MTIA-RT: Enabling a Native PyTorch Experience on Custom AI ASICs

Hangchen Yu
Meta
Menlo Park, USA
hcyu@meta.com

Adam Belay
Meta
Menlo Park, USA
abelay@meta.com

Jun Luo
Meta
Menlo Park, USA
junluo@meta.com

Chunqiang Tang
Meta
Menlo Park, USA
tang@meta.com

Roman Levenstein
Meta
Menlo Park, USA
rlevenstein@meta.com

Yige Hu
Meta
Menlo Park, USA
yige@meta.com

Anirudh Shaktawat
Meta
Menlo Park, USA
anirudh1@meta.com

Alex Mastro
Meta
Menlo Park, USA
amastro@meta.com

David Reiss
Meta
Menlo Park, USA
dreiss@meta.com

Tim Sham
Meta
Menlo Park, USA
tsham@meta.com

Jaesoo Lee
Meta
Menlo Park, USA
jaesoo@meta.com

Anirban Banerjee
Meta
Menlo Park, USA
anirbanb@meta.com

Hyunho Yeo
Meta
Menlo Park, USA
hyunho@meta.com

Youlin Zhang
Meta
Menlo Park, USA
ylzhang@meta.com

Wushi Dong
Meta
Menlo Park, USA
dongws@meta.com

Rumit Desai
Meta
Menlo Park, USA
rumitdesai@meta.com

Wang Xu
Meta
Menlo Park, USA
wangxu@meta.com

Jack Montgomery
Meta
Menlo Park, USA
jackmontgomery@meta.com

Kaustubh Gondkar
Meta
Menlo Park, USA
kgondkar@meta.com

Ajit Mathews
Meta
Menlo Park, USA
amath@meta.com

***Abstract*—The evolution of AI workloads has driven the need for specialized accelerators that provide both computational efficiency and easy-to-use software interfaces. This paper presents the host runtime (MTIA-RT) for Meta's custom AI accelerator (MTIA), which manages kernel launches, buffer lifecycles, and other critical functions. To our knowledge, MTIA-RT is the first publicly documented runtime for an AI accelerator deployed at hyperscale. Despite MTIA's different architecture compared to GPUs, MTIA-RT exposes APIs that are functionally compatible with the CUDA runtime, ensuring application portability. It provides a familiar, native PyTorch experience for developers, supporting both high-performance graph mode and flexible eager mode. Currently, it manages accelerators in production and serves a diverse set of models. By sharing MTIA-RT's design and deployment experience, we aim to provide concrete, field-tested knowledge about production accelerator runtime development.**

***Index Terms*—Runtime, ASIC, Accelerator, Eager Mode, PyTorch, MTIA.**

I. Introduction

Large language models, recommendation systems, and other emerging ML use cases are driving a surge in demand for AI accelerators, such as GPUs from NVIDIA and AMD, as well as custom ASICs developed by hyperscalers and numerous AI chip startups [1]–[11]. These chips are managed by an accelerator runtime that runs on the host and coordinates kernel launches, buffer lifecycles, and other critical functions on the accelerator. Performance is crucial, as even a small delay in the runtime could leave accelerator resources idle.

Despite the proliferation of GPUs and custom ASICs along with the numerous publications on their hardware architectures, programming models, and compilers, no work has detailed the internals of an accelerator runtime. This represents a notable gap, given the runtime's important role and the growing imperative to develop and optimize runtimes for many new types of accelerators. For example, NVIDIA's CUDA runtime is the most widely known today. It provides programming abstractions that hide hardware complexity and maintain portability across GPU generations. However, despite public documentation of CUDA's runtime API, its underlying design and implementation remain proprietary, preventing knowledge sharing with the broader community. Moreover, CUDA supports only NVIDIA GPUs, so it is unclear if its API can generalize to other types of accelerators.

To address the limited understanding of accelerator runtimes, we share our experiences developing and deploying *MTIA-RT*, the runtime for Meta's AI accelerators, MTIA [8]–[10], [12]. MTIA-RT currently manages accelerators in production at scale, achieving higher performance per unit of total cost of ownership (TCO) than comparable GPUs. To the best of our knowledge, this paper is the first to publicly document the internals of a production runtime for an AI accelerator.

One challenge in developing MTIA-RT was compatibility. MTIA has a different architecture from NVIDIA GPUs—for example, it does not provide hardware threading (SIMT [13]) and lacks virtual memory support. Despite these differences,

we found it was possible to provide a developer API that was functionally compatible with CUDA. For example, MTIA-RT uses the CUDA stream abstraction to allow programmers to control on-device concurrency. This dramatically simplified integration with existing ML frameworks like PyTorch directly or via middle layers (like OpenXLA PJRT [14]), as well as custom internal workloads at Meta. We anticipate other custom AI accelerators will face similar challenges because their architectures also differ from NVIDIA GPUs, yet many may still want to maximize compatibility and portability with the PyTorch and CUDA ecosystem. Therefore, we hope these experiences will be helpful to the broader AI-accelerator community.

Another challenge for MTIA-RT was performance. Like CUDA, MTIA-RT provides support for eager-mode execution where operators run immediately as they are encountered in code. This mode is native to PyTorch, simplifies model development, and is easier to debug. As a result, it is the most common mode for training jobs at Meta, especially for experimentation when creating new models. However, because eager mode relies on many fine-grained kernel launches, its performance is constrained by runtime overhead. To minimize this overhead, we co-designed the runtime with the accelerator firmware. This includes a runtime threading model that avoids context switching, the use of PCIe BAR writes for efficient small data transfers to the device, and pipelined kernel launches from the device firmware. When even higher performance is required, MTIA-RT also provides support for CUDA graphs, which allows batches of operations to be launched together [15]. Finally, MTIA-RT can go a step further than CUDA graphs by providing a custom monolithic graph mode that combines and launches multiple operators in a single optimized kernel that is fused by the compiler. This is useful in situations where the utmost performance is needed, such as large-scale deployments of inference workloads.

Contributions. In summary, we make the following contributions:

- Despite the runtime's important role and the increasing need to develop and optimize runtimes for rapidly emerging GPUs and accelerators, no prior work has detailed a runtime's internals. We address this gap by presenting the first end-to-end description of such a runtime.
- We demonstrate that even for custom ASICs with hardware architectures different from GPUs, it is feasible to maintain sufficient runtime compatibility with CUDA's API, easing portability and backward compatibility for existing software. Achieving this is more challenging than for AMD GPUs, which have a more similar architecture to NVIDIA GPUs [16].
- While most custom ASICs avoid eager mode due to its performance overhead [4], [7], [17]–[19], we show that an optimized runtime can enable eager mode without sacrificing performance.
- Beyond eager mode, we demonstrate that a runtime for custom ASICs can deliver a familiar, native PyTorch experience while maintaining high performance.

II. Background

In this section, we provide an overview of the CUDA programming model; describe design characteristics and compatibility with CUDA in existing runtimes; and summarize the similarities and differences between GPUs and other accelerators like MTIA [8]–[10]. We conclude by discussing runtime integration with ML frameworks, and design tradeoffs between eager mode and graph mode.

A. The CUDA Programming Model

CUDA [20] is a runtime system and parallel programming model that targets NVIDIA GPUs. It provides abstractions for issuing parallel operations while providing portability across different generations of hardware. It is widely adopted in industry, and serves as a de facto standard for many ML applications.

CUDA allows programmers to define functions that can execute on GPUs, called *kernels*. Unlike regular functions, kernels are designed to execute in parallel on many different hardware threads. These threads are clustered into groups, called thread blocks, that each run the same code. Each thread can rely on its thread index to help determine which subset of the input data it will process.

A memory hierarchy defines how data can be accessed and shared. Each thread has its own local registers and private memory, and data can be shared efficiently within a thread block or across all threads through slower but higher capacity global device memory. The host can also do direct memory access (DMA) to move memory in and out of the GPU to its own buffers. All these memory operations, i.e., allocating global device and host memory and transfers between them, are provided by CUDA APIs.

Programmers interact with CUDA through a *context*, which represents an isolated environment on a GPU. Within each context, concurrency is managed through an abstraction called a *stream*. Each stream executes operations (e.g., kernel launches, memory copies, etc.) asynchronously and in the order they were issued. Multiple streams can be used at a time and they can synchronize with each other by waiting for events (i.e., on-stream markers that signal command completions).

B. Existing Runtimes for AI Accelerators

Several runtimes have been developed for emerging AI accelerators, requiring different levels of effort to migrate applications to them from CUDA. Most do not target direct CUDA compatibility; instead, they provide interoperability at the framework level (e.g., PyTorch [21], TensorFlow [22]). Even with full framework support, developers may still need substantial effort to support new models. For example, Cerebras [7] and SambaNova [19] require changes to data preprocessing and model architectures to fit specialized dataflow schedules. AMD, by contrast, offers partial CUDA compatibility through HIPIFY [23], which automatically translates many CUDA APIs to AMD HIP equivalents but still requires manual

intervention when no direct mapping exists. Unlike MTIA-RT, AMD's underlying hardware is based on a GPU architecture.

Another difference in existing runtimes is that they are generally optimized for graph mode rather than eager mode. While this improves performance for large-scale production deployments, it makes debugging harder because individual operator states are less accessible. Graph-centric runtimes [24]–[28] take models in a graph IR (e.g., PyTorch FX [29], [30], XLA HLO [31], ONNX MLIR [32], [33]), compile them into hardware-specific executables, and execute them as optimized, mostly static computation graphs. This approach avoids the overhead of many fine-grained kernel launches and enables global optimizations such as quantization, layout transformation, tiling, cache access, operator fusion, and even distributed scheduling (e.g., TPU Pathways [28]). As a result, these runtimes can rely on simpler host interactions that often require less optimization.

C. MTIA Accelerator Hardware

Like many custom AI ASICS, MTIA lacks some GPU architectural features such as hardware threading (SIMT [13]) and virtual memory. However, our experiences with MTIA suggest that it is still feasible to retain CUDA's abstractions at a runtime level.

In this paper, we focus on Meta's recent MTIA 300 chip as a motivating example (hardware layout shown in Fig. 1) [10], but MTIA-RT can provide the same abstractions for earlier generations of MTIA hardware [8], [9]. MTIA 300 uses a SIMD execution model and features a 12×6 grid of programmable processing elements (PEs), each with 512 KB of local memory, two vector units, and several matrix multiplication units (Dot Product Engine, and Reduction Engine). MTIA 300's memory hierarchy is simpler and more manually managed than a GPU's: it has 192 MB of on-chip SRAM and 144/216 GB off-chip HBM capacity (144 GB is picked for production usage), shared across all PEs. This architecture results in some unique design challenges for the runtime layer: (1) MTIA-RT has to dynamically transform input shapes to meet cache alignment requirements for best performance. (2) MTIA 300 has no virtual address space, so multi-tenancy and memory management are more difficult. (3) Both DRAM and SRAM are visible to the programmer, instead of connecting them through a hardware cache hierarchy like in a GPU. (4) The PE grid is configurable to run different jobs on rectangular subgrids, but it is runtime-controlled and lacks hardware threading, shifting more responsibility to MTIA-RT.

D. ML Frameworks

PyTorch is among the most widely adopted AI frameworks at Meta [21], [29]. It supports a variety of accelerators through a consistent, user-facing Python frontend. It allows the registration of per-hardware backends, which hook into model compilation, tensor management, and neural network execution, and compose these operations using the runtime API of the target hardware. Both NVIDIA and AMD provide

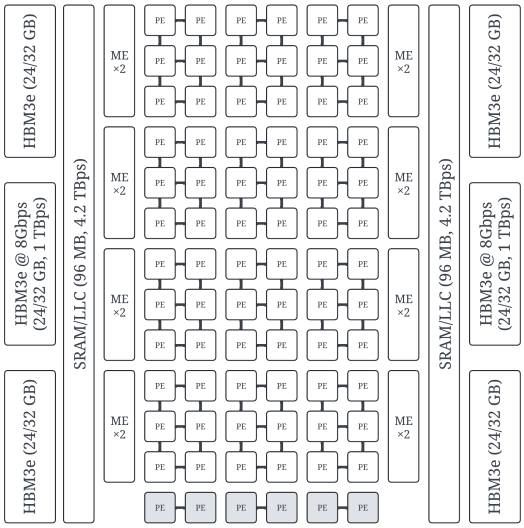


Fig. 1: Simplified MTIA 300 architecture [10].

an additional layer, `torch.cuda`, which wraps CUDA's device, context, memory, and stream APIs into Python, enabling direct device control with a programming experience close to that of the CUDA C/C++ API. Many AI models and applications (e.g., GPT2 [34], Diffusion [35]) are implemented using the unified PyTorch frontend. However, more complex AI frameworks, such as vLLM [36], extensively leverage `torch.cuda` for optimized memory management and job scheduling. Those workloads are thus more difficult to migrate to new accelerators if they cannot provide a `torch.cuda` equivalent.

E. Eager and Graph Mode Tradeoffs

Deep learning execution spans a spectrum from maximal flexibility (debuggability) to maximal performance. Pure eager mode, the default define-by-run approach in PyTorch [21], executes operators immediately, making debugging and dynamic control flow easy, but it incurs frequent kernel launch and Python overhead and limits global optimization [37]. Inductor graph mode [29] balances usability and performance by capturing Python programs into graphs, enabling fusion, and lowering launch overhead of fusion groups with CUDA graph [38]. Monolithic graph mode treats the entire model as a single optimization unit [39]–[41] to expose more cross-operator opportunities such as memory planning and end-to-end scheduling, but it is harder to compile and can be brittle when model structure or input is highly dynamic. Megakernel approaches push optimization closest to hardware by collapsing a large portion of the model into one or a few persistent kernels while sacrificing development simplicity [42]–[44].

The graph-centric nature of most existing accelerators renders PyTorch eager mode inefficient, if not entirely unsupported. Irregular data layouts, memory accesses and branching in eager or fine-grained dynamic execution hurt SIMD performance [45]–[47] and expose extra CPU-side data and control-flow overhead, as well as frequent host-device synchroniza-

tions. Most accelerators (e.g., Inferentia [48], Trainium [4], and Gaudi [27]) support only static graphs; TPUs [49]–[51] (with TensorFlow [22] and JAX [31]) simulate eager mode by buffering and lazily evaluating operators as a graph [18], [52]. According to Meta-internal MTIA adoption surveys, the lack of eager mode support is a significant concern, as it leads to a poorer development and debugging experience. We show that competitive graph and eager mode performance are still achievable on MTIA/customized ASICs with an efficient runtime that optimizes its interactions with device firmware.

We evaluate MTIA monolithic graph mode and GPU graph mode (inductor with CUDA graph) in Section VII, as these represent the most common inference production workloads. We choose to evaluate eager mode as well, because it is more sensitive to kernel launch overhead and useful in three common settings. First, developers rely on eager mode for experimentation when creating new models in order to verify model accuracy; they do not want compiler techniques (PT2 compilation) to alter the model's execution behavior and complicate the inspection of buffers passed between kernels. Second, PyTorch supports and Meta uses other compilation modes in which graph breaks commonly occur (e.g., Torch Inductor), and these can easily require hundreds of eager-mode kernel launches in practice. Third, eager mode dominates training production: 78.4% of the more than 100K distinct training jobs running on NVIDIA GPUs at Meta use eager mode today, and we expect many jobs to run on MTIA in the future.

III. Runtime Overview

A. Overview

MTIA-RT sits in between applications and the firmware running on the accelerator. It provides CUDA-like APIs for device, memory, and compute job management. Its design follows three principles:

1) **Generality:** It must work well across a wide range of production workloads and MTIA silicon generations.
2) **Interchangeability:** The API signatures and semantics should be structurally similar to CUDA APIs in order to reduce porting effort.
3) **Performance:** Kernel launch latency should be as low as possible and the device should be kept busy as often as possible. This principle is especially important for supporting eager mode efficiently.

MTIA-RT has supported four different generations of SIMD hardware: MTIA 100 [8], 200 [9], 300 [10], and the forthcoming MTIA 400, 450, 500 [12] under the generality principle. MTIA-RT's hardware-agnostic methods—including its API designs, its bypassing of drivers to reduce communication overhead between runtime and firmware, and its fast paths for specific job types—have proven broadly applicable across every hardware platform we support. At the same time, MTIA-RT incorporates hardware-specific optimizations to accommodate differences in hardware specifications, such as cache-line alignment, in order to fully exploit each platform's potential. For instance, because kernels typically reach peak efficiency only when their input buffers are properly aligned, co-design between the runtime and the kernels is essential to optimizing performance.

Fig. 2 shows the major components connected by MTIA-RT. A PyTorch model can run on MTIA seamlessly using the PyTorch MTIA extensions (see Listing 2). The model is first ① traced, transformed, and optimized into one or more FX IRs. Each IR is then converted to an ELF binary separately by an MTIA-specific LLVM compiler, with ② MTIA kernel symbols linked. The compiled ELF binaries are then ③ serialized along with other model data and metadata, such as weights and static shapes, and provided to the graph and kernel launchers—④ accessed through MTIA-RT API wrappers—to be scheduled for on-device execution. MTIA-RT ⑤ communicates with a user-space driver to configure devices and initialize hardware resources. It is also capable of ⑥ communicating directly with the device control firmware running on a control processor (Section V-A) to minimize host-device communication overhead. The device firmware, which runs on a control core on the accelerator, finally sets up the execution environments for the jobs and ⑦ initiates the PEs to perform the computations. Additionally, MTIA-RT integrates with the MTIA collectives library and driver, which are beyond the scope of this paper.

B. Public APIs

The MTIA-RT APIs are designed to be similar to CUDA runtime APIs [53] to unlock easier application migration from GPUs to MTIA. Some APIs are slightly modified to be compatible with MTIA hardware differences and other parts of the MTIA software stack: for example, MTIA exposes SRAM as a user-accessible resource, and MTIA's current code compiler does not yet support compiling mixed host and accelerator code nor does it recognize the triple angle bracket notation (i.e. <<< ... >>>) for kernel launches (more details in subsection V-C).

Listing 1 shows a simple MTIA program, where we omit error handling (all APIs return an error code), DMA data preparation, and result verification. It demonstrates several differences from CUDA runtime APIs:

- All MTIA-RT APIs support lazy device initialization, but users can disable it and initialize and destroy the context with explicit `mtiaInit` and `mtiaDestroy` calls.
- Both DRAM and SRAM are visible and accessible to users.
- `mtiaExecutableLoad` is simplified in Listing 1. In the real world, it requires reading more detailed ELF metadata such as enabled performance counters from disk or memory.

```
1 mtiaError_t err = mtiaSetDevice(0);
2
3 mtiaStream_t stream = 0;
4 err = mtiaStreamCreate(&stream);
5
6 void* hostBuf = nullptr, deviceBuf = nullptr;
7 err = mtiaHostAlloc(&hostBuf, 96);
```

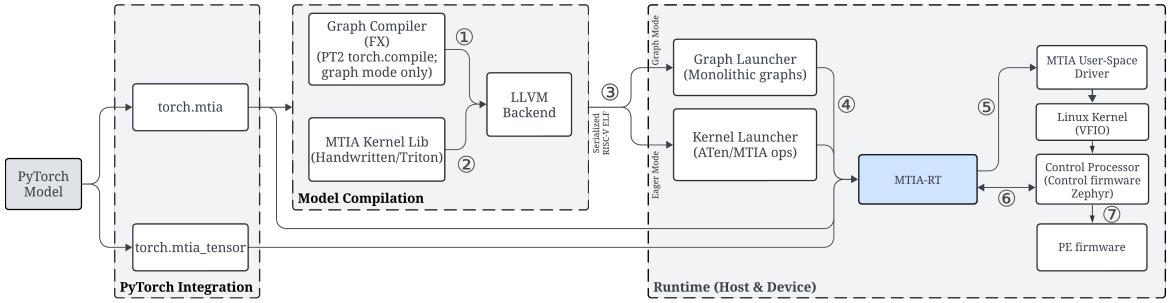


Fig. 2: Overview of MTIA-RT and its primary connected components for serving a PyTorch model end-to-end.

```
*(float*)hostBuf = 1.0;
*(float*)(hostBuf+32) = 2.0;
err = mtiaMalloc(&deviceBuf, 96, MTIA_MEM_DRAM);
err = mtiaMemcpyAsync(deviceBuf, hostBuf, 64,
    MTIA_MEMCPY_HOST_TO_DEVICE, stream);

mtiaExecutable_t exe;
err = mtiaExecutableLoad(&exe, "/tmp/add_fp32.elf");

mtiaLaunchConfig_t config = {
  .tileWidth = 8, .tileHeight = 8, .timeoutMs = 5000
};
// Execute add kernel with 2 input buffers and 1
    output.
err = mtiaRunFunction(config, exe,
  {deviceBuf, deviceBuf+32, deviceBuf+64}, stream);

err = mtiaMemcpyAsync(hostBuf+64, deviceBuf+64, 32,
    MTIA_MEMCPY_DEVICE_TO_HOST, stream);
err = mtiaStreamSynchronize(stream);
CHECK_EQ(*(float*)(hostBuf+64), 3.0);

err = mtiaExecutableUnload(exe);
err = mtiaFree(deviceBuf);
err = mtiaHostFree(hostBuf);
err = mtiaStreamDestroy(stream);
```

Listing 1: MTIA code example that runs an add operator a + b using the MTIA-RT APIs.

IV. Runtime Modules

A. Memory Management

MTIA 300 is equipped with both HBM and SRAM. SRAM is scarcer but provides higher performance. It is partitioned into a last-level cache (LLC) and a last-level scratch (LLS). The LLC is used by both the on-device control and compute firmware, while the LLS is visible to the host runtime and ML kernels. The LLC vs. LLS ratio is configurable at device initialization, allowing engineers to optimize for different model sizes and workload characteristics. For example, allocating more SRAM to LLC benefits cache-heavy workloads, whereas increasing LLS improves workloads that require static allocation such as weights and activations. MTIA-RT supports allocating and freeing DRAM and LLS buffers.

MTIA-RT supports host-device DMA, as well as inter- and intra-device memory copies. Inter-device transfers are enabled by (1) mapping each device's whole memory address space onto the PCIe bus, so that the DMA engine on the destination device can directly pull data from the source; or by (2) collectives over the NIC when two devices are on different nodes. MTIA hardware does not support intra-device memory copies directly; it is instead implemented by issuing an eager-mode operator to perform the copy from the PEs. For smaller transfers, MTIA-RT also supports direct PCIe BAR writes to improve buffer copy latency compared to DMA (V-D).

B. Executable Management

MTIA-RT registers and executes RISC-V [54] ELF binaries on the device. An ELF object file is the minimal unit of execution on MTIA which can contain one single kernel in eager mode or multiple kernels in graph mode. This is different from CUDA which runs a kernel as a function inside a fat binary [55] that can contain multiple PTX virtual assembly code sections that can each be loaded and run independently. Instead, MTIA runs each kernel as a separate executable (Section V-C). ELFs are loaded from disk or just-in-time generated at run time. Each ELF is assigned a unique ID for use by the kernel launching APIs.

Developers can choose to store ELFs in SRAM or DRAM: the MTIA firmware can load instructions and data more efficiently from SRAM, but SRAM is a scarcer resource than DRAM. The MTIA firmware itself is a small footprint OS image designed for resource constrained real-time systems and uses a single address space for executables (ELFs), the OS kernel and firmware. Therefore, ELFs are position-independent binaries, and MTIA-RT performs a two-phase compilation to offload multiple ELFs within the same address space: (a) It compiles each kernel or graph with `.TEXT` section specified to be at `0x0`, and allocates a device memory of the same size as the compiled ELF binary; (b) It relocates all symbols in the compiled ELF based on the allocated device memory address, and writes the relocated ELF to that memory buffer.

C. PE Job Submission

A PE job submission is launched through a special command with several configuration values and device buffer addresses sent from MTIA-RT to the on-device firmware. The device firmware queues the submitted PE jobs per stream (Section IV-D) until a PE subgrid of the requested size becomes available; it then sets up the kernel's execution context (e.g. caches, registers) and dispatches the job to that allocated PE subgrid. However, only in debug mode would we expect jobs

to be submitted and executed sequentially—required so that the program can stop and dump the precise state after a failure. Otherwise, there is generally a queue of ready jobs that can start to be submitted while the grid is still busy. Therefore, PE idleness can be avoided through a pipelined job dispatch optimization, where the firmware begins to set up the next job while the previous job is still executing to reduce the delay between launches.

For a PE job, there are three kinds of buffers attached to the submission as shown in Table I. MTIA-RT manages these buffers' lifetimes (V-G4) to prevent data corruption among concurrent jobs.

We extracted common configuration data needed for PE jobs into a data structure called `LaunchConfig`, which would fit into a single 64-byte cache line. The `LaunchConfig` contains information needed for a job submission, including PE subgrid size and position, PMP ranges, etc. It can be reused across PE jobs in the same stream to save the time for setting it up and moving it to the device for every job.

D. Stream Management

Streams are the abstraction of on-device parallelism which supports concurrent job submission and synchronization. The minimal unit of job submission is a `StreamCommand`, which represents a job being submitted from the host runtime to the device runtime. Here are five of the most commonly used commands (not including the collectives):

- Transfer memory: Copies source memory to a destination address on the same or a different device.
- Run PE job: Executes a selected RISC-V ELF binary on a PE subgrid with the given argument data.
- Run host function: Executes a host function on CPU with the given argument data.
- Record event: Records (inserts) an event into the stream.
- Insert cross-stream dependency: Adds a dependency for an event on another stream.

MTIA-RT's stream implementation buffers commands on the device, with firmware enforcing FIFO (first-in-first-out) order handling per stream and concurrency across streams. It maximizes throughput by saving host-device communication overhead for cross-stream synchronization and pipelining (1) command preparation on the host, (2) command transfer over PCIe, and (3) command scheduling on the device.

Stream synchronization is a host-side operation and does not impact the commands that have already been submitted to the device. MTIA-RT will block the caller thread that attempts to synchronize a stream via `mtiaStreamSynchronize`, until all previously enqueued commands are completed. The other threads can still enqueue commands to that stream in parallel, which will be submitted to the device without being interrupted by the stream synchronization.

Host functions require special handling compared to other commands. When submitting one to a stream, all subsequent enqueued commands to that stream will be buffered on the host until the host function completes its execution. A host function can also be blocking, which requires synchronizing the stream first, or non-blocking with an asynchronous callback, which requires the host function to run when MTIA-RT detects that the previous enqueued command has completed.

V. Runtime Design Case Study

A. User-space Driver and Driver Bypass

MTIA-RT communicates with a user-space device driver via D-Bus [56] IPC, which controls the device via MMIO regions and VFIO (Virtual Function I/O) [57]. However, the IPC of communicating over D-Bus introduced extra overhead that was significantly higher than the original kernel-driver based mechanism, which exposed operations through IOCTLs and other system calls. As shown in Fig. 3, we therefore added an extremely low-latency, high-throughput bypass mechanism by mapping firmware command queues and other memory resources directly into the client process's address space, similar to kernel-bypass network drivers [58]–[60]. This allowed MTIA-RT to submit commands and read completions without going through the driver, eliminating D-Bus IPC and user-kernel context switching overhead.

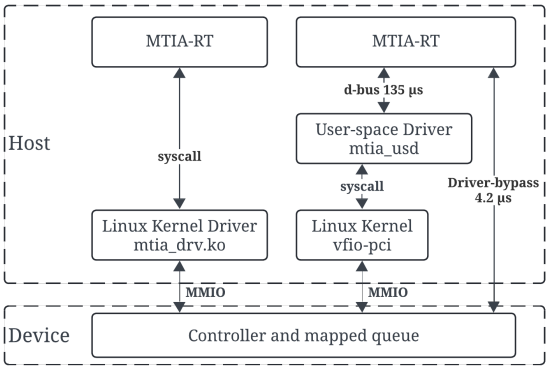


Fig. 3: Driver bypass with user-space driver. This reduced the latency of host-device communication round trip to 4.2 µs. A roundtrip through `mtia_drv.ko` took 13 µs.

B. Host-managed Device Memory

Table II compares the average latency of allocating device memory on NVIDIA H200 and MTIA. CUDA memory allocation requires coordination with the device (*e.g.,* to update virtual mappings), and its cost can increase manyfold when memory gets fragmented. Instead, MTIA-RT is responsible for managing the entire DRAM and LLS memory regions, and it reserves them during runtime initialization. All subsequent allocation requests are handled by the host, eliminating the need for communication with the device driver or firmware. Developers are provided with both DRAM and SRAM regions with flexible memory allocation policies. For example, one can invoke the `mtiaMalloc` API with the `SRAM_FALLBACK_DRAM` policy to designate that the DRAM region is permitted as a fallback when available SRAM is insufficient.

Type	Creator	Visibility	Usage
Request buffers	User program via MTIA-RT APIs	Host runtime, Kernels	Prediction request data buffers such as inputs, outputs, weights.
Firmware buffers	Host runtime	Host runtime, Firmware	Storage of firmware traces and coredump.
Special buffers	Host runtime	Host runtime, Kernels	Index tables for buffer shapes and addresses (Section V-C); storage of kernel logs, traces and coredump.

TABLE I: Categories of associated buffers to a PE job submission.

Size	cudaMalloc	mtiaMalloc
1 KB	1.60 µs	0.48 µs
1 MB	53.95 µs	1.65 µs
10 MB	69.06 µs	11.85 µs
100 MB	106.76 µs	47.23 µs
1 GB	29 849.82 µs	51.22 µs

TABLE II: Average memory allocation latency on NVIDIA H200 with CUDA 12.4 v.s. on MTIA with MTIA-RT.

C. Mitigation of Missing Single-Source Compiler

Unlike the NVCC (CUDA compiler driver) and gpucc [61] (an open-source CUDA compatible compiler), which are single-source compilers that handle compilation of GPU device functions and host code in one CUDA source and afterwards embed the compiled GPU functions as fat binary images within the host object file, the MTIA compiler does not support compiling a mixture of device functions and host code. Instead, the MTIA compiler compiles each kernel (in eager mode) or each graph (in graph mode) into a separate program binary, rather than as a function object. As a result, the MTIA kernels must be implemented and optimized for generic sizes of input tensors, and the kernel binary contains no references to data type, structure, or size information that is available in the host code. MTIA-RT passes this information via two data buffers to the device, allowing kernels to parse it in real time.

The *lookup buffer* hosts an auxiliary lookup table to index the request buffers and special buffers, and helps kernels to look up those buffers' device addresses according to their indexes or tags. The lookup buffer is protocolized with a custom C struct buffer that is inspired from Flatbuffers [62] to enforce forward and backward compatibility between MTIA-RT and kernels as well as high serialization and deserialization performance.

The *TensorView buffer* is a list of `TensorView` data structures including the corresponding tensor's shape, stride, data type, etc. for the kernel to correctly interpret tensor storage. In graph mode, the TensorView information is generated at the graph compilation and hard-coded into the compiled ELF. In eager mode, TensorViews, one per kernel argument, are generated by a shape inference procedure and serialized into the TensorView buffer on the fly.

The lookup and TensorView buffers are kept separate to optimize cases where only one buffer needs to be updated—for example, when the model has a static shape or buffers are reused across kernel executions. Both sizes scale proportionally with the number of buffers associated with the kernel or graph, which can become a performance bottleneck for job submissions (Section V-D).

D. Memory Allocation and Movement

Memory allocations—including both host and device—and data movement between host and device were identified as the primary bottlenecks affecting end-to-end latency and as major contributors to performance deviation. Previously, MTIA-RT required an average of 16.4 µs (ranging from 11 to 34 µs) at the `mtiaRunFunction` layer and 18.2 µs (ranging from 12.2 to 38 µs) at the PyTorch layer to launch a simple add operator. The optimizations described in this subsection effectively reduce both latency and deviation. Currently the same operator takes 4.2 µs (ranging from 2.9 to 6.0 µs) at `mtiaRunFunction` layer and 5.8 µs (ranging from 4.2 to 8.8 µs) at the PyTorch layer.

1) Fast Memory Allocations

Heap and device memory allocation can become a bottleneck for critical paths, such as kernel launching. To optimize this, MTIA-RT relies on pre-created, fixed-length buffers of device memory for runtime job metadata (as detailed in Section V-C). The runtime also minimizes its use of C++ container classes (e.g. `std::string`), replaces most `std::vector` objects with small vectors [63], and allocates thread-local data from an arena allocator [64]. Collectively these reduce heap allocations on the host.

2) Small Data Transfer

When data transfers are small (e.g., less than 16 KB), we found that it was more efficient to write the data to the PCIe BAR directly rather than asking the accelerator to transfer the data via DMA. Fig. 4 compares the latency between BAR writes and DMAs. In practice, more than 95% of PE jobs have less than 12 arguments so the lookup table and TensorView list can easily fit in a 16 KB buffer, allowing them to take advantage of this optimization. We also experimented with optimizing small-transfer DMA performance but decided not to pursue it because BAR writes will always be faster in the sizes we care about.

E. Threading Model

MTIA-RT uses a threading model that is designed to minimize context switches. In an earlier version of MTIA-RT, we used a pair of internal threads for each stream to handle sending commands on the submission queue (SQ) and receiving completions from the completion queue (CQ). We thought that having a separate completion thread would reduce completions handling delays, which are needed for running host functions and propagating cross-device events

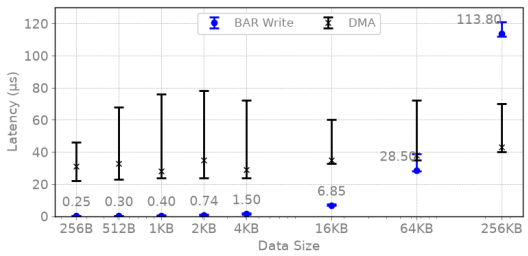


Fig. 4: PCIe BAR write v.s. DMA write latency by size. The BAR write is faster for small sizes, while its latency grows linearly but faster when the size is over 16 KB.

(both can't be delayed). However, we discovered that this design led to frequent context switches (2–3 per kernel launch, each adding a couple microseconds), which contributed to significant runtime overhead relative to the runtime's overall kernel launch cost of 2.9–6 µs.

To avoid this overhead, we switched to a design that never context switches between threads, and instead uses the thread the application is already running on to write commands directly to a stream's SQ. Completion handling from the CQ is deferred until either the stream becomes full (*i.e.,* the number of commands in flight reaches the queue size) or the the stream is synchronized via `mtiaSynchronize`, `mtiaStreamSynchronize` or `mtiaStreamWaitEvent`. In either case, completions can be reaped in-line without thread switches to maintain higher efficiency.

One challenge with this design is that all completion handling is delayed, but as mentioned above it still needs to be fast for host functions and cross-device events. To overcome this limitation, the firmware provides MTIA-RT with a separate per-stream queue that is only used for events (EQ). A dedicated internal thread continuously performs blocking reads on the EQ so that it can promptly handle these rare events out-of-band without requiring synchronization with streams. The firmware also uses the EQ to deliver notifications for other event types, such as when it detects device-level failures.

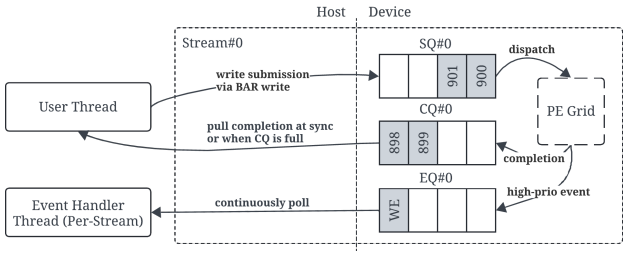


Fig. 5: Thread mode of MTIA streams: User threads write directly to the on-device submission queue (SQ) and from completion queue (CQ), while a per-stream event handler background thread continuously polls the high-priority event queue (EQ). User threads can concurrently access multiple streams.

F. Cross-Stream Synchronization

Meta recommendation system models are partitioned into several networks to balance and maximize memory and compute resources on both CPUs and accelerators. Cross-network dependency is required, for example, the dense network execution (which performs prediction) must wait for sparse network completion (which looks up embeddings) running on a different stream.

Similar to CUDA, we use `mtiaEventRecord` to mark a point in a stream execution and `mtiaStreamWaitEvent` to specify cross-stream synchronization and enforce the execution ordering correctness *on the device* to reduce round trips between host and device. As shown in Fig. 6, the MTIA stream specifies each event as a ⟨stream id, sequence number⟩ pair, and sends a special `WaitOnEvent` firmware command (in blue) with the target event to the device without stalling the submission on the host. After firmware receives that command, it checks all dependent streams running past the dependent sequence numbers (e.g., Ⓐ 901 on Stream#0_0) before submitting any new command (e.g., Command#601 on Stream#0_1). However, streams still rely on host notifications to unblock firmware queues that are waiting on events from other devices.

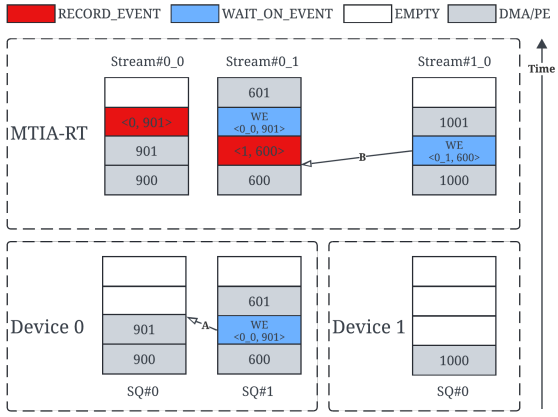


Fig. 6: Intra- and inter-device event handling with MTIA streams. Stream#X_Y means the X-th stream created and mapped to the submission queue X on Device Y. Ⓐ The intra-device dependency is enforced on the device; Ⓑ the inter-device one is done on the host and blocks Job#1001 to be submitted to the firmware.

G. Special Use Cases in Meta Production

Numerous designs and optimizations are based on special production requirements at Meta. This section will cover several representatives.

1) Production Deployment

Meta maintains all code in a monolithic repository and builds production packages with all dependent libraries statically linked on top of the tree. However, product groups have different release pipelines and cadences of their production packages, which would lead to multiple versions of the MTIA-RT libraries in the production. The MTIA software stack and

application integrations are also undergoing fast iterations, therefore we are not able to follow Meta's strategy for CUDA deployment strategy which is to update CUDA versions across all platforms quarterly.

We choose to define pure C public APIs for MTIA-RT for ABI compatibility and support dynamic linkage by default for both production and development environments. The MTIA team centralizes and manages the release of the MTIA-RT shared library and MTIA user-space driver (Section V-A) independently from product groups' pipelines. MTIA-RT and driver are bundled to prevent incompatibility between the host and device runtimes. We version the MTIA-RT APIs with $N-1$ backward compatibility guarantee, so that the workloads using old API versions of MTIA-RT will not be impacted when the MTIA-RT library is upgraded: a workload using MTIA-RT version $N-1$ can switch to version N in its next release. Deprecated APIs are removed only when no production workload is using the APIs.

2) Memory Identifiers

MTIA utilizes Physical Memory Protection (PMP) [65], a standard RISC-V feature, to control the memory access permissions for specified physical memory regions, compensating for the lack of MMU-backed isolated virtual memory address spaces. MTIA-RT maintains a lookup table to track allocation sizes by their start addresses. For each PE job, MTIA-RT configures PMP to ensure that memory outside the allocated ranges or all user memory remains inaccessible. However, unlike NVIDIA GPUs, MTIA 300 does not support unified virtual addressing (UVA) [66] to distinguish addresses from different devices. Instead, MTIA-RT distinguishes memory from different devices by including the device ordinal in the top bits of the device address, creating a unique ID (UID). This UID is returned to the `mtiaMalloc` caller and can be masked to retrieve the allocation's start address on the device.

3) Multi-Tenancy

The host-side memory region management (Section V-B), single device address space design (Section IV-B) and lack of hardware resource partitioning limit MTIA's multi-tenancy support. Nevertheless, we find two primary use cases of multi-tenancy in the MTIA production:

- Small CPU models may not utilize full hardware resources. We support static partitioning in MTIA-RT: each instance pre-allocates $\frac{1}{N}$ of memory regions and limits kernels to execute on a predetermined $\frac{1}{N}$ of the 8×8 PE grid, where $N = 1, 2, 4$ is the tenancy count.
- MTIA model trainers are ported from GPU which open all visible devices at start, but only one trainer will allocate memory or execute kernel jobs on each device. We observe that trainers' first invoked CUDA/MTIA API is `cudaSetDevice(device_id)` or `mtiaSetDevice(device_id)`. Therefore, we introduce a special mode to `mtiaSetDevice` to lazily initialize MTIA-RT by claiming resources of only the specified device.

4) Buffer Lifetime Tracking

MTIA-RT uses reference counters to manage the lifetime of device buffer allocations. The reference counters are incremented before launching DMA, P2P, and PE jobs that access them and decremented by `mtiaFree` and when the associated jobs complete.

5) Priority Stream

The model partitioning scheme described in Section V-F introduces another challenge for optimizing service QPS and device utilization. In a typical two-device setup, the sparse network is split into two shards, each assigned to a device, while the dense network is replicated across both devices. On each device, a configurable number of sparse and dense worker threads are spawned. For each incoming request, both devices asynchronously process their assigned sparse shard (usually taking several milliseconds). Afterward, one device (selected per request in a round-robin fashion) runs the dense replicate to generate the final results (usually 20–50 milliseconds). This architecture can lead to short, continuously submitted sparse jobs queuing up in the on-device submission queues and being scheduled faster than the longer-running dense jobs. Consequently, early-arriving requests may be blocked by sparse executions from later requests, negatively impacting end-to-end latency.

MTIA-RT improved average end-to-end latency by 3–5% by allowing users to create streams with priorities and assigning a higher priority to the Fully Connected (FC) worker streams: high priority jobs have higher probability to be scheduled before low priority jobs. However, low priority streams are at risk of experiencing starvation. It is the application's responsibility to properly assign the priority levels to the streams to avoid starvation—if there is a potential starvation on a low-priority stream, the application should increase the priority level of that stream.

VI. Native PyTorch Integration

We followed the standard PyTorch integration procedure to ensure native compatibility with PyTorch for MTIA devices.

A. MTIA ATen Backend and torch.mtia

The PyTorch ATen library provides unified interfaces across device types for tensor operators through its C++ frontend. The PyTorch dispatcher dispatches these ATen operators to their respective device-specific implementations based on the input tensor's device type.

The MTIA ATen backend implements all customized ATen operators and connects them to corresponding MTIA kernels, as shown in Fig. 7. It supports fallback to CPU operators for any operator being debugged or not yet available on MTIA, enabling rapid iteration during early model enablement. We provide options of in-tree operator registration that reuses PyTorch's native codegen (TorchGen [67]), meta functions, automatic type casting, etc., and out-of-tree registration that completely decouples MTIA from the PyTorch source tree for experimental features.

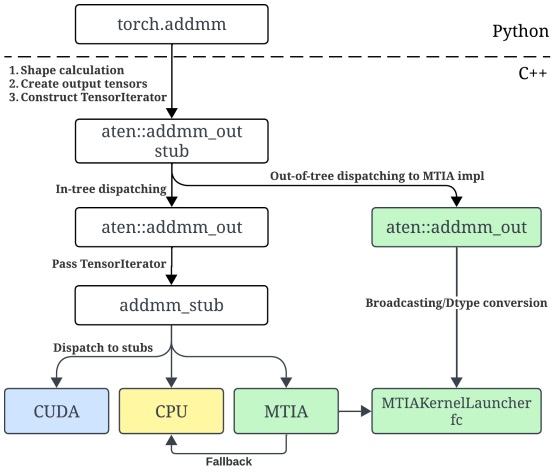


Fig. 7: In-tree and out-of-tree dispatch flow for torch.addmm. More details of TensorIterator, DispatchStub, and MTIAKernelLauncher can be found in Section VI-A. Green blocks are MTIA-specific.

To enable applications' direct access to device, memory, stream and event, we introduced the *torch.mtia* module [68] via PyTorch virtual guard and Hooks interface which separates those user-facing management interfaces and implementations. This provides an interface identical to torch.cuda, allowing seamless migration between CUDA and MTIA-RT and automatic device backend selection with torch.accelerator [69] which is another layer of abstraction on top of torch.mtia and torch.cuda.

MTIA reuses most ATen device-agnostic interfaces to prevent unnecessary branching with low maintenance.

- *MTIA tensor* is implemented by overriding the memory allocator in the `c10::Storage` creation, enabling native PyTorch tensor APIs such as `torch.empty(..., device="mtia")`.
- *Pinned memory tensor* is device-agnostic, based on the Accelerator Hooks interfaces (PyTorch PR#126376 [70]).
- *Out-of-place operators* and tensor operations (e.g., creation and resizing) are generated with a special MTIA version of `aten::empty_tensor` based on TorchGen [67] and Structured Kernels [71].
- We directly use *TensorIterator* to handle element-wise operations on tensors efficiently with features like automatic dtype casting and broadcasting and integrated with Structured Kernels.
- Some operators are dispatched to dedicated functions per underlying hardware via *DispatchStub*. We create new stubs and dispatch functions for MTIA if they do not already exist.
- The community-contributed *PyTorch test cases* were supported by modifying the device type instantiation process and reusing the rest of the implementation.

B. PyTorch Example and MTIA Migration

PyTorch is a complex ecosystem that provides many other domain libraries besides the core functionality, e.g., FBGEMM [72], TorchRec [73], and FSDP [74] (Fully Sharded Data Parallel). Those domain libraries and PyTorch-powered applications can be easily extended to support MTIA following a straightforward strategy: replacing all hard-coded device specific APIs with the (1) MTIA or (2) device-agnostic ones. Listing 2 demonstrates these two ways, while TorchRec PR#1382 [75] is a realistic example of migrating existing PyTorch recommendation systems to MTIA.

```
# Method 1: Using torch.mtia API
a = torch.ones([8, 8], dtype=torch.float16, device="
    mtia")
b = test_model(a)
c = b.to("cpu")
torch.mtia.synchronize() # Fully propogate data

# Method 2: Using torch.acclerator API
device_type=torch.accelerator.current_accelerator()
a = torch.ones([8, 8], dtype=torch.float16, device=
    device_type)
b = test_model(a)
c = b.to("cpu")
torch.accelerator.synchronize()
```

Listing 2: Example PyTorch code running on MTIA.

Table III evaluates the migration of five major Meta internal ML frameworks in engineering time and lines of code change. Most took a handful of engineer days to port core functionality, while it generally takes 1 engineer week to support complex optimizations due to the hardware differences and more weeks to fill gaps in the MTIA compiler and kernel to run full production models.

C. Graph Mode

MTIA's execution model centers on monolithic graph mode in production, which reduces host launch overhead and enables global memory planning across the entire traced graph. The program is first captured as an FX GraphModule via `torch.export` or `torch.compile`, and is then compiled into a monolithic ELF for execution. The resulting artifacts are packaged into the Graph Launcher (Fig. 2), which manages input processing (e.g., padding and type conversion), allocates intermediate tensor memory, and invokes MTIA-RT to launch the graph ELF as the minimal unit on device. In typical workloads, runtime launch overhead is negligible relative to input processing time, and host overhead is fully overlapped with PE execution; consequently, performance is normally bounded by memory bandwidth or PE compute throughput.

Dynamic shapes are a core challenge for monolithic execution. The Graph Launcher reserves intermediate buffers sized for the maximum supported shape, runs a host-side shape inference library prior to dispatch, and provides kernels with per-request shape information in the same manner as eager mode (see Section V-C).

Application	Engineering Effort	SLOC
vLLM	<2 weeks (incl. basic model enablement) for e2e examples; ≈2 weeks to implement optimizations and enable Llama4.	1K LOC in total for Llama4 with advanced optimizations like KV caching.
Llama4x Trainer	<1 day to simply replace torch.cuda.	≈160 LOC across 23 files.
General PyTorch Trainer	<1 day to port with runtime optimizations.	3 LOC in 1 file to load torch.mtia when MTIA is detected. 10-30 LOC per feature to support pipeline parallelism, overlapped communications, etc.
Python Predictor	Worked out of the box. 2 weeks to onboard the first model.	0 LOC to run minimal examples on MTIA. ≈600 LOC per model type pertaining to data processing (e.g., text tokenization for transformer-based models) and lowering.
C++ Predictor	≈2 days to implement MTIA task runner following exactly the same CUDA task runner logic.	≈150 LOC for MTIA task runner, and 200 LOC to integrate with predictor.

TABLE III: Migration cost of major Meta internal ML applications from CUDA to MTIA. The C++ predictor is written with CUDA C API. Engineering Effort is measured for a single developer.

D. Eager Mode

While MTIA and the majority of software stack are designed and optimized for graph mode to maximize performance and cost efficiency, MTIA-RT aims to deliver a close-to-GPU user experience and performance when developers try to use eager mode.

1) Graph Compiler Bypass

Existing accelerator runtimes support eager mode by reusing the graph-mode compiler and constructing graphs of either a single layer or lazily buffered layers. MTIA-RT generates IR for each operator call and directly dispatches it to the MTIA compiler backend for compilation and caching, entirely bypassing the graph compiler's tracing, optimization, and static lowering steps.

2) Runtime Layout Transformations

Unlike the MTIA-2i [9] hardware which limits kernels to access only 32-byte-aligned buffers to maintain cache coherency, MTIA 300 increases cache line to 64 bytes and lifts the limitation. However, accessing unaligned data dramatically increases kernel overhead from cache invalidation. As a workaround, MTIA-RT continued to pad the lowest dimension of all tensors to 64 bytes. Padding was originally performed on the host, requiring it to check tensor layouts before each kernel launch. If tensors were unaligned, the host had to copy them from the device, pad them, and copy them back, forcing stream synchronization and reducing host-device parallelism. This degraded end-to-end performance.

Today MTIA-RT moves padding to the device by creating padded empty tensors to reserve space and executing a *layout_transform* kernel for unaligned tensors. Eager mode end-to-end throughput was less than 10% of graph mode for both training and inference, and eliminating host-side layout transforms leads to a 2–4× boost. Notably, the layout_transform kernel is not fully optimized and is memory bound. For example, executing `aten::concat` with three FP32 2792×1024 aligned tensors takes 6.5 µs end-to-end. But it takes as long as 8 ms for unaligned inputs with on-host transformation, and it still takes 44 µs even when the layout transformation is moved to the device as well.

3) DMA and PE Job Batching with MTIA Graph

Most workloads involve the launch of hundreds or thousands of kernels per iteration, e.g., Llama 4 Scout and Maverick [76] comprise 48 transformer layers, with each layer initiating ≈10 PE jobs. To reduce and amortize the cost of host-side launch overhead, NVIDIA offers CUDA graph [15] to enable compute, memory, and collective job submissions. Similarly, MTIA-RT introduces a feature, termed *MTIA graph*. The MTIA graph execution consists of three distinct phases:

1) **Capture**: A MTIA stream switches to capture mode when `mtiaStreamBeginCapture` is called. Any DMA or PE operations are logged in a graph with all input/output parameters instead of being executed right away on the device.
2) **Instantiate**: `mtiaGraphInstantiate` instantiates a recorded graph to be executable by allocating and caching all necessary host and device resources required for execution, which includes device buffers, launch configurations, commands, etc. and constitutes the majority of the overhead in an eager mode execution.
3) **Launch**: `mtiaGraphLaunch` submits the prepared commands in the graph in a lightweight way.

MTIA graph currently has two limitations: (1) It relies on host-side node dispatch—where a lightweight job execution command is issued per graph node—rather than fully on the device as introduced in CUDA 12.6 [77]. This preserves flexibility of supporting dynamic shapes (which require on-demand shape inference) and conditional nodes, but incurs per-node host launch overhead. (2) While MTIA graph reduces host time, it does not yet improve end-to-end kernel latency because PE job fetch and initialization are serialized with execution in the firmware. Enabling pipelining of PE job fetch, initialization and execution reduces inter-kernel idle time by more than 3 µs (from ≈5 µs, compared to ≈0.9 µs on GPUs [15]).

Table Va compares the latency of eagerly launching `aten::add` operators with and without MTIA graph. MTIA-RT takes 4.4 µs host time to launch this minimal ATen operator, compared to 3 µs on the NVIDIA H200 with the same host setup. MTIA graph further reduces this overhead by a factor of 7–13.6×, amortizing the launch latency to 0.4 µs

per operator.

VII. End-to-end Performance Evaluation

We evaluate performance of two primary MTIA 300 workload types—recommendation training and GenAI inference—and compare against NVIDIA H100 and H200 with CUDA 12.4. MTIA 300 peak FLOPS is 50% of H200 with 25% higher HBM bandwidth [10]. Compared with H100, its peak FLOPS is 72% and HBM bandwidth is 2.5×. Because each MTIA 300 host contains a single card, we focus on single-node performance when possible to remove the effects of cross-host collective submission and execution. The models are shrunk to fit within the single MTIA 300 node. We also briefly discuss multi-node training and inference performance.

A. Recommendation Training

Table IV collects the throughput for three recommendation models trained on multi-node MTIA 300 and GPU H100[1]. Each H100 host contains 8 H100 cards, whereas each MTIA 300 host has a single MTIA 300 card, resulting in slightly higher network overhead for distributed workloads. Every trainer was assigned to its own card; for example, MAIv4 utilized 40 trainers, which were distributed across either 5 H100 hosts or 40 MTIA 300 hosts.

Although collective submission is not yet fully optimized in MTIA-RT and MTIA graph is not enabled for these workloads, MTIA-RT prevents the host from becoming the bottleneck. As a result, MTIA monolithic graph mode achieves 30% higher throughput than GPU Inductor graph mode, and MTIA eager mode reaches 84%–99% of GPU graph-mode throughput. On MTIA 300, eager mode achieves 75% of MTIA graph-mode throughput. Across all models, performance is effectively limited by HBM bandwidth.

Model	Trainer	G-Graph	M-Graph	M-Eager
IDXCVR	16	254K	334K (1.31×)	252K (99.2%)
MAIv4	40	388K	498K (1.28×)	376K (96.9%)
E2V	256	466K	n/a†	392K (84.1%)

TABLE IV: End-to-end IDXCVR (Index CVR), MAIv4 (Mobile App Install) and E2V (Event-to-Vec) model QPS with GPU Inductor graph mode (G-Graph), MTIA monolithic graph mode (M-Graph), and MTIA eager mode (M-Eager). † has not been enabled on MTIA due to the architecture divergence between MTIA and GPU and the development inefficiency from graph-mode model lowering.

B. GenAI Inference

1) Single-Node Performance

Fig. 8 reports single-node DeepSeek-R1 performance on MTIA 300 for monolithic graph mode and eager mode, and compares against H200 running Inductor graph mode (with CUDA graph) and eager mode. To fit within a single MTIA 300 or H200 card, the model was shrunk from the full-size DeepSeek-R1-671B.

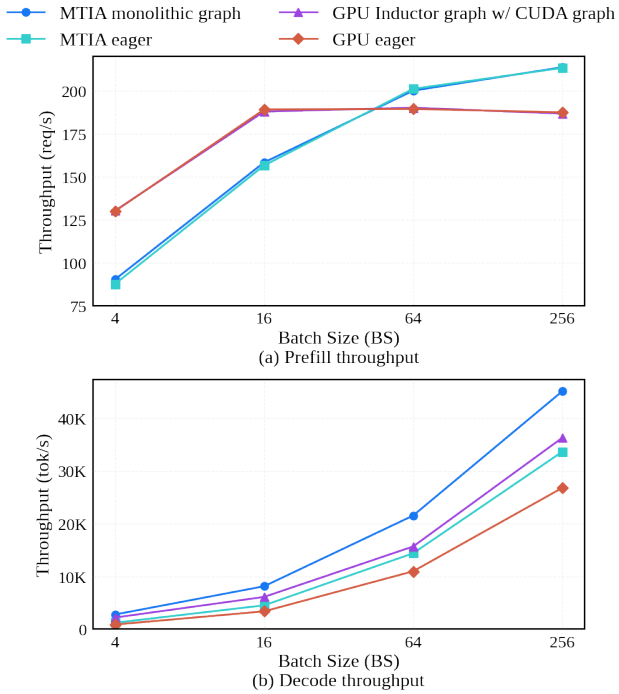


Fig. 8: Comparison of DeepSeek-R1-shrunk [78] throughput with different batch sizes and execution modes. Input sequence length is 1024.

At small batch sizes, prefill is compute bound, so GPU throughput is higher until memory becomes the bottleneck (the speedup is less than 2× because the kernels are optimized to different degrees). Peak prefill throughput is higher on MTIA 300 owing to its higher memory bandwidth. Although eager mode introduces additional host and device overhead, it does not noticeably affect throughput on either MTIA 300 or H200, thanks to the efficient MTIA-RT and CUDA runtime.

Decode is memory bound. Both MTIA-RT and CUDA runtime allow scaling before peak FLOPS/s is reached. MTIA monolithic graph mode outperforms or reaches parity with GPU in all scenarios. MTIA eager mode generally achieves 50–75% of the end-to-end throughput of monolithic graph mode, and 70–130% of eager mode on H200, even without special kernel optimizations or the use of MTIA graph. The performance ratios of eager mode versus graph mode on MTIA is close to that observed with CUDA (40%–80%) when device is not saturated. For reference, the DeepSeek-R1-shrunk decode throughput on MTIA 300 are 54% and 64% of that in monolithic graph mode at batch size 16 and 64, comparing to CUDA's 52% and 64%. Similar ratios are observed with GPT-OSS and Llama3.2-1B models. There are some remaining sources of overhead that vary across workloads, as they have different data types, shapes, and operators. For

[1]These models are yet to be provisioned on H200 and too large to fit within one node.

example, additional layout_transform calls have overhead—as mentioned in Section VI-D2—as well as kernels and collective communications that we are continuing to optimize.

2) Zoom-In the Trace

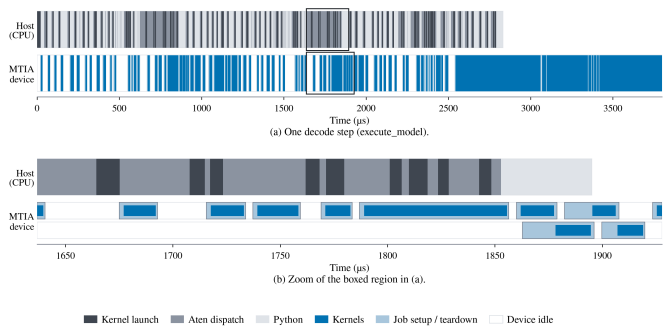


Fig. 9: Zoomed-in trace of DeepSeek-R1-shrunk [78] decode step on MTIA 300, running in eager mode.

Fig. 9 zooms into the trace of DeepSeek-R1-shrunk decode phase on MTIA 300, running in eager mode. The profiler introduces ≈20% host overhead to the PyTorch runtime and MTIA-RT. The primary sources of PyTorch dispatch overhead (shown in grey) are the CPU-side logic between kernel invocations such as output tensor allocations. Each decode step issues 87 kernel launches, with a median inter-kernel gap of 27 µs. The median kernel launch latency is 15 µs, most of which is spent on preparing and transferring the lookup and TensorView buffers, as discussed in Section V-C. Both numbers include the 20% tracing overhead. Fig. 9(b) extracts a snippet of the trace and shows an example of device job pipelining, which reduces PE job bubbles to 2–3 µs by overlapping jobs' setup with the current kernels' execution. Some bubbles last longer because the job queue starves when host-side operations, such as dynamic memory allocation, stall.

3) MTIA Graph

Table Vb compares host operator launch latency in PyTorch with GenAI model launch latency in vLLM, including Llama3.2-1B [79] and a shrunk version of DeepSeek-R1 [78] (1 dense layer and 1 MoE layer with 256 experts). MTIA graph maximizes device utilization by eliminating host-side bottlenecks from end-to-end execution. For instance, as shown in Fig. 8, Llama3.2-1B (BS=16) eager-mode decode takes 6.1 ms/token after translation. MTIA graph is able to reduce the host time spent on launching kernels from 5.9 ms to 43 µs, which results in a 5.4% end-to-end latency gain and less than 1 ms host processing time per token. The end-to-end speedup is not as significant as with GPU CUDA graph (67% with 5.2 ms launch overhead saved), because the MTIA 300 kernel is slower and its performance remains the bottleneck.

4) Multi-Node Performance

We also evaluated full-size DeepSeek-R1-671B on 8 MTIA 300 nodes. MTIA monolithic graph mode reaches 1.5× throughput compared to H200 Inductor graph mode. On MTIA 300, eager-mode prefill throughput is on par with monolithic graph mode (varying within range of 91%–117%)

# of jobs	Kernel Launch	Graph Launch	Speedup
1	4.37 µs	0.61 µs	7.12×
10	47.87 µs	3.93 µs	12.19×
100	496.01 µs	36.49 µs	13.59×

(a) Total kernel launch latency with operator-level benchmarks, based on the MTIA Kernel Launcher in Fig. 2.

Model	Kernel Launch	Graph Launch	Speedup
Llama3.2-1B	5876 µs	43 µs	136.6×
DeepSeek-R1	2528 µs	29 µs	87.1×

(b) Total kernel launch latency of end-to-end models. Batch size is 16. We observe similar speedup with other batch sizes.

TABLE V: Comparison of kernel launch latency in pure eager mode and with MTIA graph on MTIA 300.

because execution is device bound; for decode, eager-mode throughput drops to 17%–40% of monolithic graph mode, caused by inefficient collective handling and unoptimized kernels.

VIII. Conclusion

We presented the design and implementation of MTIA-RT, a runtime system for Meta's custom MTIA accelerators. Despite MTIA's architectural difference from GPUs, MTIA-RT provides a developer API that is sufficiently compatible with CUDA, simplifying integration with existing applications. It also includes several optimizations that can unlock high performance in order to support eager mode, which requires launching many fine-grained kernels, and graph mode, which includes a custom extension for a compiler-optimized monolithic graph. Finally, we demonstrated that we could provide a native PyTorch experience on top of MTIA-RT by deeply integrating with the underlying framework.

Acknowledgment

This work would not have been possible without the dedication and expertise of many colleagues across the MTIA software and firmware teams. We extend our thanks to Sameer Abu Asal, Maxim Adelman, Shishir Bharadwaj, Chris Bray, Cindy Cao, Joel Coburn, Alban Desmaison, Max Devyatov, Jordan Fix, Mario Flajslik, Cao Gao, Long Gong, Travis Graham, Wei Guo, Yunfei He, Kunming Ho, Kern Yan Hoe, Daniel Hou, Yusuo Hu, Jian Huang, Sterling Hughes, Shintaro Iwasaki, Nakul Iyer, Karan Jariwala, Zewei Jiang, Alexander Kabaev, Gleb Kurtsou, Jason Lee, Ethan Li, Richard Li, Eric Lin, Will Loughney, Blake Matheny, Emmanuel Menage, Zizeng Meng, Ilia Nadeliaev, Feny Patel, Klint Qinami, Praveen Ramachandran, Poorvaja Ramani, Zhihong Ren, Jonah Saltzman, Marc Seeger, Murat Seker, Bobo Shi, Feng Shi, Sanjeev Sivasankaran, Benny Tsang, Jessica Vandebon, Bharath Vemula, Kevin Vigor, Sergii Vystoropskyi, Andy (An) Wang, Guangyu Wang, Patrick Jun Wang, Shiguang Wang, Shiqin Wang, Richard Wareing, Jason White, Edward Yang, Jeremy Yang, Saili Yang, Tao Yang, Yimeng Yang, Ming-Ho Yee, Ana Zehtabioskuie, Alberto Zeni, Zitong Zeng, Feixiong

Zhang, Hongyang Zhao, Wenting Zhao, Boyue Zheng, Hal Zheng, Roger Zheng, Zhaoqi Zhu, Richard Zou and many others who have contributed to the design, implementation, and production operation of MTIA-RT.

REFERENCES

[1] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The Design Process for Google's Training Chips: TPUv2 and TPUv3," *IEEE Micro*, vol. 41, no. 2, pp. 56–63, 2021. [Online]. Available: https://doi.org/10.1109/MM.2021.3058217

[2] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell, J. Hwang, R. Leslie-Hurd, M. Bye, E. Creswick, M. Boyd, M. Venigalla, E. Laforge, J. Purdy, P. Kamath, D. Maheshwari, M. Beidler, G. Rosseel, O. Ahmad, G. Gagarin, R. Czekalski, A. Rane, S. Parmar, J. Werner, J. Sproch, A. Macias, and B. Kurtz, "Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 145–158. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00023

[3] M. Emani, V. Vishwanath, C. Adams, M. E. Papka, R. Stevens, L. Florescu, S. Jairath, W. Liu, T. Nama, and A. Sujeeth, "Accelerating Scientific Applications With SambaNova Reconfigurable Dataflow Architecture," *Computing in Science & Engineering*, vol. 23, no. 2, pp. 114–119, 2021. [Online]. Available: https://doi.org/10.1109/MCSE.2021.3057203

[4] X. Fu, Z. Zhang, H. Fan, G. Huang, M. El-Shabani, R. Huang, R. Solanki, F. Wu, R. Diamant, and Y. Wang, "Distributed Training of Large Language Models on AWS Trainium," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2024, pp. 961–976. [Online]. Available: https://doi.org/10.1145/3698038.3698535

[5] S. Xu and C. Ramakrishnan, " Inside Maia 100 ," in *2024 IEEE Hot Chips 36 Symposium (HCS)* . IEEE Computer Society, August 2024, pp. 1–17. [Online]. Available: https://doi.org/10.1109/HCS61935.2024.10665248

[6] J. Vasiljevic and D. Capalija, " Blackhole & TT-Metalium: The Standalone AI Computer and its Programming Model ," in *2024 IEEE Hot Chips 36 Symposium (HCS)* . IEEE Computer Society, August 2024, pp. 1–30. [Online]. Available: https://doi.org/10.1109/HCS61935.2024.10664810

[7] S. Lie, " Cerebras Architecture Deep Dive: First Look Inside the HW/SW Co-Design for Deep Learning : Cerebras Systems ," in *2022 IEEE Hot Chips 34 Symposium (HCS)* . IEEE Computer Society, August 2022, pp. 1–34. [Online]. Available: https://doi.org/10.1109/HCS55958.2022.9895479

[8] 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.

[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] M. Team, "MTIA 300: Meta's first training chip featuring built-in NICs and collective offloading engines," in *Proceedings of the 53rd Annual International Symposium on Computer Architecture*, 2026, pp. 1–13.

[11] Z. Jia, B. Tillman, M. Maggioni, and D. P. Scarpazza, "Dissecting the GraphCore IPU architecture via microbenchmarking," *arXiv preprint arXiv:1912.03413*, 2019.

[12] Y. J. Song, A. Tulloch, H. Reddy, C. Tang, and V. Thakkar. (2026) Four MTIA Chips in Two Years: Scaling AI Experiences for Billions. [Online]. Available: https://ai.meta.com/blog/meta-mtia-scale-ai-chips-for-billions/

[13] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, p. 39–55, Mar 2008.

[14] OpenXLA. PJRT - uniform device API. [Online]. Available: https://openxla.org/xla/pjrt

[15] A. Gray. (2019) Getting started with CUDA graphs. Accessed: 2025-12-04. [Online]. Available: https://developer.nvidia.com/blog/cuda-graphs/

[16] T. M. Aamodt, W. W. L. Fung, T. G. Rogers, and M. Martonosi, *General-purpose graphics processor architectures*. Springer, 2018.

[17] GraphCore. (2025) Support for TensorFlow 2. Accessed: 2025-12-04. [Online]. Available: https://docs.graphcore.ai/projects/tensorflow-user-guide/en/latest/tensorflow/targeting_tf2.html

[18] A. Suhan, D. Libenzi, A. Zhang, P. Schuh, B. Saeta, J. Y. Sohn, and D. Shabalin, "Lazytensor: combining eager execution with domain-specific compilers," *arXiv preprint arXiv:2102.13267*, 2021.

[19] M. Emani, V. Vishwanath, C. Adams, M. E. Papka, R. Stevens, L. Florescu, S. Jairath, W. Liu, T. Nama, and A. Sujeeth, "Accelerating scientific applications with sambanova reconfigurable dataflow architecture," *Computing in Science & Engineering*, vol. 23, no. 2, pp. 114–119, 2021.

[20] D. Luebke, "Cuda: Scalable parallel programming for high-performance scientific computing," in *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, 2008, pp. 836–838.

[21] B. Steiner, Z. DeVito, S. Chintala, S. Gross, A. Paske, F. Massa, A. Lerer, G. Chanan, Z. Lin, E. Yang *et al.*, "PyTorch: An imperative style, high-performance deep learning library," 2019.

[22] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: a system for large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[23] AMD. (2025) Hipify documentation. Accessed: 2025-12-04. [Online]. Available: https://rocm.docs.amd.com/projects/HIPIFY/en/latest/

[24] B. Glick, A. Sabnis, R. Kempf, A. Goel, A. Lalwani, G. Feng, and K. Ranganath, "Rdaruntime: An os for ai accelerators," in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1576–1587. [Online]. Available: https://doi.org/10.1145/3624062.3624235

[25] AWS. (2025) Developer guide - NeuronX runtime. [Online]. Available: https://awsdocs-neuron.readthedocs-hosted.com/en/latest/neuron-runtime/nrt-api-guide.html#nrt-api-guide

[26] C. S. D. (1.4.0). (2025) Host runtime and tensor streaming. [Online]. Available: https://sdk.cerebras.net/tensor-streaming

[27] Intel. (2025) Intel gaudi api reference guides. [Online]. Available: https://docs.habana.ai/en/latest/API_Reference_Guides/index.html

[28] P. Barham, A. Chowdhery, J. Dean, S. Ghemawat, S. Hand, D. Hurt, M. Isard, H. Lim, R. Pang, S. Roy, B. Saeta, P. Schuh, R. Sepassi, L. E. Shafey, C. A. Thekkath, and Y. Wu, "Pathways: Asynchronous distributed dataflow for ml," 2022. [Online]. Available: https://arxiv.org/abs/2203.12533

[29] 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.

[30] J. Reed, Z. DeVito, H. He, A. Ussery, and J. Ansel, "torch.fx: Practical program capture and transformation for deep learning in python," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 638–651, 2022.

[31] R. Frostig, M. J. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," in *SysML conference 2018*, 2019.

[32] ONNX. (2025) Open neural network exchange: ONNX-MLIR. Accessed: 2025-12-04. [Online]. Available: https://onnx.ai/onnx-mlir

[33] T. Jin, G.-T. Bercea, T. D. Le, T. Chen, G. Su, H. Imai, Y. Negishi, A. Leu, K. O'Brien, K. Kawachiya *et al.*, "Compiling onnx neural network models using MLIR," *arXiv preprint arXiv:2008.08272*, 2020.

[34] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[35] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 10 684–10 695.

[36] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 611–626. [Online]. Available: https://doi.org/10.1145/3600006.3613165

[37] S. Nadampalli, "Accelerated pytorch inference with torch.compile on AWS Graviton processors," *https://aws.amazon.com/blogs/machine-learning/accelerated-pytorch-inference-with-torch-compile-on-aws-graviton-processors/*, 2024.

[38] PyTorch. (2022) Accelerating PyTorch with CUDA graphs. Accessed: 2025-12-04. [Online]. Available: https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/

[39] S. Kashmira, J. Dantanarayana, T. Sathiyalogeswaran, Y. Yuan, N. Talati, K. Flautner, L. Tang, and J. Mars, "GraphMend: Code transformations for fixing graph breaks in PyTorch 2," 2026. [Online]. Available: https://arxiv.org/abs/2509.16248

[40] 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.

[41] D. Zhuang, Z. Zheng, H. Xia, X. Qiu, J. Bai, W. Lin, and S. L. Song, "MonoNN: Enabling a new monolithic optimization space for neural network inference tasks on modern GPU-centric architectures," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 989–1005.

[42] S. Laine, T. Karras, and T. Aila, "Megakernels considered harmful: wavefront path tracing on GPUs," in *Proceedings of the 5th High-Performance Graphics Conference*, ser. HPG '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 137–143. [Online]. Available: https://doi.org/10.1145/2492045.2492060

[43] L. Jia, Y. Liang, X. Li, L. Lu, and S. Yan, "Enabling efficient fast convolution algorithms on GPUs via megakernels," *IEEE Transactions on Computers*, vol. 69, no. 7, pp. 986–997, 2020.

[44] X. Cheng, Z. Zhang, Y. Zhou, J. Ji, J. Jiang, Z. Zhao, Z. Xiao, Z. Ye, Y. Huang, R. Lai, H. Jin, B. Hou, M. Wu, Y. Dong, A. Yip, Z. Ye, S. Wang, W. Yang, X. Miao, T. Chen, and Z. Jia, "Mirage persistent kernel: A compiler and runtime for mega-kernelizing tensor programs," 2025. [Online]. Available: https://arxiv.org/abs/2512.22219

[45] R. Zheng and S. Pai, "Efficient execution of graph algorithms on cpu with simd extensions," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 262–276.

[46] F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber, "Efficient utilization of simd extensions," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 409–425, 2005.

[47] C. Zhang, L. Ma, J. Xue, Y. Shi, Z. Miao, F. Yang, J. Zhai, Z. Yang, and M. Yang, "Cocktailer: Analyzing and optimizing dynamic control flow in deep learning," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 681–699.

[48] J. Hamilton, "Aws inferentia machine learning processor," *Aws inferentia machine learning processor*, vol. 11, 2018.

[49] C. Y. Kaz Sato. (2017) An in-depth look at google's first tensor processing unit (tpu). [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

[50] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The design process for google's training chips: Tpuv2 and tpuv3," *IEEE Micro*, vol. 41, no. 2, pp. 56–63, 2021.

[51] N. P. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. Patterson, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," 2023. [Online]. Available: https://arxiv.org/abs/2304.01433

[52] A. Agrawal, A. Modi, A. Passos, A. Lavoie, A. Agarwal, A. Shankar, I. Ganichev, J. Levenberg, M. Hong, R. Monga *et al.*, "TensorFlow eager: A multi-stage, Python-embedded DSL for machine learning," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 178–189, 2019.

[53] NVIDIA. (2025) CUDA Runtime API Documentation. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[54] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54*, p. 4, 2014.

[55] NVIDIA. (2026) nvFatbin. [Online]. Available: https://docs.nvidia.com/cuda/nvfatbin/index.html

[56] R. Love, "Get on the d-bus," *Linux Journal*, vol. 2005, no. 130, p. 3, 2005.

[57] A. Williamson. (2020) VFIO - virtual function I/O. Accessed: 2025-12-04. [Online]. Available: https://www.kernel.org/doc/Documentation/vfio.txt

[58] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble *et al.*, "Snap: A microkernel approach to host networking," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 399–413.

[59] C. Modi and S. G. Kulkarni, "Kernel bypass and user-space network frameworks for high-performance computing workloads," in *2024 IEEE 31st International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*. IEEE, 2024, pp. 187–188.

[60] R. Chen and G. Sun, "A survey of kernel-bypass techniques in network stack," in *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, 2018, pp. 474–477.

[61] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, "gpucc: an open-source GPGPU compiler," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 105–116. [Online]. Available: https://doi.org/10.1145/2854038.2854041

[62] W. Van Oortmerssen. (2014) Flatbuffers: a memory efficient serialization library. Accessed: 2025-12-04. [Online]. Available: https://android-developers.googleblog.com/2014/06/flatbuffers-memory-efficient.html

[63] C. O. diary. (2020) Small vector optimization. Accessed: 2025-12-04. [Online]. Available: https://cpp-optimizations.netlify.app/small_vectors/

[64] M. Tofte and J.-P. Talpin, "Region-based memory management," *Information and computation*, vol. 132, no. 2, pp. 109–176, 1997.

[65] P. Architecture. (2026) The risc-v instruction set manual, volume ii. Accessed: 2026-01-20. [Online]. Available: https://docs.riscv.org/reference/isa/_attachments/riscv-privileged.pdf

[66] M. Harris. (2013) Unified memory in cuda 6. Accessed: 2025-12-04. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

[67] PyTorch. (2025) ATen code generator. Accessed: 2025-12-04. [Online]. Available: https://github.com/pytorch/pytorch/tree/main/torchgen

[68] ——. (2023) torch.mtia reference apis. [Online]. Available: https://docs.pytorch.org/docs/stable/mtia.html

[69] ——. (2025) torch.accelerator. Accessed: 2025-12-04. [Online]. Available: https://docs.pytorch.org/docs/stable/accelerator.html

[70] @wizzniu. (2023) Re-implement pin_memory to be device-agnostic by leveraging the accelerator concept. [Online]. Available: https://github.com/pytorch/pytorch/pull/126376

[71] E. Z. Yang. (2023) Rfc: Structured kernel definitions in pytorch. [Online]. Available: https://github.com/pytorch/rfcs/blob/rfc-0005/RFC-0005-structured-kernel-definitions.md

[72] D. Khudia, J. Huang, P. Basu, S. Deng, H. Liu, J. Park, and M. Smelyanskiy, "Fbgemm: Enabling high-performance low-precision deep learning inference," *arXiv preprint arXiv:2101.05615*, 2021.

[73] D. Ivchenko, D. Van Der Staay, C. Taylor, X. Liu, W. Feng, R. Kindi, A. Sudarshan, and S. Sefati, "Torchrec: a pytorch domain library for recommendation systems," in *Proceedings of the 16th ACM Conference on Recommender Systems*, 2022, pp. 482–483.

[74] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer *et al.*, "PyTorch FSDP: experiences on scaling fully sharded data parallel," *arXiv preprint arXiv:2304.11277*, 2023.

[75] @egienvalue. (2023) Support mtia device type. [Online]. Available: https://github.com/meta-pytorch/torchrec/pull/1382

[76] M. AI. (2024) Llama 4: Multimodal intelligence. Accessed: 2025-12-04. [Online]. Available: https://ai.meta.com/blog/llama-4-multimodal-intelligence/

[77] H. Hoffman and F. Oh. (2024) Constant time launch for straight-line CUDA graphs and other performance enhancements. Accessed: 2026-04-07. [Online]. Available: https://developer.nvidia.com/blog/constant-time-launch-for-straight-line-cuda-graphs-and-other-performance-enhanceme

[78] D. Guo, D. Yang, H. Zhang, J. Song, P. Wang, Q. Zhu, R. Xu, R. Zhang, S. Ma, X. Bi, X. Zhang, X. Yu, Y. Wu, Z. F. Wu, Z. Gou, Z. Shao, Z. Li, Z. Gao, A. Liu, B. Xue, B. Wang, B. Wu, B. Feng, C. Lu, C. Zhao, C. Deng, C. Ruan, D. Dai, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Xu, H. Ding, H. Gao, H. Qu, H. Li, J. Guo, J. Li, J. Chen, J. Yuan, J. Tu, J. Qiu, J. Li, J. L. Cai, J. Ni, J. Liang, J. Chen, K. Dong, K. Hu, K. You, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Zhao,

L. Wang, L. Zhang, L. Xu, L. Xia, M. Zhang, M. Zhang, M. Tang, M. Zhou, M. Li, M. Wang, M. Li, N. Tian, P. Huang, P. Zhang, Q. Wang, Q. Chen, Q. Du, R. Ge, R. Zhang, R. Pan, R. Wang, R. J. Chen, R. L. Jin, R. Chen, S. Lu, S. Zhou, S. Chen, S. Ye, S. Wang, S. Yu, S. Zhou, S. Pan, S. S. Li, S. Zhou, S. Wu, T. Yun, T. Pei, T. Sun, T. Wang, W. Zeng, W. Liu, W. Liang, W. Gao, W. Yu, W. Zhang, W. L. Xiao, W. An, X. Liu, X. Wang, X. Chen, X. Nie, X. Cheng, X. Liu, X. Xie, X. Liu, X. Yang, X. Li, X. Su, X. Lin, X. Q. Li, X. Jin, X. Shen, X. Chen, X. Sun, X. Wang, X. Song, X. Zhou, X. Wang, X. Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. Zhang, Y. Xu, Y. Li, Y. Zhao, Y. Sun, Y. Wang, Y. Yu, Y. Zhang, Y. Shi, Y. Xiong, Y. He, Y. Piao, Y. Wang, Y. Tan, Y. Ma, Y. Liu, Y. Guo, Y. Ou, Y. Wang, Y. Gong, Y. Zou, Y. He, Y. Xiong, Y. Luo, Y. You, Y. Liu, Y. Zhou, Y. X. Zhu, Y. Huang, Y. Li, Y. Zheng, Y. Zhu, Y. Ma, Y. Tang, Y. Zha, Y. Yan, Z. Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Xu, Z. Xie, Z. Zhang, Z. Hao, Z. Ma, Z. Yan, Z. Wu, Z. Gu, Z. Zhu, Z. Liu, Z. Li, Z. Xie, Z. Song, Z. Pan, Z. Huang, Z. Xu, Z. Zhang, and Z. Zhang, "Deepseek-r1 incentivizes reasoning in llms through reinforcement learning," *Nature*, vol. 645, no. 8081, p. 633–638, Sep. 2025. [Online]. Available: http://dx.doi.org/10.1038/s41586-025-09422-z

[79] 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.