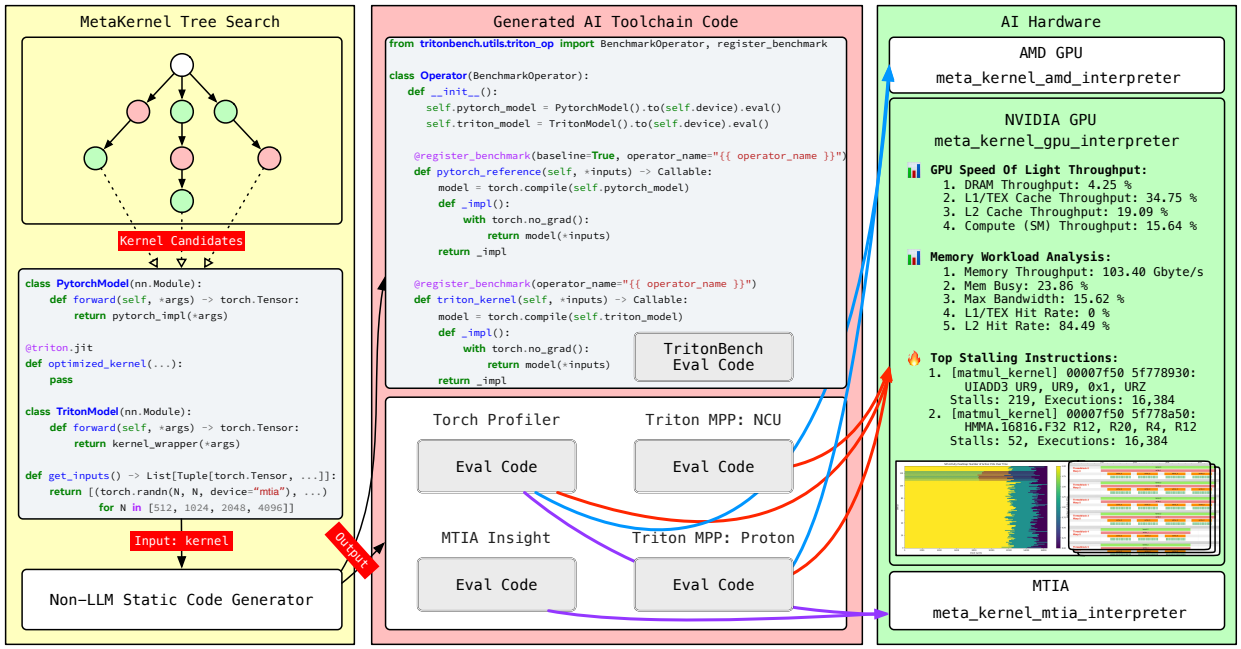

Fig. 5: **End-to-end Evaluation Pipeline.** Tree search generates kernel candidates with standardized dual implementations (PyTorch baseline, Triton optimized), executed on hardware interpreters (GPU, AMD, MTIA) collecting platform-specific profiling metrics via TritonBench, NCU, MPP, and MTIA Insight. Profiling feedback guides subsequent search iterations.

deployment. This eliminates per-evaluation dependency resolution and compilation overhead. Because compilation occurs once during interpreter deployment rather than per kernel candidate, evaluation latency drops from $\geq$10 minutes to seconds. The architectural separation also ensures that (1) evaluation code remains identical across kernel variants, guaranteeing reproducible profiling, and (2) profiling tool APIs update independently through interpreter redeployment without modifying kernel generation prompts.

Figure 5 illustrates this workflow: tree search produces kernel candidates, the evaluation code generator transforms these into tool-specific harnesses invoking TritonBench, profilers, and hardware-specific instrumentation, and hardware interpreters execute the generated evaluation code collecting platform-specific metrics that feed back to guide subsequent search iterations.

F. Contextual Memory Updates

KernelEvolve then leverages the collected evaluation metrics to augment the next iteration of kernel generation. In particular, KernelEvolve is built upon a retrieval-augmented approach, where contextual information is dynamically loaded into the LLM's context at runtime — eliminating the need to maintain comprehensive historical knowledge in working memory. In order to support this context augmentation, KernelEvolve's agentic retrieval system contains two parts – (i) a context memory sub-agent, and (ii) a deep search sub-agent.

Context Memory Sub-Agent. First, the context memory sub-agent analyzes dynamic runtime artifacts (i.e., kernel implementations, profiling measurements, error diagnostics, and correctness validation results) in order to diagnose performance bottlenecks and synthesize optimization directives. The information is held in a relational storage architecture.

Deep Search Sub-Agent. Additionally, the deep search sub-agent enables a persistent knowledge base that encodes domain expertise across hardware platforms, optimization strategies, debugging patterns, and language constraints. This is implemented as a hierarchical file system, which partitions content across the categories of constraints, guidance, hardware.

Notably, the design of the sub-agents allow for MTIA knowledge injection. Unlike widely-documented GPU architectures, MTIA's proprietary architecture and programming model remain largely absent from public training corpora. Knowledge injection allows KernelEvolve to leverage hardware-specific features such as Specialized Function Units (SFU) operations, inter-PE communication, and dual-core synchronization. This approach generalizes to emerging accelerator architectures: as new hardware platforms enter production, corresponding documentation injected into the knowledge base enables immediate LLM-based kernel generation without model retraining. The underlying persistent storage layer enables continuous learning and knowledge accumulation across optimization runs, informing context for the next kernel generation step and self-improving agent capability over time.

The knowledge base provides general hardware guidance (e.g., memory hierarchy constraints, supported intrinsics), not operator-specific optimization solutions. The agent autonomously discovers which optimization technique applies to a given operator through the iterative feedback loop. For example, the vectorized counting optimization for MBDT

(Section IV-B4) was discovered entirely by iterative search, not retrieved from the knowledge base. User input is minimal—an operator specification and hardware target—with no optimization strategy specified.

G. Experimental Setup

The evaluation uses two workloads: (1) the open-source KernelBench suite [38], comprising 250 PyTorch operators, and (2) performance-critical kernels from a production recommendation model serving real traffic (Section II). All experiments target FP16, matching the precision of the deployed inference path. KernelEvolve is numerics-format agnostic—it operates on kernel specifications and profiling feedback, validating numerical correctness against configurable error tolerances rather than assuming a fixed data type. No model-quality regressions have been observed from KernelEvolve-generated kernels. Extending the evaluation to lower-precision formats (FP8/FP4, OCP MX) and structured sparsity (2:4) is future work.

The evaluation results are based on a subset of AI hardware present in Meta's datacenter fleet, including NVIDIA A100s, H100 GPUs, AMD MI300, MI350 GPUs, and MTIA v2i [9] and the next generation MTIA v3. We tailor KernelEvolve's configuration files to support the different hardware platforms, specifying the correct backend and evaluation framework. Most kernels converge within 50–100 iterations (∼20–30K tokens each), with wall-clock time of several hours-bottlenecked by compilation and profiling rather than LLM inference. Human effort is limited to 10–60 minutes for task description preparation. For context, expert kernel development for new hardware typically requires 3–4 weeks of non-parallelizable effort; KernelEvolve executes searches concurrently across operators. At serving scale (trillions of inferences/day), even a 1% efficiency gain yields infrastructure benefits much greater than the one-time search computational cost.

IV. Evaluation Results and Analysis

A. PyTorch OSS Operators

Kernel coverage — the availability of optimized implementations for standard operators — is a fundamental prerequisite for deploying models on emerging AI accelerators. Before optimizing for performance, the system must first demonstrate its ability to generate correct kernels across the operator set. This section evaluates KernelEvolve's end-to-end capability to generate, validate, and benchmark kernels.

We curate a test suite of 160 ATen operators covering basic computational patterns, such as, element-wise arithmetic (`torch.add`, `torch.div`), transcendental functions (`torch.cos`, `torch.exp`), reductions (`torch.amax`, `torch.allclose`), and activation primitives (`torch.ops.aten.elu`). While these operators are relatively simple, they represent the foundational building blocks required for PyTorch model execution and serve as an end-to-end validation of KernelEvolve's kernel generation correctness. For each operator, KernelEvolve generates Triton kernel implementations targeting three platforms: NVIDIA H100, AMD MI350, and MTIA v3. Generated kernels are validated against the PyTorch reference implementations compiled with `torch.compile`. Numerical equivalence is verified using `torch.allclose` in TritonBench with precision-appropriate tolerances.

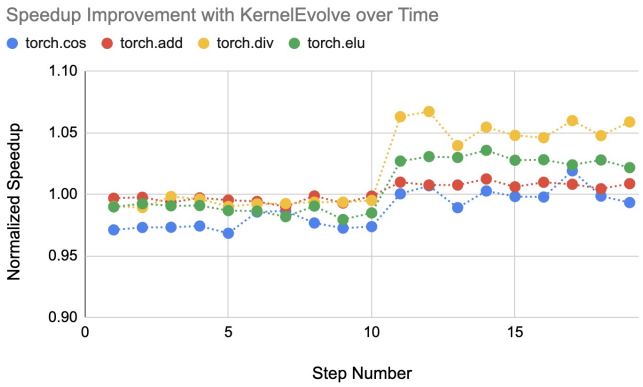

Fig. 6: Fitness score trajectories during KernelEvolve's tree search optimization for four representative ATen operators. The x-axis denotes search steps (20 in total), and the y-axis shows the normalized speedup of the generated Triton kernel over the PyTorch baseline on NVIDIA H100s. The first 10 steps correspond to the draft phase (repeated sampling without memory context), while subsequent steps represent tree expansion with execution feedback.

KernelEvolve achieves 100% correctness across all 480 operator-platform configurations (160 operators × 3 platforms). We further validate the results with KernelBench [38], achieving 100% pass rate across all three levels: Level 1 (single operators), Level 2 (fused operator patterns), and Level 3 (full model blocks). While KernelBench originally targets CUDA kernel generation, these results demonstrate that KernelEvolve reliably produces numerically correct Triton kernels across diverse system hardware and operator complexities, from individual primitives to end-to-end model components. This establishes the foundation for addressing the kernel coverage challenge on emerging hardware.

Figure 6 shows optimization trajectories for the four key PyTorch operators. The y-axis represents the normalized speedup over the PyTorch reference implementation. The search operates in two phases. In the draft phase (Steps 0-10), KernelEvolve generates candidate kernels through independent sampling without feedback. In the tree expansion phase (Steps 10–20+), each node incorporates execution feedback and profiling data, compilation status, and correctness results from its ancestors, enabling iterative refinement. The trajectories exhibit operator-dependent behavior. Three out of the four PyTorch operators — `torch.add`, `torch.elu`, and `torch.div` — shows early-stage improvement, demonstrating the benefits of iterative refinement, even if the initial kernels were suboptimal. The generated kernels can outperform compiler-generated baselines. Since the open-source PyTorch ATen op-

Precision	Tensor Shape $(B \times C_{in} \times C_{out} \times L)$	torch.conv1d (ms)	torch.conv2d (ms)	Triton (ms)	Speedup vs. conv1d	Speedup vs. conv2d
FP16	$64 \times 96 \times 96 \times 200$	0.03050	0.02019	0.01597	1.91×	1.26×
	$128 \times 96 \times 96 \times 200$	0.03840	0.02490	0.01830	2.10×	1.36×
	$256 \times 96 \times 96 \times 200$	0.05318	0.03347	0.02842	1.87×	1.18×
	$512 \times 96 \times 96 \times 200$	0.08646	0.06006	0.04982	1.74×	1.21×
	$1024 \times 96 \times 96 \times 200$	0.17226	0.11299	0.08406	2.05×	1.34×
	$2048 \times 96 \times 96 \times 200$	0.34243	0.24106	0.14864	**2.30×**	**1.62×**
	$32 \times 64 \times 64 \times 512^{\dagger}$	0.02768	0.01779	0.01264	2.19×	1.41×
	$32 \times 256 \times 256 \times 1024^{\dagger}$	0.07933	0.05485	0.06029	1.32×	0.91×
	$64 \times 768 \times 768 \times 1024^{\dagger}$	0.71549	0.55354	1.12784	0.63×	0.49×
FP32	$64 \times 96 \times 96 \times 200$	0.03501	0.02531	0.02186	1.60×	1.16×
	$128 \times 96 \times 96 \times 200$	0.04630	0.03248	0.03510	1.32×	0.93×
	$256 \times 96 \times 96 \times 200$	0.07168	0.05712	0.05789	1.24×	0.99×
	$512 \times 96 \times 96 \times 200$	0.15030	0.11517	0.11219	1.34×	1.03×
	$1024 \times 96 \times 96 \times 200$	0.32077	0.24269	0.19725	1.63×	1.23×
	$2048 \times 96 \times 96 \times 200$	0.61411	0.46384	0.35594	**1.73×**	**1.30×**
	$32 \times 64 \times 64 \times 512^{\dagger}$	0.02730	0.01978	0.01571	1.74×	1.26×
	$32 \times 256 \times 256 \times 1024^{\dagger}$	0.13469	0.10326	0.13501	1.00×	0.77×
	$64 \times 768 \times 768 \times 1024^{\dagger}$	1.26237	1.04234	2.64502	0.48×	0.39×

Yellow : production configuration. Purple †: randomly selected shapes (not optimization target).

TABLE III: Conv1D kernel performance: KernelEvolve-generated Triton kernel vs. PyTorch `conv1d` and `conv2d` baselines on H100 GPUs. The kernel is optimized for production recommendtion model tensor shapes (highlighted in yellow), achieving strong speedups. Performance on other shapes (highlighted in purple) varies: similar shapes benefit from the optimization, while out-of-distribution shapes show degraded performance.

erators are basic, we expect limited performance improvement. These operators primarily serve to validate KernelEvolve's end-to-end correctness rather than to demonstrate optimization potential. The key takeaway from Figure 6 is the convergence curve shape—demonstrating that KernelEvolve's search process reliably improves across diverse operator types—rather than absolute speedup magnitude.

In the following section, we evaluate KernelEvolve on key recommendation model operators, which compose multiple ATen primitives with application-specific logic. This exposes unique kernel fusion opportunities and memory access patterns that yield substantially larger optimization potential.

B. Production Recommendation Model Optimization

The baselines used in the following evaluation are production-optimized: (1) PyTorch 2.0 Inductor with `torch.compile()`—automatic operator fusion, memory planning, and hardware-specific codegen—the same stack deployed in Meta's serving systems; (2) vendor-optimized cuDNN/cuBLAS libraries for Conv1D/2D. The evaluated kernels are foundational modules in Meta's recommendation models that have already been profiled and optimized by domain experts. We present the kernel optimization results summarized in Figure 1 and the evaluation analysis for four important recommendation models in detail—Convolutional Transformer [15], [43] (Section IV-B1), WuKong [59] (Section IV-B2), InterFormer [57] (Section IV-B3), and Data Preprocessing (Section IV-B4).

1) Convolutional Transformer: Inspired by convolutional neural networks [43] and convolution-augmented Transformers [15], the Convolutional Transformer architecture combines convolutional and transformer components to capture both local and global patterns in user sequential events in large-scale recommendation systems. At the heart of this model is a stack of 1D convolution layers (Conv1D) that serve as an essential receptive module. Given that the Conv1D operations dominate the model inference time, it is a key kernel for performance optimization for deployment at-scale. To address this, we employ KernelEvolve to automatically generate and tune high-performance kernels through iterative refinement.

Performance Results. We compare KernelEvolve's optimized kernels with two PyTorch baselines on the production tensor shapes. The first baseline uses `torch.nn.functional.conv1d` directly. The second baseline is based on a common optimization technique, that reshapes input tensor to 2D using the `channels_last` memory format and invokes `torch.nn.functional.conv2d` to map it to cuDNN's heavily-optimized Tensor Core path for NHWC convolutions. Table III summarizes the performance results on the NVIDIA H100 GPUs across input tensor sizes with numeric precisions of FP16 for serving and FP32 for training, verified with atol=10^{-4}, rtol=5×10^{-4}.

On production tensor shape $(2048, 96, 96, 200)$, KernelEvolve achieves 2.30x speedup for Conv1D and 1.62x for Conv2D in FP16 consistently. For FP32 training workloads, speedups reach 1.73x for Conv1D and 1.30x for Conv2D.

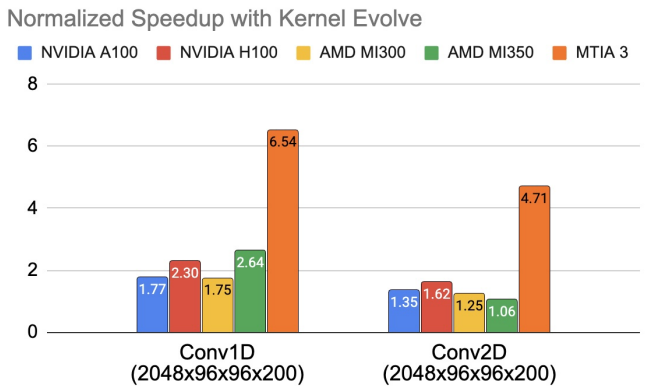

Fig. 7: Conv1D and Conv2D kernel speedup across heterogeneous hardware platforms on production shape $(2048, 96, 96, 200)$, FP16.

The generated kernel is deliberately specialized on out-of-distribution shapes (e.g., $64 \times 768 \times 768 \times 1024$), it underperforms the baselines (0.49-0.63x), confirming that KernelEvolve can effectively target optimizations on production distributions rather than arbitrary input tensor shapes (Table III).

Figure 7 presents the speedup results for Conv1D and Conv2D, respectively, over NVIDIA A100/H100 GPUs, AMD MI300/MI350 GPUs, and MTIA v3, in Meta's heterogeneous datacenter fleet. Overall, KernelEvolve is able to further improve already-optimized kernels, and it shines on the newly-introduced MTIA v3 hardware. KernelEvolve introduces 6.54x and 4.71x speedup for Conv1D and Conv2D on the MTIA v3 hardware, respectively. The framework helps accelerates the development velocity of mapping model computations Programmability and performance portability

Kernel Optimization Analysis. We dive more deeply into the source of the aforementioned performance improvements through execution trace analysis. PyTorch Conv1D launches five separate kernels that employ multiple layout transformations (`nchwToNhwcKernel`, `nhwcToNchwKernel`), Tensor Core GEMM (`sm90_xmma_fprop`), and a Triton-generated fusion kernel. Each launch incurs host-GPU synchronization and additional data movement overhead. The Conv2D baseline reduces this to four kernels, using optimized NHWC layout but still requires separate operations for layout manipulation.

KernelEvolve fuses the entire operation into two kernels: weight preparation and one main convolution kernel. Critically, the main kernel performs *both CUDA core operations (layout transformations, indexing) and Tensor Core computation (matrix multiplication) in a single launch*. This keeps the GPU fully utilized during the Triton kernel execution, enabling better parallelism between operation types.

Microarchitecture-Specific Optimizations. Beyond fusion, the generated kernel employs microarchitecture optimizations that are discovered during the search phase of KernelEvolve. KernelEvolve encompasses autotuning exploration over block sizes, warp counts, and pipeline stages, tailoring to input dimensions and convolution parameters. A 3D grid launch parallelizes grouped convolution channels, eliminating inter-group dependencies. Double-buffered execution prefetches next data blocks while computing current blocks, overlapping memory access with Tensor Core operations. Differentiated cache modifiers optimize memory hierarchy usage, using `.ca` for streaming activations, and `.cg` to reuse weights.

The kernel fusion and microarchitecture-specific performance optimizations are the results of KernelEvolve's tree search algorithm with execution feedback. This search trajectory demonstrates that graph-based search with performance-guided selection discovers increasingly efficient implementations through inference-time scaling, automatically identifying the fusion strategies and tiling configurations that manual development would require weeks to explore.

2) WuKong: For WuKong, we focus performance optimization on OptimizedFM — a core computational primitive [59]. In factorization machine (FM)-based recommendation model architectures, the pairwise dot product $XX^\top$ has $O(n^2d)$ complexity, which becomes prohibitive for real-world datasets with thousands of features. WuKong exploits the low-rank property of $XX^\top$ by introducing a learnable projection matrix $Y \in \mathbb{R}^{N \times K}$ where $K \ll N$, reducing the output from $N \times N$ to $N \times K$. Leveraging associativity, the computation reorders to: $\texttt{out} = X \cdot (X^\top Y)$, where $X \in \mathbb{R}^{B \times N \times D}$ and $Y \in \mathbb{R}^{B \times N \times K}$. Computing $X^\top Y$ first reduces complexity from $O(N^2D)$ to $O(NKD)$. This two-stage batched matrix multiplication presents a fusion opportunity: the intermediate result $X^\top Y \in \mathbb{R}^{B \times D \times K}$ can remain in registers or shared memory, eliminating global memory round-trips.

Performance Results. We evaluate KernelEvolve's ability to generate fused Triton kernels for OptimizedFM and compare it with PyTorch's native implementation using `torch.compile`. We evaluate on production shapes extracted from a deployed Wukong variant: $(B, N, D, K) \in \{(1024, 24, 224, 2198), (1024, 40, 224, 448), (1024, 48, 224, 448)\}$. Figure 8 examines speedup as a function of output dimension K with fixed $B = 1024$, $D = 224$. Small N values (24-32 features) maintain 3.0-3.5× speedup across the entire K range (256-2304), demonstrating robust performance. Medium N values (40-64 features) achieve 2.0-2.5× speedup, while larger N (96-256 features) show diminishing returns, approaching 1× as N increases. This degradation occurs because larger feature counts require more tiles to fit in SRAM—as the number of tiles grows, the overhead of tile management and accumulation eventually surpasses the benefits of on-chip computation, making direct HBM execution competitive.

Kernel Optimization Analysis. KernelEvolve generates a fused Triton kernel exploiting two key optimizations:

- *Operator fusion eliminates intermediate materialization.* The PyTorch baseline with `torch.compile` generates two separate `extern_kernels.bmm` calls. This means two matrix multiplications are executed independently. KernelEvolve fuses the operations into a single kernel: inputs are loaded once, the intermediate result $X^\top Y$

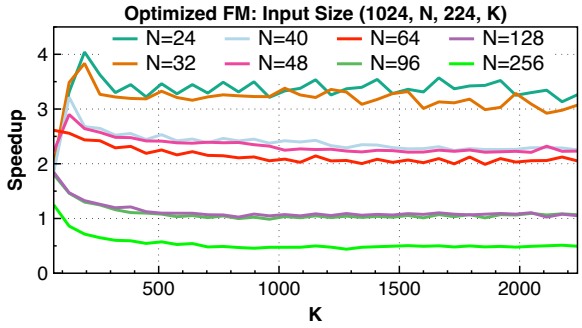

Fig. 8: Optimized FM speedup on production shapes.

remains in SRAM throughout computation, and only the final output writes to HBM. This reduces memory traffic by approximately 2x, eliminating one full read-write cycle for the intermediate tensor.

- *Shape-specific tiling optimizes for SRAM locality.* Rather than using PyTorch's standardized autotuning templates, KernelEvolve generates custom tiling configurations tailored to production tensor shapes. The kernel decomposes inputs into tiles to fit both multiplications and resulting memory requirement within the SRAM capacity. For production configurations where $B = 1024$, $D = 224$ remain fixed, tile dimensions are optimized to maximize SRAM utilization while ensuring the fused operation executes entirely on-chip.

Over the design space of production input tensor shapes where $N \leq 64$ — the majority of deployed WuKong model variants, the generated kernel demonstrates consistent speedups. For model specifications with larger feature counts, where tiling overhead dominates, the system falls back to PyTorch's unfused baseline. This shape-specific optimization strategy ensures performance gains based on workload characteristics without risking performance regressions on out-of-distribution inputs. The results validate that KernelEvolve's search-based optimization can discover kernel fusion strategies and tiling configurations tailored to deployment distributions. It achieves competitive performance while reducing development time from weeks to hours.

3) InterFormer: For InterFormer, we focus performance optimization on Personalized FeedForward Network (PFFN) — a key component of the model in Meta's recommendation and ranking system [57]. In recommendation models, user engagement sequences are inherently noisy. For example, users can explore items of interest randomly, making pure sequential modeling ineffective. InterFormer addresses this by enabling bidirectional information flow between non-sequential features and sequential features. And, PFFN is the model component that transforms sequence embeddings conditioned on non-sequence context.

The PFFN module comprises five operations executed sequentially: (1) a feed-forward neural network (batched matrix multiplication with bias), (2) GELU activation, (3) root-mean-square normalization (RMSNorm), (4) another feed-forward layer, and (5) final RMSNorm. This operator chain processes tensors $X \in \mathbb{R}^{B\times N\times D}$ with weight matrices $W_1 \in \mathbb{R}^{B\times D\times K}$ and $W_2 \in \mathbb{R}^{B\times K\times D}$, where B denotes batch size, N sequence length, D input dimension, and K hidden dimension.

Performance Results. We evaluate KernelEvolve-generated Triton kernels for the PFFN layer on production tensor shapes extracted from the deployed InterFormer models: $(B, N, D, K) \in \{(1024, 200, 256, 160), (1024, 200, 192, 96), (1024, 400, 256, 160), (1024, 150, 96, 192)\}$. The PyTorch baseline with `torch.compile` generates two separate kernels: (1) `extern_kernels.bmm` for matrix multiplication (single pass: load inputs, compute, write output), and (2) a two-pass fused Triton kernel `triton_per_fused_rms_norm_add_gelu` where the first pass loads data to perform bias addition and accumulate RMSNorm statistics, and the second pass reloads data to apply normalization. This results in three round-trips to the memory: one for BMM and two for the fused operations. While PyTorch exploits fusion opportunities among element-wise operations, the multi-pass execution and kernel separation incur redundant memory traffic.

Figure 9 analyzes speedup across production shape variations. It shows performance as a function of batch size B for five production configurations extracted from deployed InterFormer models. Peak speedups of 2.0-2.6× occur at small batch sizes ($B \leq 256$), where the fused kernel's reduced memory traffic dominates performance. As batch size increases beyond 512, speedup stabilizes at 1.2-1.4× across all configurations. This convergence reflects a fundamental trade-off: larger batches amortize kernel launch overhead for both optimized and baseline implementations, reducing the relative advantage of fusion as compute-to-memory ratio increases. The configuration $(N = 200, D = 256, K = 160)$—representative of high-dimensional production embeddings—maintains consistent 1.2× speedup at large batch sizes, validating robust performance on primary deployment targets.

Critically, all configurations maintain speedup $\geq 1.0\times$ across the tested parameter space, with the majority achieving 1.2-2x improvement. The absence of performance regressions validates KernelEvolve's shape-aware optimization approach. Generated kernels exploit fusion opportunities when tile configurations permit on-chip execution, while avoiding pathological cases through adaptive tiling strategies during search.

The PFFN case study demonstrates KernelEvolve's ability to discover non-obvious fusion opportunities through systematic search over operator compositions and tiling configurations. While human experts might identify the matrix multiplication and normalization fusion conceptually, determining the precise tile dimensions and reuse strategies that maximize SRAM occupancy across production shape distributions requires extensive trial-and-error—effort KernelEvolve automates through graph-based search with execution feedback. The 1.5-2× speedups achieved on production workloads validate that automated synthesis can match expert-level kernel implementations while reducing development cycles from weeks to hours.

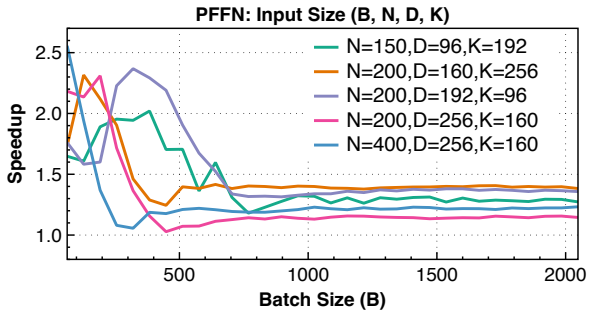

Fig. 9: PFFN speedup on various production shapes.

Kernel Optimization Analysis. KernelEvolve generates two kernel variants targeting different operator chains: First is by fusing feed-forward network with RMSNorm, and second by fusing feed-forward network, GELU, and RMSNorm. We select the highest-performing variant for production deployment, evaluating on FP16 precision matching production serving requirements. The generated kernel achieves performance improvements through two key optimizations:

- *Shape-specific tiling for production distributions.* KernelEvolve's search process incorporates production input shape ranges during kernel generation. The generated kernel employs customized tiling configurations that maximize SRAM utilization for target dimensions—in contrast to PyTorch's templated BMM kernel using generic tiling heuristics. For production shapes where $D \in [96, 256]$ and $K \in [96, 256]$, the specialized tiling ensures tiles remain SRAM-resident throughout computation, avoiding HBM fallback that occurs with one-size-fits-all tile sizes.
- *Cross-operation tile reuse.* KernelEvolve generates a unified single-pass kernel that loads tiles once, performs the complete operator chain (matrix multiplication, bias addition, GELU, RMSNorm) while data resides in SRAM, and writes final results to HBM—requiring only one load and one write per tile. The PyTorch baseline executes PFFN through two separate kernels with three total passes: (1) the first kernel (`extern_kernels.bmm`) loads inputs, performs matrix multiplication, and writes intermediate results; (2-3) the second kernel (`triton_per_fused_rms_norm_add_gelu`) executes in two passes — the first pass reloads data to perform bias addition and accumulate RMSNorm statistics and the second pass reloads data again to apply normalization.

4) Data Preprocessing Kernels: Running PyTorch models on MTIA hardware requires kernel implementations for all ATen operators in the model graph. Missing kernels force either model rewrites or fallback to CPU execution. Neither is acceptable for production latency requirement. We deploy KernelEvolve to address both model enablement and kernel optimization. On hardware with limited operator implementation coverage, KernelEvolve provides the missing implementations required for model execution. On hardware with higher coverage, KernelEvolve enables further performance improvement through kernel fusion and MTIA-specific tuning. Figure 1 presents performance speedup for the key data preprocessing kernels. In this section, we focus on the data preprocessing kernel that maps continuous features to discrete bin indices for embedding lookup in deep learning recommendation models, called Merge Bucketized Dense Transform (MBDT). Given an input tensor and per-feature border lists, MBDT performs batched bucketization — a vectorized binary search assigning each value to its corresponding bin.

Performance Results. Figure 10 compares KernelEvolve-generated kernels against PyTorch (with `torch.compile`) across various input tensor shapes on MTIA v2i and v3. On MTIA v2i, KernelEvolve achieves substantial speedups ranging from 2.94x to 9.25x. Speedup scales with input tensor sizes — smaller tensors ($64 \times 2 \times 2$) achieve 3.19x, while larger tensors ($2048 \times 2 \times 4$) reach 9.25x. This scaling behavior reflects the kernel's ability to better amortize launch overhead and exploit parallelism as working set size increases.

On MTIA v3, KernelEvolve achieves 2.31-3.09x speedup across all tensor sizes consistently also. Compared to MTIA v2i, the lower speedup for MTIA v3 is expected because the native operator coverage for MTIA v3 is higher, resulting in a stronger PyTorch baseline with less room for optimization. Nevertheless, kernel fusion and vectorized execution still deliver meaningful performance gains.

Kernel Optimization Analysis. KernelEvolve generates a fused Triton kernel with several MTIA-specific optimizations:

- *Kernel Fusion:* The entire bucketization pipeline — border lookup, binary search, and offset computation — executes in a single kernel launch, eliminating inter-kernel communication.
- *Vectorized Counting:* Instead of a scalar binary search, the kernel uses SIMD-vectorized counting — `values > border_val` is applied to blocks of 64–256 elements simultaneously. For typical small border arrays (3-10 elements), this $O(n)$ approach outperforms $O(\log n)$ binary search due to reduced control flow overhead and branch-free execution.
- *Adaptive Block Sizing:* Block size is tuned, based on input dimensions — 64 for small, 128 for medium, 256 for large input tensor sizes — to maximize Processing Element (PE) utilization.
- *Register-Resident Computation:* Intermediate results, such as, left and right counts or averages, remain in registers throughout computation. No intermediate tensor allocations occur and computation results are written directly to the output buffer.
- *MTIA-Specific Optimization:* The kernel avoids constructs that fail MTIA compilation (e.g., `tl.where` in loops), using direct boolean-to-int conversion instead. Memory loads are coalesced with proper masking, and small border arrays are cached across PE blocks.

V. Experiences and Lessons Learned

Thanks to the fast advancing capabilities of large language models specialized in coding and reasoning, the key pro-

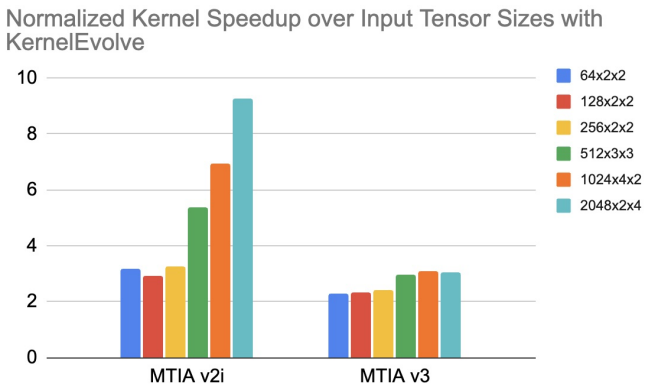

Fig. 10: MBDT kernel latency comparison. Configuration format: Batch × Features × Borders. KernelEvolve achieves 2.94-9.25x speedup on v2i and 2.31–3.09x on v3, with larger speedups at higher batch sizes.

grammability challenge to deploy a large number of deep learning recommendation tasks of distinct model architectures to different kinds of AI hardware accelerators has been significantly improved. KernelEvolve demonstrates that LLM agents can generate production-quality kernels for heterogeneous AI hardware, but this work is just the first step to unlock performance efficiency by effectively utilizing hardware heterogeneity fleetwide for production deep learning recommendation model inference. Looking forward, we expect KernelEvolve to enable a plethora of new machine learning system optimization opportunities to realize the vision of *fully automated, heterogeneous hardware-aware code generation that scales across the entire AI infrastructure stack at Meta.*

- **Hardware co-design insights from agentic exploration:** Since MTIA is absent from public LLM training data, the agent discovers optimizations purely through iterative exploration, making the resulting hardware insights particularly genuine. Two findings emerged: (1) *Local storage as pipelining bottleneck*—KernelEvolve's search consistently converged on maximizing circular buffer depth, revealing that per-PE Local Storage (384KB in MTIA v2, 512KB in v3) is the binding constraint limiting DMA-compute pipelining depth. This signal informed the LS increase to 1MB in the next-generation chip. (2) *Inter-PE communication overhead*—for Conv1D, KernelEvolve converged on a two-phase scratch-buffer approach with explicit global barriers because the hardware lacks general-purpose inter-PE data sharing primitives. The overhead of this workaround accounted for a significant fraction of kernel latency, motivating the addition of hardware-native row-wise reduction in the next-generation chip. More broadly, KernelEvolve provides a scalable co-design methodology: by observing where an agent consistently works around hardware limitations across many iterations, architects gain empirical signals about which features to prioritize in future generations.
- **New system abstract definition and operating granularity tailoring to LLM research agents:** As AI infrastructures in the industry evolves to include multiple generations and unique kinds of AI hardware, the diversity of optimization targets will grow exponentially. We envision KernelEvolve as the unified kernel generation framework for the fleet of an ever-increasing degree of heterogeneity at Meta, where AI workloads can automatically adapt to new hardware through updated specifications rather than manual engineering. This requires developing hardware abstraction primitives that capture memory hierarchies, compute capabilities, and ISA specifications in a format amenable to LLM reasoning.
- **From operators to models – rich kernel fusion optimization:** Existing optimizations target individual operators and small modules, but the most significant performance improvement potential lies at the model level. KernelEvolve expands the kernel design space beyond the boundary of a single or a few adjacent kernels. It unlocks exploration of various kernel fusion combinations without the costly human engineering time. It is already speeding up generation of novel fused kernels of different sizes. Future work can build upon KernelEvolve to reason about cross-layer fusion, global memory allocation, and end-to-end computation graphs. Combined with model transformation techniques — quantization, sparsity, architecture search, this enables co-optimization of model structure and kernel implementation, potentially discovering novel operator compositions that neither humans nor traditional compilers can identify today.
- **New abstraction boundaries for code generation and transformation:** In addition to co-optimization between model struture and kernel implementation, an interesting, yet less understood dimension is *what is the most effective code transformation layer for an LLM agent?* Triton provides a productive abstraction, but certain optimizations require lower-level control. Extending KernelEvolve to modify MLIR dialects, direct PTX/SASS, or hardware routines could unlock performance-critical scenarios where Triton's abstractions are limiting. This vertical integration — from high-level DSLs to bare-metal code — could unlock significantly higher performance efficiency for AI.
- **Global resource optimization with abstracted hardware heterogeneity**: Prior works, exploring computation scheduling across hardware of different ISAs, have been bottlenecked by the programmability and hardware diversity challenge of AI systems fleet wide. With KernelEvolve, we expect significant room for improvement in performance and energy efficiency by globally optimizing computation resources at-scale for workloads of interest.

KernelEvolve goes beyond improving operational efficiency by accelerating model inference deployment with improved inference time speedup. It also unlocks the use of heterogeneous hardware with improved utilization at-scale for Meta's datacenter fleet. While existing foundation models lack the

knowledge of proprietary hardware, such as, MTIA, we expect reinforcement learning from execution feedback to offer a path that can effectively adapt general-purpose frontier LLMs to specialized hardware without exposing proprietary innovations. From the perspective of AI sustainability, KernelEvolve advances carbon efficiency by lowering AI's energy use with kernel computation efficiency optimization and, at the same time, amortizing the upfront capex cost and embodied carbon in the AI hardware with increased utilization. We could also embed energy and carbon cost into the search process [49]. Putting sustainability as an optimization objective aligns KernelEvolve for sustainable AI [55].

VI. Related Work

High-performance kernel development has traditionally relied on vendor-optimized libraries and auto-tuning frameworks. Halide decouples algorithm specification from scheduling, enabling portable optimization across hardware targets [40]. TVM extends this with learned cost models for automated schedule search [7]. Triton provides a Python-embedded DSL that abstracts GPU programming at the block level while exposing performance-critical tiling decisions [46]. NVIDIA introduces CuTe DSL [35] to provide composable layout and tensor abstractions for Tensor Core programming with Python JIT compilation. Meta's TLX [31] adds warp-aware intrinsics and explicit pipeline control for NVIDIA Hopper and Blackwell architectures. OpenAI's Gluon dialect [37] exposes lower-level layout encoding within the Triton compiler stack whereas TileLang [51] offers a composable tiled programming model with automatic layout inference across NVIDIA and AMD GPUs. Helion [2] compiles high-level PyTorch-like syntax to autotuned Triton code. These abstractions reduce development effort but still require substantial domain expertise for novel kernel transformations and struggle to generalize across heterogeneous hardware without manual adaptation.

Beyond traditional kernel optimization, earlier work have applied evolutionary computations to improve general-purpose GPU code by designing code transformation operators at the LLVM/MLIR level [25]–[28]. More recently, LLMs have demonstrated remarkable capabilities in general-purpose code synthesis [6], [22], [41], [45]. Recent work extends these capabilities to performance-critical domains: AlphaCode [22], [44] achieves competitive programming performance through large-scale sampling, while CodeRL [20] incorporates execution feedback via reinforcement learning (RL). These advances establish the foundation for applying LLMs to kernel development, where correctness and speedup are both essential.

Several recent systems start to apply LLMs specifically for GPU kernel synthesis. KernelBench [38] benchmarks LLM capabilities across operator, fusion, and model-level difficulty tiers. AutoTriton [21] applies RL to Triton programming and KernelLLM [13] explored supervised baseline for Triton kernel generation from PyTorch modules, while TritonRL [54] trains models with execution-guided rewards. KernelAgent [1] was designed around KernelLLM to generate verified Triton kernels from PyTorch programs based on a multi-agent system. GEAK [50] targets AMD MI300X through agentic workflows, and Kevin [4] employs multi-turn RL for CUDA generation. More recently, TritorX has been introduced as an agentic AI system that aims to generate functionally correct Triton kernels from PyTorch ATen operators [18] – an important first step to enable automatic kernel generation for production deployed models to run on MTIAs. AlphaEvolve [34] leverages evolutionary search with LLMs to optimize TPU/GPU kernels. While these systems demonstrate competitive results on isolated benchmarks, they target single hardware platforms with synthetic workloads, lacking heterogeneous hardware support, production operator coverage, and deployment infrastructure integration required for industry-scale adoption.

Finally, recent work demonstrates that model performance improves predictably with increased test-time compute [42]. Chain-of-thought prompting [53], tree-of-thought search [56], and self-consistency decoding [52] can enable complex reasoning through multi-path exploration. KernelEvolve builds on these insights, applying a tree-based search algorithm with execution feedback to systematically explore kernel optimization spaces for a wide collection of heterogeneous AI hardware. Other search algorithms, such as, [48], can be easily integrated to further improve KernelEvolve's design space exploration and performance optimization results.

VII. Conclusion

This paper presents KernelEvolve— an agentic kernel coding framework to tame the ever-increasing heterogeneity challenge for AI model deployment at-scale. By carefully orchestrating the steps in the kernel generation process using state-of-the-art LLMs, we demonstrate that KernelEvolve can generate higher performing kernels for a variety of AI hardware effectively. KernelEvolve has been deployed in production to improve developer velocity and performance for a diverse collection of recommendation model architectures on NVIDIA and AMD GPUs, as well as Meta's in-house MTIAs. Taking a step further, we share insights for where the impressive performance improvement comes from. We hope the KernelEvolve framework will serve the foundation to make machine learning inference fast, efficient, and sustainable.

Acknowledgment

We thanks our colleagues at Meta for their helpful input and feedback on this work: Yoram Bachrach, Rick Chang, Wenyuan Chi, Wyatt Cook, Yuanwei Fang, Zhou Fang, Jun Ge, Kaustubh Gondkar, Wei Guo, Karen Hambardzumyan, Alec Hammond, Yujia He, Kunming Ho, Nathan Hu, Barney Huang, Keren Huang, Joe Isaacson, Martin Josifoski, Minjiang Kim, Richard Li, Zhouyang Li, Irene Liu, Alexey Loginov, Zach Marine, Yaobin Qin, Srivatsan Ramesh, Praveen Ramachandran, Mark Saroufim, Dev (Devashish) Shankar, Bidit Sharma, Ketan Singh, Jake Siso, Oleksandr Stashuk, Tejas Venkateswaran, Edan Toledo, Laura Wang, Shiguang Wang, Zhaodong Wang, Noah Weller, Tao Yang, Hongtao Yu, Abdul Zainul-Abedin, Feixiong Zhang, Qing Zhang, Sean Zhang, Haishan Zhu, Mingjie Zhu.

REFERENCES

[1] "Kernelagent — multi-agent gpu kernel synthesis," 2025. [Online]. Available: https://github.com/meta-pytorch/KernelAgent

[2] J. Ansel, "Helion: Python-embedded domain-specific language (dsl) for authoring machine learning kernels," 2025. [Online]. Available: https://github.com/pytorch/helion

[3] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski *et al.*, "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, pp. 929–947.

[4] C. Baronio, P. Marsella, B. Pan, S. Guo, and S. Alberti, "Kevin: Multi-turn rl for generating cuda kernels," *arXiv preprint arXiv:2507.11948*, 2025.

[5] Q. Carbonneaux, G. Cohen, J. Gehring, J. Kahn, J. Kossen, F. Kreuk, E. McMilin, M. Meyer, Y. Wei, D. Zhang *et al.*, "Cwm: An open-weights llm for research on code generation with world models," *arXiv preprint arXiv:2510.02387*, 2025.

[6] M. Chen, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[7] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[8] Accessed 12/13/2025. [Online]. Available: https://www.anthropic.com/news/claude-opus-4-5

[9] J. Coburn, C. Tang, S. A. Asal, N. Agrawal, R. Chinta, H. Dixit, B. Dodds, S. Dwarakapuram, A. Firoozshahian, C. Gao *et al.*, "Meta's second generation ai chip: Model-chip co-design and productionization experiences," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 1689–1702.

[10] DeepSeek, "Deepgemm," *github*, 2025. [Online]. Available: https://github.com/deepseek-ai/DeepGEMM

[11] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv e-prints*, pp. arXiv–2407, 2024.

[12] A. Firoozshahian, J. Coburn, R. Levenstein, R. Nattoji, A. Kamath, O. Wu, G. Grewal, H. Aepala, B. Jakka, B. Dreyer *et al.*, "Mtia: First generation silicon targeting meta's recommendation systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[13] Z. V. Fisches, S. Paliskara, S. Guo, A. Zhang, J. Spisak, C. Cummins, H. Leather, G. Synnaeve, J. Isaacson, A. Markosyan, and M. Saroufim, "Kernelllm: Making kernel development more accessible," 6 2025, corresponding authors: Aram Markosyan, Mark Saroufim. [Online]. Available: https://huggingface.co/facebook/KernelLLM

[14] Accessed 12/13/2025. [Online]. Available: https://openai.com/index/introducing-gpt-5/

[15] A. Gulati, J. Qin, C.-C. Chiu, N. Parmar, Y. Zhang, J. Yu, W. Han, S. Wang, Z. Zhang, Y. Wu *et al.*, "Conformer: Convolution-augmented transformer for speech recognition," *arXiv preprint arXiv:2005.08100*, 2020.

[16] U. Gupta, S. Hsia, J. Zhang, M. Wilkening, J. Pombra, H.-H. S. Lee, G.-Y. Wei, C.-J. Wu, and D. Brooks, "Recpipe: Co-designing models and hardware to jointly optimize recommendation quality and performance," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, 2021.

[17] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The Architectural Implications of Facebook's DNN-Based Personalized Recommendation ," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[18] A. M. Hammond, A. Markosyan, A. Dontula, S. Mahns, Z. Fisches, D. Pedchenko, K. Muzumdar, N. Supper, M. Saroufim, J. Isaacson, L. Wang, W. Hunt, K. Gondkar, R. Levenstein, G. Synnaeve, R. Li, J. Kahn, and A. Mathews, "Agentic operator generation for ml asics," 2025. [Online]. Available: https://arxiv.org/abs/2512.10977

[19] Z. Jiang, D. Schmidt, D. Srikanth, D. Xu, I. Kaplan, D. Jacenko, and Y. Wu, "Aide: Ai-driven exploration in the space of code," 2025. [Online]. Available: https://arxiv.org/abs/2502.13138

[20] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 314–21 328, 2022.

[21] S. Li, Z. Wang, Y. He, Y. Li, Q. Shi, J. Li, Y. Hu, W. Che, X. Han, Z. Liu *et al.*, "Autotriton: Automatic triton programming with reinforcement learning in llms," *arXiv preprint arXiv:2507.05687*, 2025.

[22] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[23] G. Liao, Y. Liu, J. Chen, and D. J. Abadi, "Bullion: A column store for machine learning," *Proceedings of 15th Conference on Innovative Data Systems Research (CIDR)*, 2025.

[24] G. Liao, H. Qin, Y. Wang, A. Golden, M. Kuchnik, Y. Yetim, J. J. Ang, C. Fu, Y. He, S. Hsia, Z. Jiang, D. Li, U. Pashkevich, V. Puvvada, F. Shi, M. Steiner, R. Xiao, N. Yan, X. Yu, Z. Fang, R. Levenstein, K. Ho, H. Zhu, A. Hammond, R. Li, A. Mathews, K. Gondkar, A. Zainul-Abedin, K. Singh, H. Yu, W. Chi, B. Huang, S. Zhang, N. Weller, Z. Marine, W. Cook, C.-J. Wu, and G. Liu, "Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta," 2026. [Online]. Available: https://arxiv.org/abs/2512.23236

[25] J.-Y. Liou, M. Awan, S. Hofmeyr, S. Forrest, and C.-J. Wu, "Understanding the power of evolutionary computation for gpu code optimization," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*, 2022.

[26] J.-Y. Liou, S. Forrest, and C.-J. Wu, "Genetic improvement of gpu code," in *2019 IEEE/ACM International Workshop on Genetic Improvement (GI)*, 2019.

[27] J.-Y. Liou, X. Wang, S. Forrest, and C.-J. Wu, "Gevo: Gpu code optimization using evolutionary computation," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, 2020.

[28] ——, "Gevo-ml: a proposal for optimizing ml code with evolutionary computation," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '20, 2020, p. 1849–1856.

[29] M. Lui, Y. Yetim, Ö. Özkan, Z. Zhao, S.-Y. Tsai, C.-J. Wu, and M. Hempstead, "Understanding capacity-driven scale-out neural recommendation inference," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021, pp. 162–171.

[30] Meta, "Pytorch profiler," *Pytorch at Meta*, 2021. [Online]. Available: https://docs.pytorch.org/tutorials/recipes/recipes/profiler_recipe.html

[31] ——, "Tlx - triton low-level language extensions," 2025. [Online]. Available: https://github.com/facebookexperimental/triton

[32] ——, "Tritonbench," *Pytorch at Meta*, 2025. [Online]. Available: https://github.com/meta-pytorch/tritonbench

[33] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," 2019. [Online]. Available: https://arxiv.org/abs/1906.00091

[34] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. R. Ruiz, A. Mehrabian, M. P. Kumar, A. See, S. Chaudhuri, G. Holland, A. Davies, S. Nowozin, P. Kohli, and M. Balog, "Alphaevolve: A coding agent for scientific and algorithmic discovery," 2025. [Online]. Available: https://arxiv.org/abs/2506.13131

[35] NVIDIA, "Introduction to cute dsl," 2025. [Online]. Available: https://docs.nvidia.com/cutlass/latest/media/docs/pythonDSL/cute_dsl_general/dsl_introduction.html

[36] Nvidia, "Nsight compute cli - v2025.3.1," *Nvidia Docs*, 2025. [Online]. Available: https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

[37] OpenAI, "Introduction to gluon," 2025. [Online]. Available: https://github.com/triton-lang/triton/blob/main/python/tutorials/gluon/01-intro.py

[38] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, and A. Mirhoseini, "Kernelbench: Can llms write efficient gpu kernels?" *arXiv preprint arXiv:2502.10517*, 2025.

[39] PyTorch, "Fbgemm," *github*, 2025. [Online]. Available: https://github.com/pytorch/FBGEMM

[40] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[41] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[42] C. Snell, J. Lee, K. Xu, and A. Kumar, "Scaling llm test-time compute optimally can be more effective than scaling model parameters," *arXiv preprint arXiv:2408.03314*, 2024.

[43] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[44] A. Team, "Alphacode 2 technical report," 2023. [Online]. Available: https://storage.googleapis.com/deepmind-media/AlphaCode2/AlphaCode2_Tech_Report.pdf

[45] F. C. team, J. Copet, Q. Carbonneaux, G. Cohen, J. Gehring, J. Kahn, J. Kossen, F. Kreuk, E. McMilin, M. Meyer, Y. Wei, D. Zhang, K. Zheng, J. Armengol-Estapé, P. Bashiri, M. Beck, P. Chambon, A. Charnalia, C. Cummins, J. Decugis, Z. V. Fisches, F. Fleuret, F. Gloeckle, A. Gu, M. Hassid, D. Haziza, B. Y. Idrissi, C. Keller, R. Kindi, H. Leather, G. Maimon, A. Markosyan, F. Massa, P.-E. Mazaré, V. Mella, N. Murray, K. Muzumdar, P. O'Hearn, M. Pagliardini, D. Pedchenko, T. Remez, V. Seeker, M. Selvi, O. Sultan, S. Wang, L. Wehrstedt, O. Yoran, L. Zhang, T. Cohen, Y. Adi, and G. Synnaeve, "Cwm: An open-weights llm for research on code generation with world models," 2025. [Online]. Available: https://arxiv.org/abs/2510.02387

[46] P. Tillet, H.-T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019, pp. 10–19.

[47] E. Toledo, K. Hambardzumyan, M. Josifoski, R. Hazra, N. Baldwin, A. Audran-Reiss, M. Kuchnik, D. Magka, M. Jiang, A. M. Lupidi, A. Lupu, R. Raileanu, K. Niu, T. Shavrina, J.-C. Gagnon-Audet, M. Shvartsman, S. Sodhani, A. H. Miller, A. Charnalia, D. Dunfield, C.-J. Wu, P. Stenetorp, N. Cancedda, J. N. Foerster, and Y. Bachrach, "Ai research agents for machine learning: Search, exploration, and generalization in mle-bench," *arXiv*, 2025. [Online]. Available: https://arxiv.org/abs/2507.02554

[48] ——, "Ai research agents for machine learning: Search, exploration, and generalization in mle-bench," 2025. [Online]. Available: https://arxiv.org/abs/2507.02554

[49] I. Wang, N. Ardalani, M. Elhoushi, D. Jiang, S. Hsia, E. Sumbul, D. Mahajan, C.-J. Wu, and B. Acun, "Catransformers: Carbon aware transformers through joint model-hardware optimization," 2025. [Online]. Available: https://arxiv.org/abs/2505.01386

[50] J. Wang, V. Joshi, S. Majumder, X. Chao, B. Ding, Z. Liu, P. P. Brahma, D. Li, Z. Liu, and E. Barsoum, "Geak: Introducing triton kernel ai agent & evaluation benchmarks," *arXiv preprint arXiv:2507.23194*, 2025.

[51] L. Wang, Y. Cheng, Y. Shi, Z. Tang, Z. Mo, W. Xie, L. Ma, Y. Xia, J. Xue, F. Yang *et al.*, "Tilelang: A composable tiled programming model for ai systems, 2025," *URL https://arxiv. org/abs/2504.17577*, vol. 2, 2025.

[52] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," *arXiv preprint arXiv:2203.11171*, 2022.

[53] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[54] J. Woo, S. Zhu, A. Nie, Z. Jia, Y. Wang, and Y. Park, "Tritonrl: Training llms to think and code triton without cheating," *arXiv preprint arXiv:2510.17891*, 2025.

[55] C.-J. Wu, R. Raghavendra, U. Gupta, B. Acun, N. Ardalani, K. Maeng, G. Chang, F. A. Behram, J. Huang, C. Bai, M. Gschwind, A. Gupta, M. Ott, A. Melnikov, S. Candido, D. Brooks, G. Chauhan, B. Lee, H.-H. S. Lee, B. Akyildiz, M. Balandat, J. Spisak, R. Jain, M. Rabbat, and K. Hazelwood, "Sustainable ai: Environmental implications, challenges and opportunities," 2022. [Online]. Available: https://arxiv.org/abs/2111.00364

[56] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *Advances in neural information processing systems*, vol. 36, pp. 11 809–11 822, 2023.

[57] Z. Zeng, X. Liu, M. Hang, X. Liu, Q. Zhou, C. Yang, Y. Liu, Y. Ruan, L. Chen, Y. Chen *et al.*, "Interformer: Effective heterogeneous interaction learning for click-through rate prediction," in *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, 2025, pp. 6225–6233.

[58] J. Zhai, L. Liao, X. Liu, Y. Wang, R. Li, X. Cao, L. Gao, Z. Gong, F. Gu, J. He, Y. Lu, and Y. Shi, "Actions speak louder than words: trillion-parameter sequential transducers for generative recommendations," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24, 2024.

[59] B. Zhang, L. Luo, Y. Chen, J. Nie, X. Liu, D. Guo, Y. Zhao, S. Li, Y. Hao, Y. Yao *et al.*, "Wukong: Towards a scaling law for large-scale recommendation," *arXiv preprint arXiv:2403.02545*, 2024.

[60] B. Zhang, L. Luo, Y. Chen, J. Nie, X. Liu, D. Guo, Y. Zhao, S. Li, Y. Hao, Y. Yao, G. Lakshminarayanan, E. D. Wen, J. Park, M. Naumov, and W. Chen, "Wukong: Towards a scaling law for large-scale recommendation," 2024. [Online]. Available: https://arxiv.org/abs/2403.02545

[61] B. Zhang, L. Luo, X. Liu, J. Li, Z. Chen, W. Zhang, X. Wei, Y. Hao, M. Tsang, W. Wang, Y. Liu, H. Li, Y. Badr, J. Park, J. Yang, D. Mudigere, and E. Wen, "Dhen: A deep and hierarchical ensemble network for large-scale click-through rate prediction," 2022. [Online]. Available: https://arxiv.org/abs/2203.11014

[62] M. Zhao, N. Agarwal, A. Basant, B. Gedik, S. Pan, M. Ozdal, R. Komuravelli, J. Pan, T. Bao, H. Lu, S. Narayanan, J. Langman, K. Wilfong, H. Rastogi, C.-J. Wu, C. Kozyrakis, and P. Pol, "Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022.

[63] M. Zhao, D. Choudhary, D. Tyagi, A. Somani, M. Kaplan, S.-H. Lin, S. Pumma, J. Park, A. Basant, N. Agarwal, C.-J. Wu, and C. Kozyrakis, "Recd: Deduplication for end-to-end deep learning recommendation model training infrastructure," 2023. [Online]. Available: https://arxiv.org/abs/2211.05239

[64] K. Zhou, Y. Fang, and C. Robeck, "Proton: Portable performance profiling," *Triton Conference'25*, 2025. [Online]. Available: https://docs.google.com/presentation/d/1rBniQNaFUYJEQ_6bZtyhnSSNZmE7l9tc