AGENTIC OPERATOR GENERATION FOR ML ASICS

Alec M. Hammond [*1] **Aram Markosyan** [*1] **Aman Dontula** [1] **Simon Mahns** [1] **Zacharias Fisches** [1] **Dmitrii Pedchenko** [1] **Keyur Muzumdar** [1] **Natacha Supper** [1] **Site Cao** [1] **Haishan Zhu** [1] **Mark Saroufim** [1] **Joe Isaacson** [1] **Laura Wang** [1] **Warren Hunt** [1] **Kaustubh Gondkar** [1] **Roman Levenstein** [1] **Gabriel Synnaeve** [1] **Richard Li** [1] **Jacob Kahn** [1] **Ajit Mathews** [1]

ABSTRACT

We present TritorX, an agentic AI system designed to generate functionally correct Triton PyTorch ATen kernels at scale for emerging accelerator platforms. TritorX integrates large language models with a custom linter, JIT compilation, and a PyTorch OpInfo-based test harness. This pipeline is compatible with both real Meta Training and Inference Accelerator (MTIA) silicon and in hardware simulation environments for next-generation devices. In contrast to previous kernel-generation approaches that prioritize performance for a limited set of high-usage kernels, TritorX prioritizes coverage. Our system emphasizes correctness and generality across the entire operator set, including diverse data types, shapes, and argument patterns. In our experiments, TritorX successfully generated kernels and wrappers for 481 unique ATen operators that pass all corresponding PyTorch OpInfo tests (over 20,000 in total). TritorX paves the way for overnight generation of complete PyTorch ATen backends for new accelerator platforms.

1 INTRODUCTION

The rapid adoption of machine learning (ML) and artificial intelligence (AI) hardware is projected to drive US datacenter power consumption to between 6.7% and 12.0% of total electricity usage by 2028 (Shehabi et al., 2024), highlighting an urgent need for efficient accelerator hardware to support large-scale model inference and training. In response, the industry is investing heavily in heterogeneous datacenter fleets that incorporate a variety of accelerator solutions, including custom silicon ASICs tailored to specific workloads and requirements (Silvano et al., 2025). Namely, the Meta Training and Inference Accelerator (MTIA) currently serves recommendation models (DLRM) (Naumov et al., 2019) to billions of users across Facebook, Instagram, and Threads, while simultaneously reducing total cost of ownership by 44% compared to GPUs (Coburn et al., 2025).

However, despite the obvious advantages offered by in-house accelerators, each new platform requires significant engineering labor to build a software ecosystem compatible with existing tools such as PyTorch (Paszke et al., 2019), given the large set of required tensor operators (Kahn et al., 2022). An aspect therein is *operator coverage*, or the fraction of operators in ATen (Paszke et al., 2017) — PyTorch's

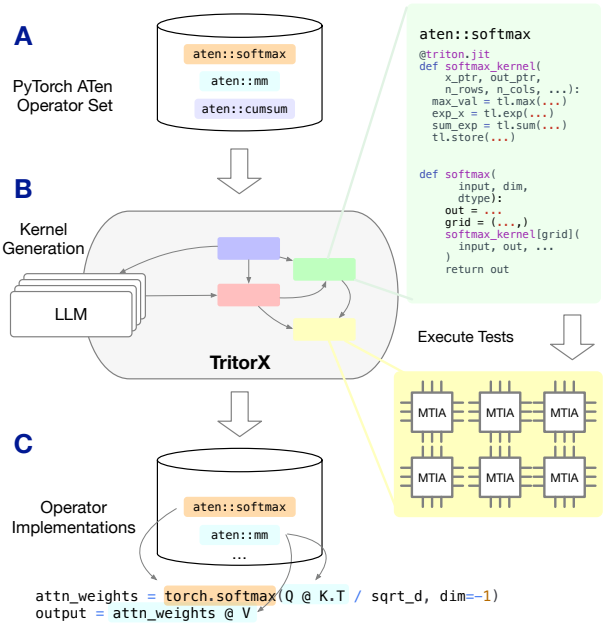

Figure 1. TritorX System Overview. **(A)** OpInfo operators and their PyTorch docstring/signature are selected for generation. **(B)** TritorX iterates on each operator in a finite-state machine feedback loop. Kernel-wrapper pairs are generated using a large language model (LLM). Production infrastructure allows for simultaneous generation and testing at scale. **(C)** The generation task is successful if an operator passes all corresponding OpInfo tests.

[*]Equal contribution [1]Meta, Menlo Park, USA. Correspondence to: Alec Hammond <alechammond@meta.com>, Jacob Kahn <jacobkahn@meta.com>.

Proceedings of the 9^{th} MLSys Conference (Industry Track), Bellevue, WA, USA, 2026.

tensor library — that have kernels executing natively on a particular accelerator. Establishing and maintaining comprehensive operator coverage is an arduous task that is needed towards running inference with new or for prototyping new model architectures for training. In other words, while new accelerator platforms provide competitive machine performance per unit cost, unleashing this performance *requires* a comprehensive kernel backend for an amenable developer experience.

To address this challenge, we present an agentic AI system capable of generating functionally correct Triton ATen kernels for MTIA at scale. This new tool, which we call TritorX, leverages open- and closed-source, large language models (LLM) paired with execution feedback in a finite state machine (FSM) harness to generate, compile, and test hundreds of kernels *directly* on MTIA hardware. TritorX is compatible with MTIA's production infrastructure, producing kernel-wrapper pairs that can be immediately registered within PyTorch and can be used for experimental model training, or in a production inference model. TritorX can also generate new PyTorch backends for upcoming accelerator generations via hardware simulation, providing important feedback to hardware and compiler engineers *before* tape-out. In the limit, we envision implementing a kernel backend for a new chipset overnight.

We note that TritorX differs from other kernel generation efforts, in that our framework strictly optimizes for correctness and generalizability across an entire backend, rather than performance over a narrow subset of critical path kernels (Lange et al., 2025a). For example, TritorX is configured to generate kernel-wrapper pairs that are compatible with a wide range of quantization data types, tensor shapes, or PyTorch argument inputs. In some cases, TritorX will generate multiple kernel implementations for a particular operator, and the corresponding dispatch logic is implemented in the wrapper function. The execution feedback mechanism of the TritorX harness takes into account the semantics and intrinsics of the MTIA-specific triton dialect known as Triton MTIA. This mechanism allows TritorX to use standard language models to produce working kernels without any additional MTIA-specific model training or fine-tuning. Notably, we only provide TritorX with the ATen operator docstring to generate the corresponding implementation. Fig. 1 describes the end-to-end workflow for accelerator enablement.

TritorX performs in-context learning iteratively, distilling hardware requirements and their corresponding Triton semantics based on the feedback obtained directly via tools like the compiler. Previous works demonstrated that agentic harnesses will often "cheat" at the generation task by dispatching to the host or calling other undefined PyTorch functions in the operator wrapper (Lange et al., 2025b). We avoid this by incorporating a custom linter that catches unauthorized uses of these functions or utilities and forces correction via feedback.

Central to TritorX is the integration of multiple *testing* frameworks to ensure correctness across different data types, tensor shapes, or input arguments. We use OpInfo[1], a PyTorch-native testing framework, along with a custom test harness that pulls test data from models in production.

In summary, our contributions are as follows:

- **Systems.** TritorX highlights a minimalistic set of key design components that allow the LLM with *execution feedback* to iteratively refine a kernel proposal to achieve *functional correctness on custom hardware*. TritorX runs either directly on deployed silicon, enabling *rapid ASIC enablement*, or via hardware simulation, enabling *prototyping for future devices*. We deploy a custom linter to detect and prevent "cheating." To our knowledge, it is one of the first design proposals in the literature to tackle AI-kernel generation for custom hardware that is different from GPUs.
- **Testing suite**. For testing the functional correctness of generated kernels, we chose the OpInfo test suite that contains more than *20,000 correctness tests*. We further validate the generated kernels on production data not used during the generation process. To our knowledge, this is one of the first attempts in the literature to employ a *thorough testing suite* that goes beyond a few unit tests to check for kernel functional correctness.
- **Operator Coverage.** We focus on maximizing the kernel *coverage* for ATen operators to enable the full calculation of a model's forward pass *on-device*. This approach allows us to quickly obtain a working implementation and focus on benchmarking and profiling for future performance refinement. In total, TritorX produced 481 ATen Op kernels that passed all their corresponding OpInfo tests, which amounts to *84.0% MTIA-compatible ATen Ops* in our ensemble evaluation in Fig. 3.
- **Practical Utility**. We observe that the comprehensiveness of OpInfo tests is a good *proxy for production deployment conditions*. Namely, we capture inputs for several production models deployed in production at Meta. In §4.1 we show that more than 80% of kernels that pass OpInfo tests also pass the correctness tests on production inputs. With the execution feedback from additional iterations of TritorX on production inputs, we can achieve *87-100% ATen operator coverage on production inputs*.

[1] OpInfo in the PyTorch core.

2 BACKGROUND

In this section, we describe the MTIA architecture, the Triton MTIA dialect, and how TritorX can be instrumented to generate kernels at scale.

The MTIA architecture employs a grid of 8x8 processing elements (PEs) responsible for executing the core kernel workloads (Coburn et al., 2025). Each PE consists of two RISC-V cores and various fixed-function units (FFUs) responsible for implementing dedicated computations, such as direct memory accesses (DMAs) and dot products. Fig. E.6 illustrates the MTIA architecture.

In particular, MTIA features a unique memory hierarchy with a significant amount of local SRAM available to all PEs via a series of crossbars. Dedicated PE FFUs facilitate the abstraction of circular buffers, enabling efficient pipelining of computation and communication needed to amortize the overall latency resulting from data movement. The computational savings gained by this approach allow MTIA to leverage cheaper LPDDR DRAM instead of HBM. Importantly, significant effort was made to mitigate overhead related to kernel or job dispatch from the host, enabling eager-mode workflows.

To execute kernels on MTIA, developers can author the workload using a C++ API, which is compiled with an LLVM backend, or using a custom Triton dialect adapted specifically for MTIA. Triton is an open-source Python library that provides a domain-specific language (DSL) for writing highly efficient custom GPU kernels (Tillet et al., 2019b). It simplifies GPU programming by allowing developers to express complex parallel computations using intuitive block-based abstractions while automatically handling low-level details such as memory access patterns and synchronization. This enables users to achieve performance comparable to hand-written CUDA code, without requiring deep expertise in GPU architectures.

Although Triton was written to express the semantics of GPU computation, several of the existing semantics can be directly translated to corresponding MTIA hardware features. For example, instead of mapping Triton blocks to GPU threads, we can map them to the MTIA PE grid and rely on masking within loads and stores such that tensor boundaries are respected. Furthermore, loads and stores can take advantage of the MTIA DMA engine for structured memory access. When intrinsics do not perfectly match, we can *augment* the underlying Triton feature-set with specific device libraries (e.g. to leverage FFUs that implement nonlinear activations). Triton MTIA is a dedicated dialect intended to facilitate the above mappings, along with various other performance optimizations via the compiler backend.

Although Triton MTIA intends to preserve the existing Triton semantics as much as possible, there are certain hardware requirements that force notable deviations. For example, MTIA requires 32-byte aligned memory access patterns, and load/store operations will fail if this is not satisfied. On the surface, this may seem problematic when trying to generate kernels using off-the-shelf models. However, Triton MTIA has detailed assert messages and error handling which provides the necessary feedback that the models need to adapt the vanilla Triton code for MTIA. Indeed, building a compiler tool-chain that gives descriptive feedback is an important part of leveraging automated approaches for code generation.

The challenge is then to implement an execution pipeline that is compatible with the existing production infrastructure. MTIA (and all future hardware versions) are deployed in a productionized Linux container ecosystem (Tang et al., 2020). Typical production workflows require an end-to-end workload that can e.g. serve or train a model for a particular service at scale, which also requires a complicated kernel registration stack compatible with the PyTorch ecosystem. However, the Triton JIT allows us to generate, compile, and test kernels on the fly, even within these productionized containers, allowing us to run numerous experiments in parallel.

A naive approach to generating Triton MTIA code would be to add comprehensive specifications of the hardware requirements and corresponding Triton semantics differences to the first prompt of the LLM to see if the generated code passes relevant tests. In practice, however, comprehensive accelerator documentation lags behind other stack components. Early attempts at generating Triton for MTIA with a simple prompt-engineering approach resulted in significant manual labor and did not scale.[2]

In contrast, TritorX effectively performs in-context learning, iteratively distilling hardware requirements and their corresponding Triton semantics based on the feedback obtained directly via interaction with the linter, compiler, and debugger.

Although recent kernel generation frameworks often rely on a dedicated reasoning agent with prescribed tool calling (Lange et al., 2025b; Wang et al., 2025; Chen et al., 2025b; Andrews & Witteveen, 2025; Li et al., 2025a;b), the integration of an FSM architecture in our production environment offers several benefits. An FSM offers explicit guardrails around what is executed and performed, and allows faster debugging of the key components of the system, which is an important requirement when dealing with production-ready systems. Additionally, the backend of the FSM enables compilation and testing on both the deployed MTIA machines and on the QEMU simulator of the future MTIA generations.

[2] See zero-shot results for GEAK.

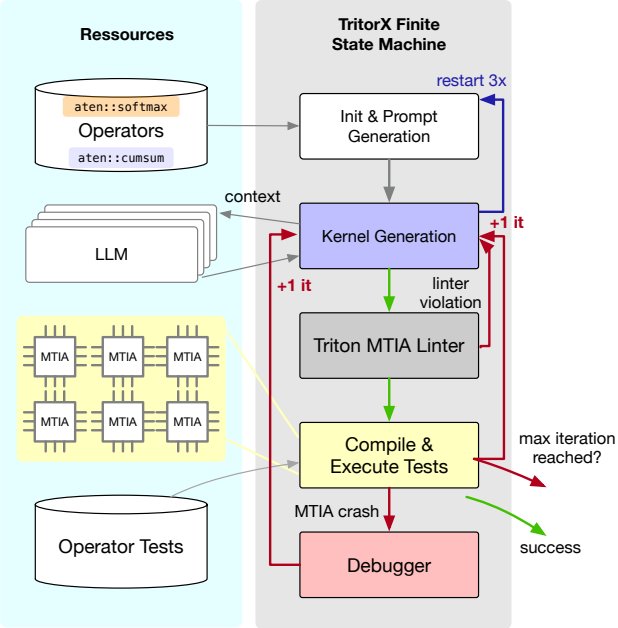

Figure 2. Finite State Machine of our kernel-generation agent TritorX. Proposal kernel-wrapper pairs are only generated during the "Kernel Generation" state (which dispatches to the LLM). All other states process the result and update a feedback prompt, where needed.

3 System Design

We implemented TritorX as a finite-state machine (FSM) with dedicated tools and routines, including linting, compiling, testing, debugging, and LLM calls. Here, we describe the system architecture behind TritorX, along with the test harness used to validate the results during generation.

3.1 TritorX Agent

Fig. 2 illustrates the overall design of the TritorX system. The harness integrates the key components shown in prior work to provide effective inference-time feedback for the kernel generation task (Ouyang et al., 2025, §5.1.2; Baronio et al., 2025, §4.1; Li et al., 2025c, §3).

Each operator is generated in a self-contained session during which all prescribed tests are performed. This session is configurable up front, allowing us to easily prototype different LLM models, disable/enable individual states (like the linter), and sweep TritorX hyperparameters (e.g., max number of iterations). The design of the system is as follows:

Init. The routine begins with an initial prompt consisting of the task description, output requirements, the documentation (docstring) of the PyTorch operator, and three handcrafted examples (Appendix B). Specifically, the prompt asks the model to generate a python wrapper matching the designated PyTorch operator signature (described in the docstring) and one or more Triton kernels that implement the functionality itself. Often times, ATen docstrings will reference the docstrings of other operators (e.g., `argmax` references `max`). We built a directed acyclic graph of all docstrings, allowing us to include "nested" docstrings for completeness. Further instructions prescribing the expected output format expected by the downstream parser are also provided.

Kernel Generation & Linter. The reasoning LLM uses the prompt to generate an initial candidate wrapper/kernel pair, which is then sent to a custom Triton MTIA Linter. The linter is responsible for the following tasks: (1) ensuring the output wrapper and kernel code is compatible with the Triton JIT harness; (2) ensuring the provided implementation does not "cheat" by dispatching into other operators that may not yet be implemented or operators on the CPU; (3) ensuring the provided code uses valid Triton MTIA syntax and libraries, as not all of upstream Triton is available on MTIA. The linter is lightweight and configurable (Appendix D). If a lint violation is detected, a structured report is generated and sent back to the model as feedback for correction (Appendix B). The process is repeated until no lint errors are produced or the maximum number of LLM calls is reached. This is in contrast to some agentic frameworks which also rely on an LLM to check for cheating (Li et al., 2026).

Testing. If no linter errors are detected, the wrapper and kernel code is passed to a dedicated Triton JIT compilation harness compatible with the MTIA infrastructure. Depending on the operator configuration, which prescribes the supported datatypes, a series of tests derived from and production-data are synthesized. The test runner loops through each test, recompiling as needed (e.g. for new datatypes). If compilation is successful and the test executes without any runtime errors, the same inputs are moved to the host and executed using a reference ATen CPU implementation of the operator. The outputs for both the generated MTIA kernel and the CPU reference kernel are compared using a heuristic the same tolerances specified by PyTorch. If the results are within the specified tolerance, the process repeats with the next test. As soon as the runner encounters a compilation failure, a runtime error, or an accuracy error, the routine breaks and proceeds to a "feedback" state responsible for determining what to do next.

Feedback & Debugger. The feedback state analyzes how successful the test runner was and determines what kind of feedback prompt is needed for another LLM iteration, or if further debugging is needed. For example, if the most recent run resulted in a runtime crash which produced a crash dump, the crash dump is loaded in an LLDB-based debugger. The debugger pulls basic information about the backtrace, decoded registers, and other frame information to

provide as context for the revised prompt. Example insights include details around memory access violations.

In the case of a compiler failure, depending on the hyperparameter config, we optionally summarize the compiler log using a secondary LLM instance (also configurable up front). Triton MTIA compiler logs can easily consume thousands of tokens, so surfacing the most relevant facets of the compiler error to the main LLM session serves towards managing limited context windows.

If the feedback state detects an accuracy error, a summary of the MTIA output tensor(s) and the CPU output tensor(s) is included in the feedback prompt. Even in the case of large output tensors, an abbreviated summary of the tensor values is often enough context for the model to reason about the potential inaccuracy (Appendix C).

Termination. Once an appropriate feedback prompt is crafted, the process repeats until one of the following conditions occurs: (1) all tests pass, in which case the routine exits successfully; (2) the maximum number of prescribed LLM calls has been reached, and the routine exits; (3) the LLM context window saturates, and a new LLM dialog session starts using the most recent wrapper/kernel generation as an initial proposal; (4) an unexpected error crashes the main process. We implemented comprehensive exception handling throughout the TritorX, including launching containerized subprocesses where necessary, to avoid crashing the main process whenever possible.

In order to generate operators at scale, the above process can be executed in an embarrassingly parallel fashion for every operator specified and even repeated for operators that failed. These large-scale runs are configured by the operators of interest, the desired datatypes, the LLM parameters (e.g., model, context length, temperature), the run parameters (e.g., maximum number of LLM calls, maximum number of dialog sessions) and the testing complexity. The operators are compiled and executed on productionized MTIA machines. The LLM calls themselves are processed by a centralized inference platform service capable of handling a high volume of requests needed for large-scale runs.

3.2 ATen Operators and OpInfo Testing

To evaluate the viability of our approach at scale, we generate kernels for the operators defined within the PyTorch OpInfo test suite. Importantly, OpInfo aims to rigorously test an operator's coverage by providing "samples" for all the supported data-types, tensor shape, and input arguments. For example, OpInfo contains *hundreds* of tests for the `linalg.vector_norm` operator, which rigorously test different input and output tensor shapes, datatypes, and argument configurations.

Using OpInfo as our primary test harness creates an end-to-end generation pipeline with orders of magnitude more tests than the state of the art (Ouyang et al., 2025). By covering more of the input space during the generation process, we expect the resulting implementation to more reliably work in arbitrary prototyping and production environments. That being said, we recognize that ensuring perfect test coverage across the entire input space is impossible. To account for this, we introduce a secondary test harness specifically for production models that consists of production input data. This additional test suite allows us to quantify how well an operator was generalized using the OpInfo test suite. If gaps are identified during the generation stage (such that production-data tests failed), then TritorX is able to resolve the coverage gap.

There are certain limitations with MTIA hardware such that certain operators and tests are either not compatible or not relevant for the target workloads. For example, MTIA does not support complex numbers, so we remove those corresponding operators from the generation list (e.g., FFT operators). Similarly, validating the outputs for *random-number operators* between the device and host is particularly challenging due to differences in the underlying random number generation algorithm. As such, we also remove these operators from consideration. The resulting operator list consists of 568 unique operators (filtered down from 629). We also only test for `bfloat16`, `float16`, `float32`, `int32`, and `int64`. In total, this results in more than 20,000 tests in all operators, and we only classify an operator as successful if it passes all the corresponding operator tests.

As we mentioned earlier, for all of these operators, we specifically target the MTIA dialect of Triton. While there are limitations to the granularity of kernel generation we can achieve with Triton (compared to, for e.g. C++), targeting Triton yields kernels that engineers can easily read, debug, and modify, allowing the library to evolve without regeneration. Additionally, LLMs are good at code generation tasks involving Python, and Triton being a Python DSL allows us to take advantage of that strength.

4 Experiments and Results

We now present the experiments used to validate our approach. We first present an aggregate result consisting of kernels that span all MTIA-compatible OpInfo operators over multiple large-scale runs. From this set of generated operators, we "productionize" various first- and third-party models. We further expand our test harness for these operators by incorporating additional correctness tests that leverage *production data* and identify additional gaps not originally captured by OpInfo. Finally, we ablate over various TritorX configurations to highlight which aspects of our pipeline matter and why.

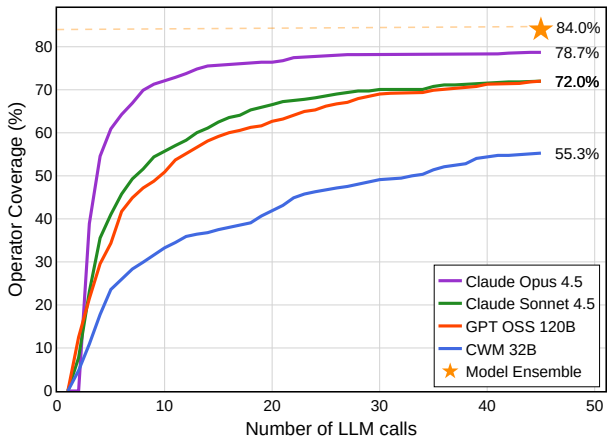

Figure 3. Number of LLM calls per operator to produce a correct kernel, cumulatively plotted for different harness configurations and models. "Ensemble" results display *total coverage* achieved by taking the union of all displayed runs, indicating some non-overlapping coverage exists from different models.

Our baseline setup for these experiments is to run TritorX over all the 568 MTIA-compatible OpInfo operators with the following configuration:

- Maximum of 3 TritorX attempts (i.e., LLM dialog sessions) per operator to generate a kernel that passes all tests and declares Success;
- Each attempt is allowed a maximum of 15 LLM calls, or, in other words, 15 full iterations through the state machine until Failure is declared for the attempt;
- Either Code World Model (CWM, (Copet et al., 2025)) or GPT-OSS 120B (OpenAI, 2025), Claude Sonnet 4.5 (Anthropic, 2025b) or Claude Opus 4.5 (Anthropic, 2025a) were used as the kernel-generating LLM. All models were configured with a context length of $131,072$ and temperature set to 1.0. We set the top-P to 0.95 for CWM and 1.0 for GPT-OSS. The GPT-OSS reasoning was set to "high."
- Llama-4-Maverick is used as the feedback summarization model with the same generation parameters as CWM.

We dispatch the generation jobs across 200 production MTIA devices (either silicon or simulation), which are able to finish 95% of a run in 2 hours. The remaining tail often results from e.g., poor reasoning trajectories, and can take another 6-8 hours to complete. New runs can be dispatched concurrently. With this infrastructure in place, we executed multiple runs, with subsequent runs focusing on operators that failed previous runs.

From these aggregated runs, we achieved **84.0%** operator coverage on all MTIA-compatible OpInfo operators. Here we consider an operator covered if the generated kernel-wrapper pair passes 100% of the sample OpInfo tests. These results were aggregated across multiple runs, including ablations over models and configuration parameters. Fig. 3 illustrates the cumulative operator coverage as a function of LLM calls for different configurations.

We identified two dominant failure modes from the uncovered operators: (1) half-precision numerical errors, where generated kernels produce correct results for higher precision but lack safeguards for lower precision inputs; (2) unfamiliarity or incompatibility with MTIA's Triton dialect, which restricts some features available on GPU versions. Approximately 80% of failed operators still produce code that compiles and passes at least some tests, and 30% pass over 80% of their test suite.

We further heuristically divide the operators into 7 categories depending on the intended functionality of each operator. Table 1 shows that different categories present different difficulties to TritorX: while TritorX achieves 96.0% coverage on Shape Manipulation operators, the coverage significantly drops for operators from the "Deep Learning Category." This is somewhat expected, as the operators consist of computational design patterns that do not always map perfectly to Triton semantics.

Finally, we executed a run with GPT-OSS on a future generation using a QEMU simulator (Bellard, 2005) for execution feedback. This single run yielded a coverage of 73.1%. We aggregated the compiler failures and feature gaps encountered during generation, and shared this data with our compiler and ASIC engineers.

4.1 End-to-end Application

From our baseline set of OpInfo operators, we used TritorX to enable various first- and third-party models on MTIA. While robust, we recognize that production workflows may contain operators and operator arguments (eg. shapes, scalar values, etc.) outside of the distribution represented in the OpInfo testsuite. To mitigate this, we decompose various target models into their individual operators, extract all operator inputs observed during eager-mode inference on the GPU, and then use these inputs within the TritorX validation loop instead of those generated by OpInfo.

Concretely, we instrumented forward and backward passes of several representative models, NanoGPT (Karpathy, 2023), DLRM (Naumov et al., 2019), and two internal recommendation models (denoted MM), using `__torch_dispatch__` to intercept and record the tensor and scalar data passed to each operator. All four models were evaluated with a fixed batch size of 1024 and trained

Op Category	Op Count	Operator Coverage (%) CWM	GPT-OSS	Sonnet	Opus
Elementwise	161	80.1	84.6	84.0	85.9
Deep Learning	90	64.4	71.1	71.4	76.2
Linear Algebra	78	71.8	79.5	78.2	78.2
Other	78	75.6	74.3	76.9	75.6
Shape Manipulation	75	96.0	96.0	94.7	96.0
Reduction	63	69.8	74.6	76.2	71.4
Indexing & Selection	34	73.5	79.4	91.2	91.2

Table 1. TritorX Coverage by operator category and LLM model name.

for single iteration, with the latter three (DLRM, MM1, and MM2) executed using real production data rather than randomized inputs.

Additionally, we introduce a new step to TritorX, first matching a given operator with a pre-generated OpInfo operator (should it exist), then immediately testing it with the inputs gathered from the full e2e run. Should the kernel not pass all tests out of the box, it is used as a starting point from which TritorX then refines (Table 2, column B: MIS).

Model	Operator Coverage (%) A. Full Model Op Set	B. OpInfo Subset *OpInfo*	*MIS*
NGPT	87.2	80.0	100.0
DLRM	81.4	80.0	90.0
Meta M1	79.8	83.8	91.9
Meta M2	80.6	81.7	87.3

Table 2. Operator coverage across four model types: Nano-GPT, Deep Learning Recommendation Model, "Meta Model 1," and "Meta Model 2." An operator is considered "covered" if the corresponding kernel passes all tests with model input shapes (MIS). (A) We run TritorX on all model operators with the MIS feedback. (B) We run TritorX on a subset of model operators that have tests available in the OpInfo suite. *(OpInfo)* We directly test kernels created with OpInfo feedback with MIS. *(MIS)* We run TritorX to refine kernels created with OpInfo feedback with MIS feedback.

Across these experiments, TritorX achieves high kernel coverage, enabling nearly 80% of all kernels required to execute a model end-to-end. Furthermore, for operators where a pre-existing OpInfo-validated kernel is available, over 80% of these kernels pass all end-to-end production tests without additional prompting. After refinement, TritorX further improves this by an additional 6–20% across models. This not only underscores the robustness of TritorX, but also establishes a sandbox for continuous testing and optimization of production-ready kernels.

While kernel performance is not the primary focus, we measure TritorX-generated kernels against hand-written kernels for NanoGPT to gain some visibility into performance metrics. Out of 300+ (kernel, unique tensor shape) pairs, the majority of generated kernels achieve at least 70% of handwritten kernel performance. Only a small percentage outperform our handwritten ones since we did not integrate any performance-targeted optimizations into the harness.

4.2 TritorX Harness Ablation

To better understand which aspects of TritorX contribute to its success, we ablate over various configurations. Table 3 summarizes the results of these experiments.

Method	Operator Coverage (%) CWM	GPT-OSS	Sonnet	Opus
Baseline (single run)	55.3	72.0	72.2	78.7
w/o linter	35.7	46.7	51.2	71.3
w/o summarization	48.2	71.5	69.4	71.8

Table 3. Ablations over TritorX harness features. Number of iterations per run are kept constant.

We examine the importance of the custom Triton MTIA linter and the optional summarization model. Removing the linter resulted in a significant drop in performance ($55.3\% \rightarrow 35.7\%$ for CWM, $72.0\% \rightarrow 46.7\%$ for GPT OSS). As mentioned previously, the linter not only helps the agent identify intrinsics unique to the Triton MTIA dialect, but also helps prevent "cheating" by flagging the unauthorized use of other torch operators (Appendix D).

Removing the summarization agent also resulted in a decrease in performance for CWM ($55.3\% \rightarrow 48.2\%$). Without a separate summarization agent, the entire compilation log, which can consist of thousands of tokens, is fed directly into the LLM dialog session, and the model performance can degrade as we approach the context limit (Hsieh et al., 2024). However, we did not see a similar decrease in performance for GPT OSS when we removed the summarization agent.

5 Related Work

Custom ASICs, MTIA & Deep Learning Compilers. Since Triton introduced a Python-first DSL for high-performance GPU kernels (Tillet et al., 2019a) and became

widely used by PyTorch Inductor (Ansel et al., 2024) to generate fused operators, several ecosystems now offer Python-level kernel DSLs. NVIDIA's Warp (Macklin, 2022) is a Python DSL for authoring CUDA kernels, with an optional tile-based programming model for tensor-core GEMMs. In JAX, Google's Pallas (Bradbury et al., 2018) allows users to write custom kernels in Python; it targets GPUs via Triton and TPUs (Jouppi et al., 2017) via Mosaic (Bansal et al., 2023), enabling fine-grained fused operations within the JAX ecosystem. Meta has also adopted Triton for MTIA to improve developer efficiency with PyTorch.

Kernel Research. Optimization of custom kernels for specific devices has received sustained attention in recent years: (Lavin & Gray, 2016; Dao et al., 2022; Dao, 2023; Shah et al., 2024). This interest is both driven by and enabling the rapid growth of training and inference compute of frontier models: (Cottier & Rahman, 2024; Hooker, 2021).

Benchmarks. The community has responded by releasing benchmarks to measure the functional correctness and performance of LLM-generated kernels that primarily target GPUs through CUDA and Triton. KERNELBENCH (Ouyang et al., 2025) and TRITONBENCH (Li et al., 2025a) evaluate whether models can generate correct and performant GPU kernels across representative operator suites, while NPU-EVAL (Kalade et al., 2025) targets AMD NPUs. ALGOTUNE (Press et al., 2025) in particular targets LLM's ability to speed up scientific computing problems from natural language descriptions. Automatically generated kernels are prone to exploiting holes in the test suite. Addressing this has seen significant effort: (Lange et al., 2025b; METR, 2025). Notably, similar to our work BACKENDBENCH (Saroufim et al., 2025) also uses PyTorch's OpInfo as a comprehensive test suite to ensure correctness and comprehensiveness.

Training-Time Approaches. Most LLM-based kernel generation relies on prompting techniques combined with test-time compute use as a form of search, and recent systems propose additional training for further refinement. KERNELLLM (Fisches et al., 2025) provides a supervised baseline for the generation of Triton kernels from PyTorch modules. AUTOTRITON (Li et al., 2025b) adds reinforcement learning from verifiable rewards. Multi-turn reinforcement learning for CUDA kernel generation has also been explored in (Baronio et al., 2025). CUDA-L1 (Li et al., 2026) further optimizes performance with contrastive reinforcement learning.

Inference-Time Refinement. Orthogonally, several approaches scale inference compute with existing LLMs, covering a spectrum of prompting, agentic, and evolutionary techniques combined with verification through unit tests: (Lange et al., 2025b; METR, 2025; Chen et al., 2025a; Wei et al., 2025; Wang et al., 2025). While several approaches exist that target CUDA and GPUs, we believe only NPU-Eval (Kalade et al., 2025) is targeting non-GPU devices with C++. TritorX similarly operates at inference time, using multi-turn feedback loops to correct errors, but leverages LLDB debugging and custom linters in addition to compiler feedback.

6 DISCUSSION AND FUTURE WORK

TritorX's FSM framework provides a robust harness for generating functionally correct Triton MTIA kernels. Thanks to its flexibility, we identify several additional directions to further improve coverage and performance and reflect on what aspects are most important for success.

Self-consistent operator generation. To prevent "cheating," the linter restricts the agent from utilizing other ATen operators within the wrapper (beyond tensor allocation). A more efficient and possibly more performant approach is to allow the wrapper to dispatch other operators, provided they are also implemented in the new backend and the operators do not result in cyclic dependencies. This requires a self-consistent generation scheme where the agent is aware of the entire backend state (and is thus no longer embarrassingly parallel).

Optimized prompting. As mentioned earlier, we found that simpler prompts without dedicated MTIA documentation worked best, but there remains room for prompt tuning. Furthermore, we can improve the quality of the example kernel-wrapper pairs themselves using strategies like localization, perhaps even bootstrapping the process over sequential runs.

Fully agentic pipeline. At a higher level than these improvements, we have discussed making the entire FSM agentic – converting most current states into tools for the LLM to call upon, as well as adding new tools related to debugging, such as giving the LLM a sandbox to execute code in.

Dedicated model post-training. Both models used throughout this work were open source and off-the-shelf. All of the MTIA-specific context was gained via interactions with the linter, compiler, and debugger. An orthogonal research direction would be to further post-train the LLM for Triton MTIA kernel generation and contrast that with the existing results for vanilla Triton in the literature.

The importance of scale. Due to the stochastic nature of the underlying language model, running the benchmark repeatedly (with a nonzero temperature) will produce results with nonidentical pass-rates. Thus, simply aggregating the passing operators across runs, a technique known as test-time scaling, can yield significant increases in operator coverage. For example, just aggregating between two benchmark runs using CWM increased the coverage from $55\% \rightarrow 64\%$. We suspect that further scaling this and exploring more complex

strategies such as evolutionary scaling (Lange et al., 2025b) will bring significant coverage improvements. Furthermore, supporting additional hardware generations, DSLs, and operator definitions will require a flexible and scalable pipeline for generating operator sets.

Establishing a backend-maintenance environment. As AI-driven kernel generation evolves, AI will eventually produce more kernels than humans can feasibly review—especially as hardware diversity increases and multiple generations of hardware coexist within production fleets. Establishing a robust, fully automated framework that does not require human review will be essential for the success of AI-generated kernels. This need will become even more critical as each model update triggers a comprehensive refresh of the kernel library to optimize performance.

Test suite rigor. We have discussed the benefits of the OpInfo test suite above, but we acknowledge that the presence of such a robust, high-quality test suite is a limitation to broad generalization. To address both this limitation and the impossibility of covering the entire sample space, we plan to explore input generation techniques based off *operator specifications*, which we hope is more generalizable than the PyTorch- and ATen-specific OpInfo.

Performance tuning. Integrating performance tuning into TritorX is an important next step. This will require incorporating additional data sources into the LLM, including mechanisms for capturing architecture-specific patterns and antipatterns, validating and profiling performance, and ensuring that optimizations do not introduce correctness regressions. We will also prioritize performance work by developing filtering strategies to identify which production-grade kernels most need optimization.

7 Conclusion

We introduced TritorX, a scalable, coverage-first system that automates bring-up of the PyTorch backend for custom ML ASICs, demonstrated on Meta's MTIA. TritorX orchestrates an existing LLM with a finite-state workflow and production-compatible tooling, executed on both deployed silicon and a QEMU-based simulator for future devices. It generated over **481** ATen operators that pass all of their corresponding OpInfo tests (**20,000+** total tests), achieving an overall pass rate of **84.0%**. From these operators, we were able to onboard **80+%** of the multiple first- and second-party large-scale models on the device. Ablations over model and system factors isolate which components most influence coverage, turning backend enablement into a measured engineering process rather than an artisanal effort.

Beyond MTIA, the design is applicable to other accelerators and naturally extends to adjacent tasks such as cross-generation kernel migration and early hardware/compiler feedback via simulation. Although performance tuning is out of the scope of this work, TritorX provides the substrate on which autotuning, schedule search, and learned code optimization can be layered. Future directions include expanding operator families (reductions, sparse/quantized ops), stronger safety/containment, and formal checks for code generation, tighter integration with compiler IRs, and automated pathways for performance refinement.

Taken together, these results offer a practical blueprint for democratizing toolchain creation and a force multiplier for kernel engineers—freeing expert effort to focus on truly performance-critical paths while bringing entire backends online in hours rather than months.

8 Acknowledgments

We thank the PyTorch Team, in particular the authors of BackendBench for their support. We acknowledge that our work stands on the shoulders of the entire MTIA team and are grateful for the enablement of our research. We also thank Dipal Saluja for his early feedback.

References

Andrews, M. and Witteveen, S. Gpu kernel scientist: An llm-driven framework for iterative kernel optimization. *arXiv preprint arXiv:2506.20807*, 2025.

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., 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*, pp. 929–947, 2024.

Anthropic. Claude opus 4.5 system card. Technical report, Anthropic, November 2025a. URL https://www.anthropic.com/system-cards.

Anthropic. Claude sonnet 4.5 system card. Technical report, Anthropic, September 2025b. URL https://www.anthropic.com/system-cards.

Bansal, M., Hsu, O., Olukotun, K., and Kjolstad, F. Mosaic: An interoperable compiler for tensor algebra. *Proceedings of the ACM on Programming Languages*, 7(PLDI): 394–419, 2023.

Baronio, C., Marsella, P., Pan, B., and Alberti, S. Kevin: Multi-turn rl for generating cuda kernels. *arXiv preprint arXiv:2507.11948*, 2025. URL https://arxiv.org/abs/2507.11948.

Bellard, F. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*

(ATEC '05), pp. 41, Berkeley, CA, USA, April 2005. USENIX Association. URL https://dl.acm.org/doi/10.5555/1247360.1247401.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Chen, T., Xu, B., and Devleker, K. Automating gpu kernel generation with deepseek-r1 and inference-time scaling. NVIDIA Technical Blog, 2 2025a. URL https://developer.nvidia.com/blog/automating-gpu-kernel-generation-with-deepseek-r1-and-inference-time-scaling/.

Chen, W., Zhu, J., Fan, Q., Ma, Y., and Zou, A. CUDA-LLM: Llms can write efficient cuda kernels. *arXiv preprint arXiv:2506.09092*, 2025b.

Coburn, J., Tang, C., Asal, S. A., Agrawal, N., Chinta, R., Dixit, H., Dodds, B., Dwarakapuram, S., Firoozshahian, A., Gao, C., 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*, pp. 1689–1702, 2025.

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

Cottier, B. and Rahman, R. The training compute of notable ai models has been doubling roughly every six months. Epoch AI Data Insight, 2024. URL https://epoch.ai/data-insights/cost-trend-large-scale. Accessed 2025-10-21.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023. URL https://arxiv.org/abs/2307.08691.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems (NeurIPS 2022)*, 2022. URL https://openreview.net/forum?id=H4DqfPSibmx. 36th Conference on Neural Information Processing Systems. (Peer-reviewed).

Fisches, Z. V., Paliskara, S., Guo, S., Zhang, A., Spisak, J., Cummins, C., Leather, H., Synnaeve, G., Isaacson, J., Markosyan, A., and Saroufim, M. Kernelllm: Making kernel development more accessible, 6 2025. URL https://huggingface.co/facebook/KernelLLM.

Hooker, S. The hardware lottery. *Communications of the ACM*, 64(12):58–65, 2021.

Hsieh, C.-P., Sun, S., Kriman, S., Acharya, S., Rekesh, D., Jia, F., Zhang, Y., and Ginsburg, B. Ruler: What's the real context size of your long-context language models?, 2024. URL https://arxiv.org/abs/2404.06654.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

Kahn, J. D., Pratap, V., Likhomanenko, T., Xu, Q., Hannun, A., Cai, J., Tomasello, P., Lee, A., Grave, E., Avidov, G., et al. Flashlight: Enabling innovation in tools for machine learning. In *International Conference on Machine Learning*, pp. 10557–10574. PMLR, 2022.

Kalade, S. et al. NPUEval: Optimizing npu kernels with llms and open source compilers. *arXiv preprint arXiv:2507.14403*, 2025. URL https://arxiv.org/abs/2507.14403.

Karpathy, A. nanogpt: The simplest, fastest repository for training/finetuning medium-sized gpts, 2023. URL https://github.com/karpathy/nanoGPT. GitHub repository.

Lange, R. T., Sun, Q., Prasad, A., Faldor, M., Tang, Y., and Ha, D. The ai cuda engineer: Agentic cuda kernel discovery, optimization and composition. arXiv preprint arXiv:2509.14279 and project archive, 2025a.

Lange, R. T., Sun, Q., Prasad, A., Faldor, M., Tang, Y., and Ha, D. Towards robust agentic cuda kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025b. URL https://pub.sakana.ai/static/paper.pdf.

Lavin, A. and Gray, S. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4013–4021, 2016.

Li, J., Li, S., Gao, Z., Shi, Q., Li, Y., Wang, Z., Huang, J., Wang, H., Wang, J., Han, X., Liu, Z., and Sun, M. Tritonbench: Benchmarking large language model capabilities for generating triton operators. *arXiv preprint arXiv:2502.14752*, 2025a. URL https://arxiv.org/abs/2502.14752.

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

Li, X., Sun, X., Wang, A., Li, J., and Shum, C. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. 2025c. doi: 10.48550/arXiv.2507.14111. URL https://arxiv.org/abs/2507.14111.

Li, X., Sun, X., Wang, A., Li, J., and Shum, C. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning, 2026. URL https://arxiv.org/abs/2507.14111.

Macklin, M. Warp: A high-performance python framework for gpu simulation and graphics. https://github.com/nvidia/warp, March 2022. NVIDIA GPU Technology Conference (GTC).

METR. Measuring automated kernel engineering. Blog post, 2 2025. URL https://metr.org/blog/2025-02-14-measuring-automated-kernel-engineering/.

Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

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

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. 2017.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

Press, O., Amos, B., Zhao, H., Wu, Y., Ainsworth, S. K., Krupke, D., Kidger, P., Sajed, T., Stellato, B., Park, J., et al. Algotune: Can language models speed up general-purpose numerical programs? *arXiv preprint arXiv:2507.15887*, 2025.

Saroufim, M., Wang, J., Maher, B., Paliskara, S., Wang, L., Sefati, S., and Candales, M. Backendbench: An evaluation suite for testing how well llms and humans can write pytorch backends, 2025. URL https://github.com/meta-pytorch/BackendBench.

Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024. URL https://arxiv.org/abs/2407.08608.

Shehabi, A., Hubbard, A., Newkirk, A., Lei, N., Siddik, M. A. B., Holecek, B., Koomey, J., Masanet, E., Sartor, D., et al. 2024 united states data center energy usage report. 2024.

Silvano, C., Ielmini, D., Ferrandi, F., Fiorin, L., Curzel, S., Benini, L., Conti, F., Garofalo, A., Zambelli, C., Calore, E., et al. A survey on deep learning hardware accelerators for heterogeneous hpc platforms. *ACM Computing Surveys*, 57(11):1–39, 2025.

Tang, C., Yu, K., Veeraraghavan, K., Kaldor, J., Michelson, S., Kooburat, T., Anbudurai, A., Clark, M., Gogia, K., Cheng, L., et al. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 787–803, 2020.

Tillet, P., Kung, H.-T., and Cox, D. 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 (MAPL@PLDI)*, pp. 10–19, 2019a. doi: 10.1145/3315508.3329973. URL https://dl.acm.org/doi/10.1145/3315508.3329973.

Tillet, P., Kung, H.-T., and Cox, D. 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*, pp. 10–19, 2019b.

Wang, J., Joshi, V., Majumder, S., Chao, X., Ding, B., Liu, Z., Brahma, P. P., Li, D., Liu, Z., and Barsoum, E. Geak: Introducing triton kernel ai agent & evaluation benchmarks. 2025. doi: 10.48550/arXiv.2507.23194. URL https://arxiv.org/abs/2507.23194.

Wei, A., Sun, T., Seenichamy, Y., Song, H., Ouyang, A., Mirhoseini, A., Wang, K., and Aiken, A. Astra: A multi-agent system for gpu kernel performance optimization. 2025. doi: 10.48550/arXiv.2509.07506. URL https://arxiv.org/abs/2509.07506.