Pulse: A Profiling and Visualization Infrastructure for Heterogeneous Managed Systems

Michail Papadimitriou
The University of Manchester
Manchester, UK
michail.papadimitriou@manchester.ac.uk

Maria Xekalaki
The University of Manchester
Manchester, UK
maria.xekalaki@manchester.ac.uk

Orion Papadakis
The University of Manchester
Manchester, UK
orion.papadakis@manchester.ac.uk

Ruiqi Ye
The University of Manchester
Manchester, UK
ruiqi.ye@manchester.ac.uk

Athanasios Stratikopoulos
The University of Manchester
Manchester, UK
athanasios.stratikopoulos@manchester.ac.uk

Christos Kotselidis
The University of Manchester
Manchester, UK
christos.kotselidis@manchester.ac.uk

Abstract

TornadoVM is an open-source framework that enables Java applications to execute on heterogeneous hardware accelerators, such as GPUs and FPGAs. While TornadoVM simplifies programmability, understanding and optimizing the performance of applications on heterogeneous systems remains a significant challenge.

This paper introduces Pulse, a profiling infrastructure designed to collect, correlate, and visualize detailed performance metrics for application components offloaded to accelerators. At its core, Pulse includes a profiler that integrates information from both the managed runtime and the underlying hardware. The collected metrics include fine-grained execution timing, data-transfer costs, compilation overheads, and power consumption during accelerator execution. In addition, Pulse provides an interactive visualization layer that presents these metrics in an accessible and actionable manner, helping developers analyze performance bottlenecks and improve the efficiency of Java applications running on heterogeneous hardware through TornadoVM.

Using a Java-native implementation of the Llama3 inference pipeline (`GPULlama3.java`) as a case study, this paper demonstrates how Pulse facilitates profiling-guided optimization. By following profiling information generated by Pulse, developers were able to refactor and fuse fine-grained GPU kernels, resulting in reduced kernel launches per layer from 19 to 13, a 38% decrease in data-transfers cost, and an overall 18.4% improvement in the execution time per-layer. Finally, the instrumentation overhead of Pulse is evaluated to be 1.5% for coarse-grained microbenchmarks, and up to 30% for workloads dominated by short-lived kernels such as `GPULlama3.java` - representing a worst-case scenario for fine-grained profiling.

CCS Concepts

• **Software and its engineering → Software maintenance tools**.

Keywords

TornadoVM, Profiling, Visualization, Heterogeneous Systems, Performance Optimization, Java, GPUs

1 Introduction

The increasing availability of heterogeneous hardware accelerators - such as GPUs and FPGAs - has made them a compelling target for high-performance and energy-efficient computing. In the Java ecosystem, TornadoVM [14, 19, 23, 24] provides a portable and high-level programming model that enables Java applications to execute on OpenCL, PTX, and SPIR-V compatible devices. By integrating with OpenJDK [3] and GraalVM [32], TornadoVM transparently compiles Java bytecode to accelerator-specific kernels and manages the execution and data transfers required to run on heterogeneous hardware.

While this abstraction simplifies programmability, it also introduces significant challenges for understanding and optimizing performance [17]. Performance behavior in heterogeneous systems depends on multiple factors, such as kernel execution time, data-transfer overheads, Just-In-Time (JIT) compilation costs, device power characteristics, and runtime decisions made by TornadoVM. These metrics arise from both the managed runtime environment and the hardware accelerators themselves, forming a complex interplay that can be difficult for developers to reason about. Existing profiling tools either lack insight into TornadoVM's execution model or do not correlate metrics across the boundary between the main processor and the co-processor devices, limiting their usefulness for performance debugging and tuning.

To address these challenges, this paper introduces Pulse, an open-source[1] profiling infrastructure designed for Java applications running on heterogeneous accelerators. The proposed infrastructure provides:

(1) A profiler that captures detailed performance attributes across the execution stack, including fine-grained kernel

[1] https://github.com/beehive-lab/TornadoVMPulse

timing, data-movement statistics, compilation timing, and device-level power measurements.
(2) A visualization layer that ingests profiling logs and presents them through an interactive dashboard, allowing developers to intuitively explore performance behavior, identify bottlenecks, and understand the impact of TornadoVM's execution decisions.

Although Pulse is implemented within the TornadoVM ecosystem, its modular architecture along with its capability to integrate metrics from both the managed runtime and the hardware accelerators allow it to be adapted to other heterogeneous execution frameworks.

The remainder of this paper is organized as follows. Section 2 presents background information on TornadoVM and outlines the challenges in profiling applications in heterogeneous managed systems (i.e., managed runtime systems that target heterogeneous hardware). Section 3 describes the Pulse profiling infrastructure and its visualization layer, while Section 4 presents the experimental evaluation and a case-study that showcases how Pulse is used to optimize the design and performance of a Java-native GPU-accelerated Large Language Model (LLM) inference engine. Section 5 describes the related work in the context of Pulse. Finally, Section 6 concludes this paper and outlines the future work.

2 Background

This section provides the necessary background for understanding the design and motivation of Pulse. Section 2.1 present an overview of TornadoVM, describing its execution model, orchestration layer, and internal bytecode mechanism used to coordinate computation across heterogeneous hardware. Section 2.2 outlines the challenges associated with profiling applications in managed heterogeneous systems, highlighting the limitations of existing tools and motivating the need for a unified cross-layer profiling infrastructure.

2.1 TornadoVM Overview

TornadoVM [14, 16] is a plug-in for OpenJDK and GraalVM that enables Java applications to automatically compile and run computations on heterogeneous hardware. It adopts a task-based programming model in which developers identify parallelizable units of work (i.e., *tasks*) that can be offloaded to accelerators. TornadoVM JIT compiles at runtime these tasks to OpenCL, PTX or SPIR-V, enabling their execution on a wide range of devices including multi-core CPUs, GPUs from multiple vendors, and FPGAs.

TornadoVM follows a layered architecture composed of four main components, as shown in Figure 1. At the top level, developers interact with the TornadoVM API to express computation using tasks. A task is a Java lambda function that will later be compiled and executed on an accelerator. For more advanced usage, developers can construct *TaskGraphs* - groups of tasks that share data and are executed on the same device - to enable data persistence and reduce redundant memory transfers. The API also exposes annotations and low-level primitives (e.g., local memory, synchronization barriers) for developers who require fine-grained control over performance.

Once the TaskGraphs are defined, TornadoVM performs a set of optimizations to reduce unnecessary memory transfers and subsequently generates *TornadoVM bytecodes*. These bytecodes, described below, orchestrate JIT compilation, data management, scheduling, and execution on the target accelerator.

2.1.1 TornadoVM Orchestration. TornadoVM uses an internal bytecode layer that acts as an intermediate representation between Java bytecode and device-specific kernels. This internal bytecode system encodes a sequence of operations - such as memory allocation, data transfers, JIT compilation, kernel launches, and synchronization - that are abstracted away from the user and executed transparently at runtime. The Graph Optimizer produces these bytecodes, which are then interpreted by the TornadoVM Orchestrator.

The bytecode set includes operations such as:

- `BEGIN/END`: Mark the boundaries of execution contexts
- `ALLOC`: Allocate memory on a device
- `STREAM_IN/STREAM_OUT`: Transfer data between host and device
- `COPY_IN/COPY_OUT`: Allocate and transfer data in a single operation
- `LAUNCH`: Execute a kernel on a device
- `BARRIER`: Synchronize execution
- `DEALLOC`: Free memory on a device

Analyzing these bytecodes provides valuable insight into task dependencies, execution ordering, memory usage, and performance-critical operations. However, this analysis becomes increasingly complex as the number of tasks, devices, or data dependencies grows. In Section 4.1, we demonstrate this complexity through a Java software library that encompasses more than 2000 TornadoVM bytecodes, illustrating the need for automated profiling support.

2.2 Profiling in Heterogeneous Managed Systems

Profiling applications running on managed runtimes, such as the JVM, poses challenges not present in traditional native environments. Native profilers typically instrument binaries or read hardware counters directly, but managed systems introduce additional layers - including JIT compilation, garbage collection, and runtime scheduling - that influence performance in ways that are difficult to attribute precisely.

2.2.1 Challenges of Profiling in Managed Heterogeneous Systems. Systems like TornadoVM require profiling across both *managed-runtime events* (e.g., JIT compilation, task scheduling, bytecode interpretation) and *device-level events* (e.g., kernel launches, data transfers, power measurements). These events occur across different address spaces, hardware devices, and execution APIs (OpenCL, PTX, SPIR-V), each with distinct timing semantics. A unified profiling solution must therefore: (i) correlate heterogeneous execution events to the originating Java tasks; (ii) track cross-device data movement; (iii) provide accurate timing information with minimal runtime perturbation.

2.2.2 Limitations of Existing Tools. Table 1 summarizes representative profiling tools used across managed and native environments. Vendor-specific GPU profilers, such as *NVIDIA Nsight*, *Intel VTune*,

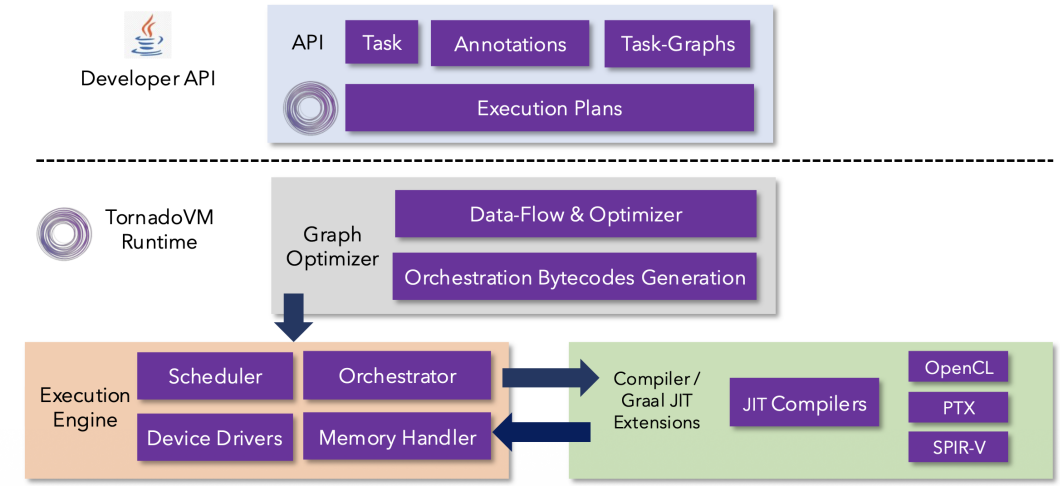

Figure 1: The TornadoVM architecture.

and *AMD Radeon Profiler*, expose detailed hardware-level statistics but lack integration with high-level Java or managed-language semantics. Conversely, JVM-level profilers (e.g., *Java Flight Recorder*) provide rich insights into Java threads, garbage collection, and method-level events but remain unaware of offloaded accelerator computations.

None of these tools can natively correlate Java-level constructs (such as TornadoVM TaskGraphs) with device-level performance metrics. As a result, developers must often combine multiple tools, manually align timestamps, and infer host–accelerator relationships - an error-prone and time-consuming process.

2.2.3 Need for a Unified Profiler. The combination of JIT compilation, as well as the offloading of the compiled methods on heterogeneous hardware accelerators in TornadoVM, highlights the need for a dedicated profiling infrastructure that bridges the semantic gap between managed runtimes and accelerator-level execution. Such a system must (i) instrument TornadoVM's orchestration bytecodes transparently, (ii) expose end-to-end timing and power information per task, and (iii) remain portable across hardware backends and vendors.

To address these challenges, we developed Pulse, a cross-layer profiling framework that captures fine-grained traces of orchestration bytecodes, kernel executions, and data transfers. Pulse aggregates these events into unified timelines aligned with TornadoVM's TaskGraph semantics, enabling developers to reason about performance holistically. The next section describes the design and implementation of this profiling infrastructure and its associated visualization capabilities.

3 The Pulse Profiling Infrastructure

This section presents the design and architecture of the Pulse infrastructure, which combines two tightly integrated components: (i) a profiler that collects fine-grained performance metrics (Section 3.1), and (ii) a visualization layer that consumes the profiler's logs and presents them through an interactive dashboard (Section 3.2). Together, these components enable end-to-end profiling of Java applications running on heterogeneous hardware. Figure 2 illustrates the modular architecture of Pulse and the interaction between its profiling and visualization components.

Table 1: Feature comparison of heterogeneous profiling tools (✓: fully supported; ✗: not supported; △: partially or conditionally supported).

Feature	Pulse	Nsight	VTune	Radeon	JFR
Java/Managed Lang.	✓	✗	△	✗	✓
Heterogeneous HW	✓	△	△	△	✗
Cross-vendor	✓	✗	✗	✗	✗
Task Graph Aware	✓	✗	✗	✗	✗
Power Monitoring	✓	✓	✓	△	✗
JIT Profiling	✓	✗	△	✗	✓
Data Transfer	✓	✓	✓	✓	✗
Open Source	✓	✗	✗	✓	✓
Overhead	1.5-30%	2-10%	5-15%	3-12%	2-5%

3.1 Pulse Profiler

At the core of Pulse lies a profiler that integrates directly with the runtime system it is embedded into. In the context of this work, Pulse is integrated with the TornadoVM runtime, enabling developers to obtain detailed insights into the execution behavior of accelerated Java tasks. The profiler collects a rich set of performance metrics during the various stages of TornadoVM's execution pipeline, including JIT compilation of Java methods into accelerator-specific code, data transfers between the host and device, task dispatching, and kernel execution on heterogeneous hardware.

Table 2: Summary of key profiling metrics collected by Pulse.

Metric Name	Description	Unit
COPY_IN_TIME	Time for data copy from host to device for the TaskGraph.	ns
COPY_OUT_TIME	Time for data copy from device to host for the TaskGraph.	ns
DISPATCH_TIME	Time spent dispatching a submitted OpenCL command for the TaskGraph.	ns
TOTAL_KERNEL_TIME	Sum of all kernel execution times within the TaskGraph.	ns
TOTAL_TASK_GRAPH_TIME	Total execution time for the TaskGraph, including all overheads.	ns
TOTAL_GRAAL_COMPILE_TIME	Graal compilation time (Java to OpenCL C/PTX/SPIR-V) for the TaskGraph.	ns
TOTAL_DRIVER_COMPILE_TIME	Driver compilation time for the TaskGraph.	ns
TASK_KERNEL_TIME	Kernel execution time for a specific task.	ns
TASK_COPY_IN_SIZE_BYTES	Total bytes copied-in for a specific task.	bytes
TASK_COPY_OUT_SIZE_BYTES	Total bytes copied-out for a specific task.	bytes
POWER_USAGE_mW	Power consumed to execute a given task.	mW
BACKEND	TornadoVM backend selected (SPIRV, PTX, or OpenCL).	N/A

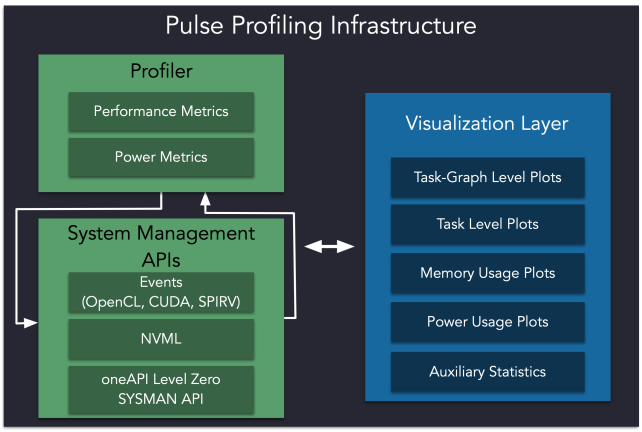

Figure 2: The architecture of the Pulse infrastructure.

3.1.1 Key Profiling Metrics. The profiler records a comprehensive collection of timers and counters (reported in nanoseconds, unless stated otherwise) to provide a holistic view of the application's performance. These metrics, summarized in Table 2, are grouped into several categories:

- **TaskGraph Timers:** These metrics capture high-level performance across an entire TaskGraph. The metrics include COPY_IN_TIME and COPY_OUT_TIME for data transfers between the host memory and the device memory, DISPATCH_TIME for command submission, TOTAL_KERNEL_TIME aggregating all kernel executions within the graph, and TOTAL_TASK_GRAPH_TIME for end-to-end execution. It also reports compilation and code generation metrics, such as TOTAL_GRAAL_COMPILE_TIME (from Java to intermediate representation, e.g., Graal-IR), TOTAL_CODE_GENERATION_TIME (the code generation from the intermediate representation to OpenCL C/PTX/SPIR-V), TOTAL_DRIVER_COMPILE_TIME (the final-sta- ge compilation of the generated kernels to machine code performed by the driver), and TOTAL_BYTE_CODE_GENERATION (TornadoVM's internal bytecode generation [16]). The explicit measurement of these distinct compilation phases allows developers to distinguish framework overheads from true kernel performance, which is particularly important in JIT-compiled systems where initial warm-up costs can dominate short executions.
- **Per-Task Metrics:** For each individual task, Pulse records TASK_KERNEL_TIME, per-task compilation times (TASK_COMPILE_GRAAL_TIME, TASK_COMPILE_DRIVER_TIME, TASK_CODE_GENERATION_TIME), and data transfer volumes (TASK_COPY_IN_SIZE_BYTES, TASK_COPY_OUT_SIZE_BYTES). These metrics enable fine-grained performance analysis across the tasks of a TaskGraph.
- **Device and Backend Information:** The profiler logs the selected backend (OpenCL, PTX, or SPIR-V), together with device identifiers and device names, allowing correlations between performance behavior and the underlying hardware devices.
- **Power Consumption:** The POWER_USAGE_mW metric captures instantaneous power usage for each task. Including power counters is increasingly important as energy efficiency becomes a first-order concern in heterogeneous systems, especially for high-intensity GPU kernels.

Coverage of Profiling Metrics. The Pulse metric set is designed to provide concise yet comprehensive coverage of all performance-relevant phases in the TornadoVM heterogeneous execution pipeline. TornadoVM operates at the boundary between the JVM and hardware accelerators, dynamically managing multi-stage JIT compilation, runtime orchestration, data movement, and kernel execution. As a result, performance bottlenecks frequently arise outside the visibility of conventional JVM-level or device-level profilers.

To validate the coverage of the selected metrics, we have profiled a representative workload consisting of a single TornadoVM TaskGraph that executes an 8192×8192 matrix multiplication on the same testbed that is described in Section 4. Figure 3 shows stacked bars of all captured metrics for both cold and warm executions. For a cold run, which include JIT compilation, the aggregated metrics account for 94.3% of the total execution time. As shown in the left bar, the kernel execution is responsible for 84.2% of the total task-graph time, while the compilation metrics consume a noticeable fraction: Graal compilation (3.4%), driver compilation (2.7%), and code generation (0.3%). Additionally, the data movements from the

```
1 long timer = meta.getProfiler().getTimer(ProfilerType.TOTAL_KERNEL_TIME);
2 // Register globalTime
3 meta.getProfiler().setTimer(ProfilerType.TOTAL_KERNEL_TIME, timer + tornadoKernelEvent.getElapsedTime());
```

Listing 1: Example of recording the kernel execution time using the Pulse profiling API. This snippet shows how the `TOTAL_KERNEL_TIME` metric is retrieved and updated using the Pulse profiler within the TornadoVM runtime.

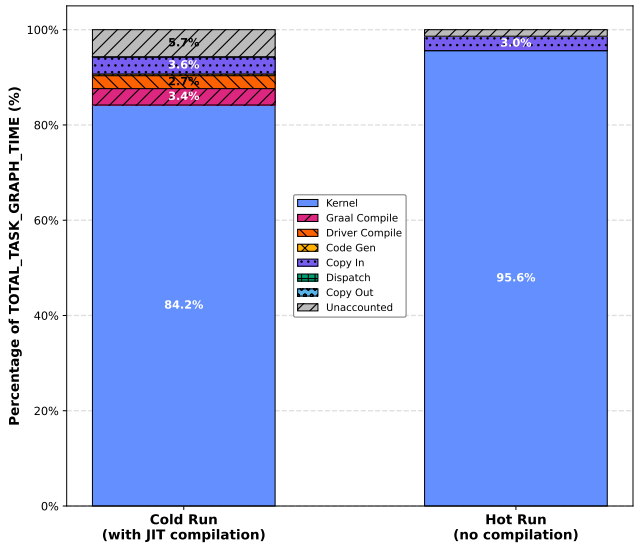

Figure 3: Execution time breakdown for cold and warm executions of a TornadoVM TaskGraph performing matrix multiplication.

host memory to the device memory and vice-versa, takes 3.6% and less than 0.1%, respectively. The orchestration time spent in dispatching the kernel execution and data movements is below 0.1%. Hence, the remaining 5.7% corresponds to unaccounted time.

On the other hand, the right bar presents the break-down analysis of the captured metrics for a warm run, where the compiled code is reused from the code cache. In this case, the captured metrics account for 98.6% of the total execution time, with the most dominant fraction being the kernel execution time (95.6%).

These results indicate that the metrics collected by Pulse capture the dominant sources of the execution cost in TornadoVM workloads, including compilation, kernel execution, data movement, and runtime orchestration, resulting in sufficient coverage for diagnosing performance bottlenecks in heterogeneous Java applications.

3.1.2 The Design of the Profiling API. The Pulse profiling API is designed to expose performance events at the level of TornadoVM's internal orchestration, rather than relying on sampling or external instrumentation. Profiling hooks are embedded directly within the runtime layer at well-defined execution stages (e.g., compilation stages, data transfers, kernel launches), allowing metrics to be recorded only when these events occur. This explicit, event-driven interface avoids periodic sampling and reduces perturbation by limiting instrumentation to performance-critical operations already present in TornadoVM's execution pipeline. In contrast to traditional JVM profilers or vendor GPU profilers that operate independently at the managed runtime layer or the device layer, the Pulse API is intentionally positioned between the two layers, enabling cross-layer correlation with minimal instrumentation overhead as will be discussed in Section 4.2. Additionally, its ability to capture execution traces across multiple layers—from the managed runtime to native accelerator execution - makes it well suited to augment general-purpose frameworks (e.g., Kieker [18, 31], OpenTelemetry [22]) that do not natively support GPU tracing.

Metrics are updated through lightweight timer accumulation and counter updates, as illustrated in Listing 1 (line 3), and profiling can be enabled or disabled at runtime without recompilation. The profiler can be enabled either via command-line flags (e.g., `tornado --enableProfiler console` or `tornado --enableProfiler silent`) or during programming via the `TornadoExecutionPlan` API (e.g., `executionPlan.withProfiler(ProfilerMode.CONSOLE)`) [7].

In `console` mode, the profiler prints JSON output to `stdout` for each TaskGraph execution, while `silent` mode enables non-intrusive data collection for programmatic use. The profiling output may be saved to a file using `-dumpProfiler <FILENAME>`, producing a JSON log of the recorded metrics, or it can be streamed to a remote consumer via `-Dtornado.dump.to.ip= IP:PORT`. Using JSON as the output format promotes portability, as it is both human-readable and easily integrable with external tools, including the Pulse visualization layer. Listing 2 illustrates an indicative profiling trace of running the `fusedFeedForwardWithSiLUAndGLUActivation` task on Apple silicon as returned from Pulse in JSON.

3.1.3 Backend Support and Metric Collection. Pulse supports all TornadoVM backends: OpenCL, NVIDIA PTX, and SPIR-V. Timers for data transfers and kernel executions are obtained through backend-specific event mechanisms (e.g., OpenCL, CUDA, SPIR-V events).

Power metrics (`POWER_USAGE_mW`) are retrieved using vendor-supplied system management libraries. For NVIDIA GPUs, TornadoVM integrates with the NVIDIA Management Library (NVML) [9], while Intel GPUs using the SPIR-V backend rely on the Level Zero SYSMAN API [10]. Correct installation and configuration of these libraries (e.g., `libnvidia-ml.so`, or `ZES_ENABLE_SYSMAN=1` for Level Zero) is required. If these APIs are unavailable or if tasks are too short to trigger measurable energy deltas, power values may be reported as "n/a".

3.2 Pulse Visualization Layer

While the profiler component generates comprehensive raw data, manually interpreting large JSON logs can be both tedious and error-prone. The visualization layer addresses this challenge by providing

```
"layer_0.fused_ffn_w1_w3": {
   "BACKEND": "OPENCL",
   "METHOD":"TCL.fusedFeedForwardWithSiLU...",
   "DEVICE_ID": "0:0",
   "DEVICE": "Apple M3 Pro",
   "TOTAL_COPY_IN_SIZE_BYTES": "24",
   "POWER_USAGE_mW": "n/a",
   "SYSTEM_POWER_CONSUMPTION_W": "n/a",
   "SYSTEM_VOLTAGE_V": "n/a",
   "TASK_KERNEL_TIME": "34473"
},
```


Listing 2: JSON output of the Pulse profiler.

an interactive, web-based dashboard specifically designed for exploring and analyzing Pulse profiling data. Built using the Streamlit framework in Python [13], the Pulse visualization layer transforms profiler output into intuitive charts and tables that expose kernel execution behavior, data-transfer patterns, memory usage, and power measurements. By converting low-level logs into a modern, user-friendly interface, Pulse lowers the barrier to performance analysis - especially for developers who may have limited experience with heterogeneous systems.

3.2.1 Core Features. Pulse streamlines the performance-analysis workflow through several core features:

- **Log Ingestion and Processing:** The tool accepts Pulse profiler logs in their native JSON format, as well as raw text logs or pre-processed CSV files. Raw logs are automatically normalized and converted into CSV for internal analysis, and users may download the processed files for external use.
- **Interactive Dashboard:** The dashboard offers a highly interactive developer experience. A sidebar allows configuration of which metrics to display, selection of visualization types, and adjustment of time units (nanoseconds, milliseconds, or seconds). Collapsible information panels help manage screen space, and all processed data can be exported as downloadable CSV.

3.2.2 Key Charts of the Pulse Visualization Component. Pulse provides a collection of charts and bars that illuminate different aspects of application performance. These linked views support a hierarchical analysis workflow, from identifying high-level bottlenecks across entire TaskGraphs to pinpointing specific costly kernels or data-movement operations. This layered approach is essential for effective debugging and optimization in heterogeneous environments.

The key visualization features are grouped in the following categories:

- **Overall Task Time Distribution:** This group summarizes the execution time of individual tasks across the entire application run. A bar chart (Figure 4) shows the total kernel execution time per task, color-coded by the corresponding TaskGraph, helping identify globally expensive kernels. This helps recognizing the most computationally-intensive kernel. A pie chart (Figure 5) shows each task's proportional contribution to the total kernel execution time, immediately highlighting dominant workloads.
- **Total Execution Time Breakdown (Per TaskGraph):** These charts decompose execution time by TaskGraph and then by execution component (Kernel, Dispatch, Copy-In, Copy-Out). A stacked bar chart (Figure 6) compares how each TaskGraph spends its time across different phases. For example, the "logit" graph shows kernel execution as the primary contributor, while others may reveal heavy data-copy costs. A sunburst chart (Figure 7) presents a hierarchical breakdown: the outer ring divides each TaskGraph's internal components (kernel, copies, dispatch), while the inner ring shows their contribution to total runtime. For instance, it shows that "logits" contributes 18% to the total time.
- **TaskGraph-specific Analysis (Per-Task Breakdown):** Developers can select an individual TaskGraph from a dropdown menu to view detailed per-task metrics. A bar chart (Figure 8) displays how kernel execution, data movement, and dispatch contribute to runtime within the selected graph. For example, in a "layer_3" TaskGraph, the `layer_3.fused_ffn` task may appear as a dominant contributor.
- **Memory and Power Usage Charts:** Additional charts report per-graph memory usage (via `TASK_COPY_IN_SIZE_BYTES`) and average power consumption (`POWER_USAGE_mW`), supporting the identification of memory-heavy or power-intensive components.
- **Copy-In/Copy-Out Analysis**: Dedicated tables and bar charts summarize the time spent on data-transfer operations and their overall contribution to execution time, allowing developers to quickly assess host–device data-movement overheads.
- **Raw Data and Auxiliary Statistics**: For deeper inspection, Pulse exposes the raw profiler data as an interactive DataFrame and provides summary statistics for selected metrics.

4 Experimental Evaluation

This section evaluates the effectiveness and practicality of the Pulse profiling infrastructure. Section 4.1 presents a detailed case study based on `GPULlama3.java`, a Java-native GPU-accelerated LLM inference engine, demonstrating how Pulse enables profiling-guided optimization. The case study illustrates the iterative transformation from fine-grained task decomposition to fused-kernel execution and shows how Pulse exposes performance bottlenecks that guide these improvements. Section 4.2 quantifies the runtime overhead introduced by Pulse across both microbenchmarks and real-world workloads, highlighting how instrumentation cost varies with task granularity.

All experiments were conducted on a Linux server equiped with an Intel i9-14900 CPU, 64GB RAM, and an NVIDIA RTX 3070 GPU using TornadoVM v1.1.1 with the OpenCL backend. The OS used was Pop!_OS 22.04 and the GPU driver is 580.119.02.

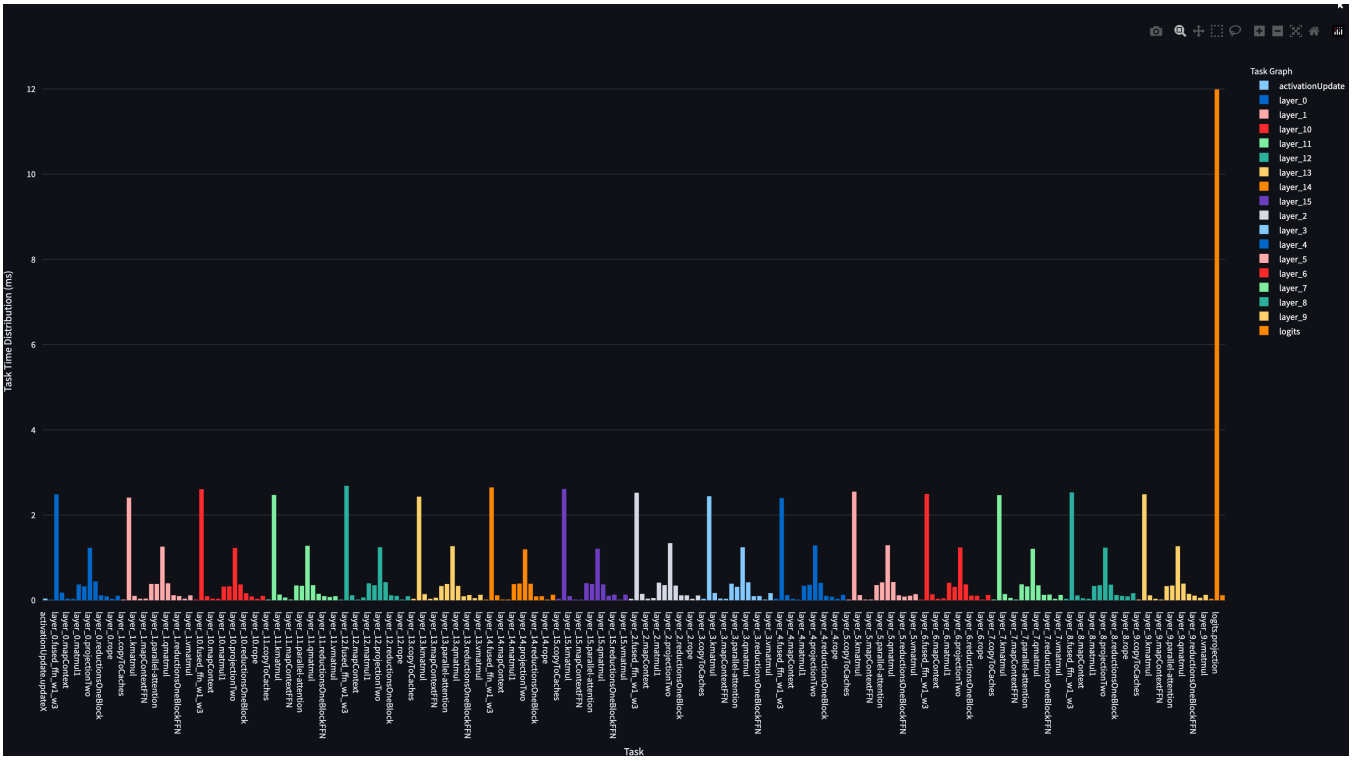

Figure 4: The distribution of the kernel execution time of each task within a TaskGraph. Each TaskGraph is depicted in different colour. This chart eases the identification of the most expensive tasks within all TaskGraphs.

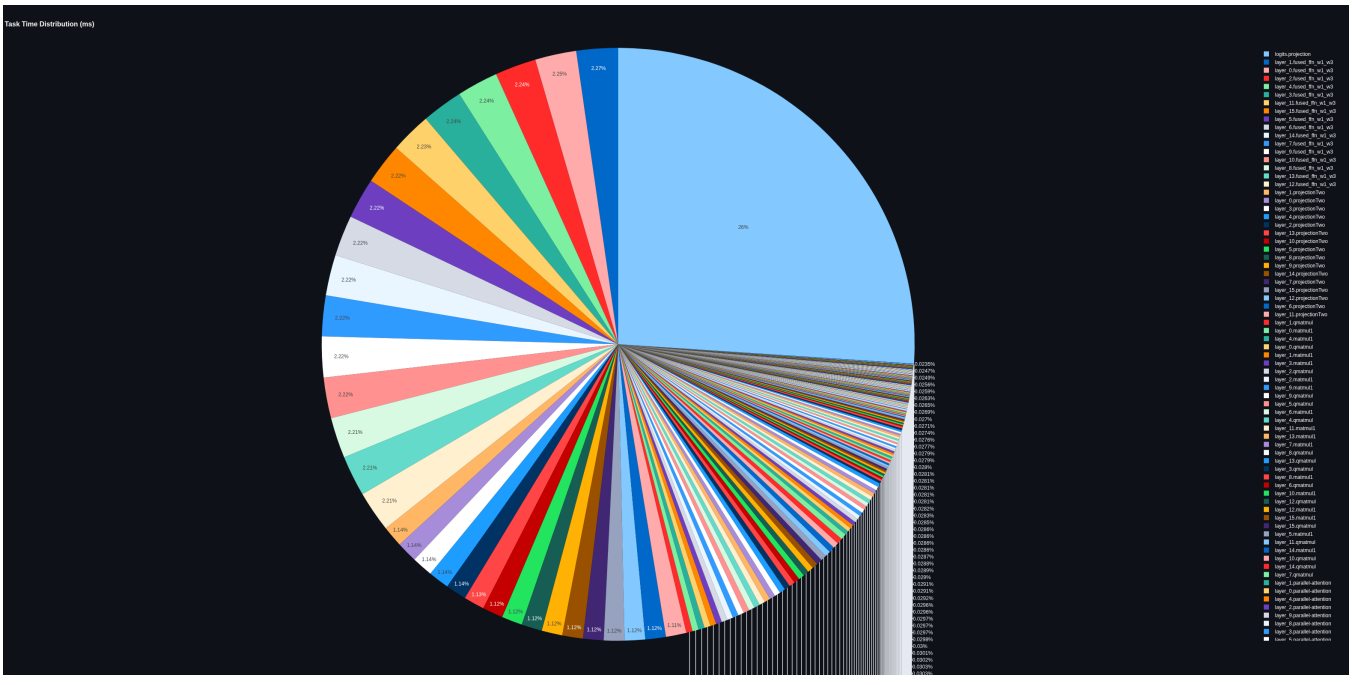

Figure 5: A pie chart showing the proportional distribution of the kernel execution time for all tasks.

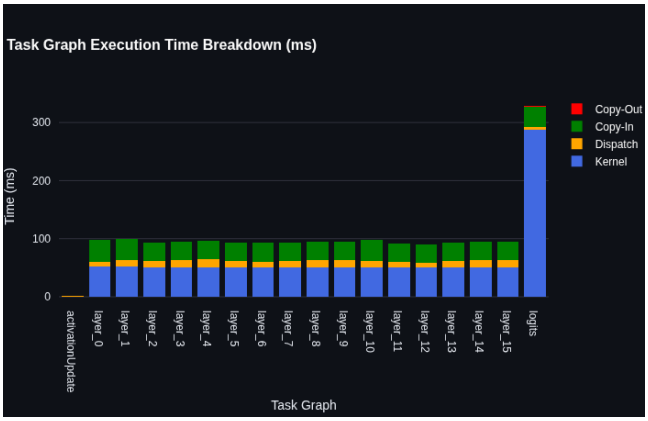

Figure 6: A breakdown analysis of the TaskGraph execution time for each TaskGraph in a stacked bar, showing the time spent in Kernel, Dispatch, Copy-In, and Copy-Out.

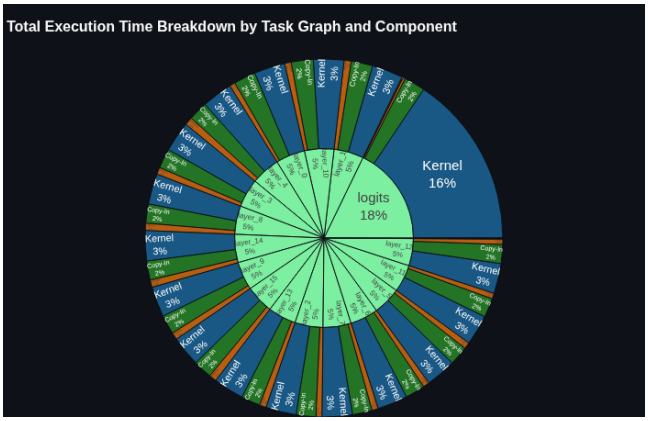

Figure 7: A sunburst chart that illustrates the outer ring that focuses on the internal view (i.e., Kernel, Dispatch, Copy-In, and Copy-Out) of a TaskGraph, while the inner ring shows the contribution of all TaskGraphs to the total execution time.

4.1 Case-study: Optimizing a Java GPU-Accelerated LLM Inference Engine.

As mentioned earlier, optimizing AI workloads under a managed runtime remains challenging due to the interplay between JIT compilation, data movement, and runtime scheduling. Driven by these challenges and to showcase the effectiveness of Pulse, we built a case study based on `GPULlama3.java` [25], an open-source Java-native implementation of Llama3 built on top of TornadoVM.

`GPULlama3.java` leverages TornadoVM's task-based programming model to express each Transformer layer as a `TaskGraph` composed of parallelizable TornadoVM tasks with explicit data dependencies. For instance, Listing 3 shows a snippet of the feed-forward

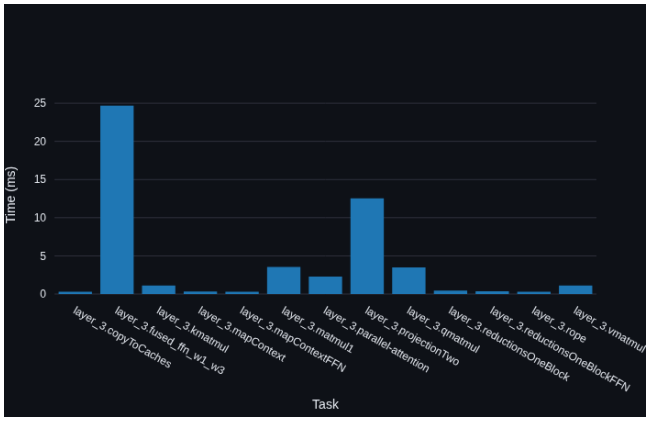

Figure 8: A bar chart showing the execution time of individual tasks within a selected TaskGraph.

operation (`fusedFeedForward- WithSiLUAndGLUActivation`), implemented as a TornadoVM task within a larger `TaskGraph`[2]. Each layer consists of multiple operations, such as normalization, attention, and feed-forward computation which are individually JIT-compiled and executed on the GPU, thereby making it an ideal application for applying profiling-guided performance optimization via Pulse.

```
1 ...
2 .task("fused_ffn_w1_w3",
      ↪ TCL::fusedFeedForwardWithSiLUAndGLUActivation,
      ↪ context,
3       state.wrapXb, state.wrapHb,
            ↪ weights.w1Layered[layerIndex],
            ↪ weights.w3Layered[layerIndex], config.dim,
            ↪ config.hiddenDim,  LOCAL_WORK_GROUP_SIZE_ALLOC)
4 ...
```

Listing 3: The definition of the fusedFeedForwardWithSiLUAndGLUActivation kernel as a TornadoVM task.

4.1.1 Profiling-Guided Transition from Fine-Grained Kernels to Fused Operators. The initial implementation of `GPULlama3.java` adopted a fine-grained design, in which each mathematical operation was encoded as an independent task. Although this modular approach simplified debugging, it incurred significant orchestration overhead due to frequent host–device synchronization points and redundant data transfers.

Pulse was used as a high-level profiling tool to diagnose these inefficiencies. As shown in Figure 9, the visualization reveals that a specific phase of the inference pipeline is dominated by the launch of a large number of short-lived GPU kernels. This execution pattern results in substantial kernel dispatch overhead and poor GPU utilization, as a significant fraction of time is spent scheduling

[2]https://github.com/beehive-lab/GPULlama3.java/blob/main/src/main/java/com/example/tornadovm/TornadoVMLayerPlanner.java

work rather than executing it, thereby limiting both instruction throughput and memory bandwidth utilization.

By exposing this behavior at task and kernel granularity, Pulse enables the identification of the exact phase responsible for the observed performance degradation without relying on low-level, vendor-specific GPU profilers. Based on this observation, the TaskGraph is restructured to reduce kernel fragmentation by fusing multiple operations into fewer GPU kernels, amortizing kernel launch overhead and reducing synchronization costs. The effect of this transformation is illustrated in Figure 10, which shows fewer, longer-running kernels and a more balanced distribution of execution time across pipeline stages.

4.1.2 Baseline implementation: fine-grained task decomposition. Listing 4 shows an early transformer layer implemented as a sequence of independent TornadoVM tasks. Each operation (normalization, projections, attention, and residual connections) is represented by its own task, resulting in numerous kernel launches and intermediate memory transfers.

4.1.3 Optimized implementation: fused operator composition. As the workflow evolved, the computation was restructured to leverage TornadoVM's support for larger fused kernels. Listing 5 shows the redesigned `TaskGraph`, where normalization, activation, and projection operations are consolidated into fewer, more efficient kernels. The feed-forward network (FFN) block, in particular, was redesigned into a single fused operator performing normalization, SiLU activation, and GLU gating in one pass. This reduces the number of tasks per layer, from 19 to 13, and decreases the synchronization overhead.

By reducing task count and avoiding unnecessary host–device communication, the optimized layout achieves lower dispatch overhead and improved memory locality.

4.1.4 Iterative optimization workflow. The feed-forward network exemplifies the iterative optimization process. Initially implemented as separate normalization, activation, and projection tasks, it was progressively fused into `fusedFeedForwardWithSiLUAndGLUActivation`. Profiling successive versions with Pulse revealed consistent reductions in data-transfer time and dispatch overhead, confirming the effectiveness of kernel fusion.

4.1.5 Quantitative results. The measurements collected on the NVIDIA RTX 3070 GPU of our testbed using the OpenCL backend demonstrate the practical gains of kernel fusion. The fused implementation results in:

- Reduced kernel launches per layer from 19 to 13,
- Lower copy time from 2.1 ms to 1.3 ms (a **38%** reduction),
- Improved overall per-layer execution time by **18.4%**.

These improvements stem from eliminating dispatch overhead and retaining intermediate results in registers or shared memory rather than global memory. This confirms that kernel fusion is a highly effective optimization strategy for GPU-accelerated workloads where orchestration and data movement dominate execution time.

4.1.6 Key takeaways. This case study demonstrates that achieving high performance for GPU-accelerated Java workloads requires a deep understanding of the interactions between the managed runtime and the underlying heterogeneous hardware. Even minor architectural decisions, such as the the granularity of which TaskGraphs are composed, or the definition of data movements, can pose measurable influence on end-to-end performance. By iteratively profiling and refining the computational graphs, developers can progressively approach native-like efficiency while retaining the productivity benefits of the Java ecosystem.

4.2 Profiling Overhead Analysis

As with any instrumentation system, Pulse introduces runtime overhead when enabled. This overhead is a direct consequence of its event-driven profiling API and its tight integration with TornadoVM's execution pipeline. In contrast to sampling-based profilers, Pulse records metrics explicitly at well-defined runtime layers, which enables accurate cross-layer attribution but introduces overhead proportional to the frequency of these events.

In practice, the observed overhead arises from three primary sources: (i) frequent timer reads and metric updates for compilation phases, kernel execution, dispatch, and data transfers; (ii) crossings between Java and native code via the Java Native Interface (JNI) when querying backend-specific events or device interfaces; and (iii) structured JSON logging of profiling data for post-mortem analysis and visualization. While each individual operation is lightweight, their cumulative cost becomes visible in workloads composed of many short-lived tasks.

We evaluated the profiling overhead of Pulse under two scenarios:

- **Low-complexity benchmark:** A microbenchmark consisting of a single `TaskGraph` executed for 100 iterations. The results show an average overhead of **1.5%** in end-to-end execution time, demonstrating that Pulse imposes negligible cost for coarse-grained workloads where the instrumentation overhead is effectively amortized over long-running kernels.
- **High-complexity application (GPULlama3.java):** The `GPULlama3.java` workload executes more than **9,000** JIT-compiled tasks to generate 50 tokens. Many of these tasks correspond to short-lived GPU kernels, each with execution times on the order of tens of microseconds. As Pulse collects multiple per-task metrics - including timing, compilation, data transfer, and power usage - and serializes each metric to JSON, the average measurement overhead across 20 runs reaches **30%**.

The latter scenario represents a worst case for fine-grained profiling, as the combination of a large number of tasks and short-lived kernel execution maximizes the frequency of profiling events relative to useful computation. For applications with fewer, coarser-grained tasks or longer-running kernels, the relative overhead is substantially lower. This behavior indicates that profiling cost scales primarily with instrumentation frequency rather than with the complexity of the Pulse infrastructure itself.

5 Related work

A wide range of profiling tools exist for both heterogeneous hardware and managed runtimes. Vendor-specific profilers, such as

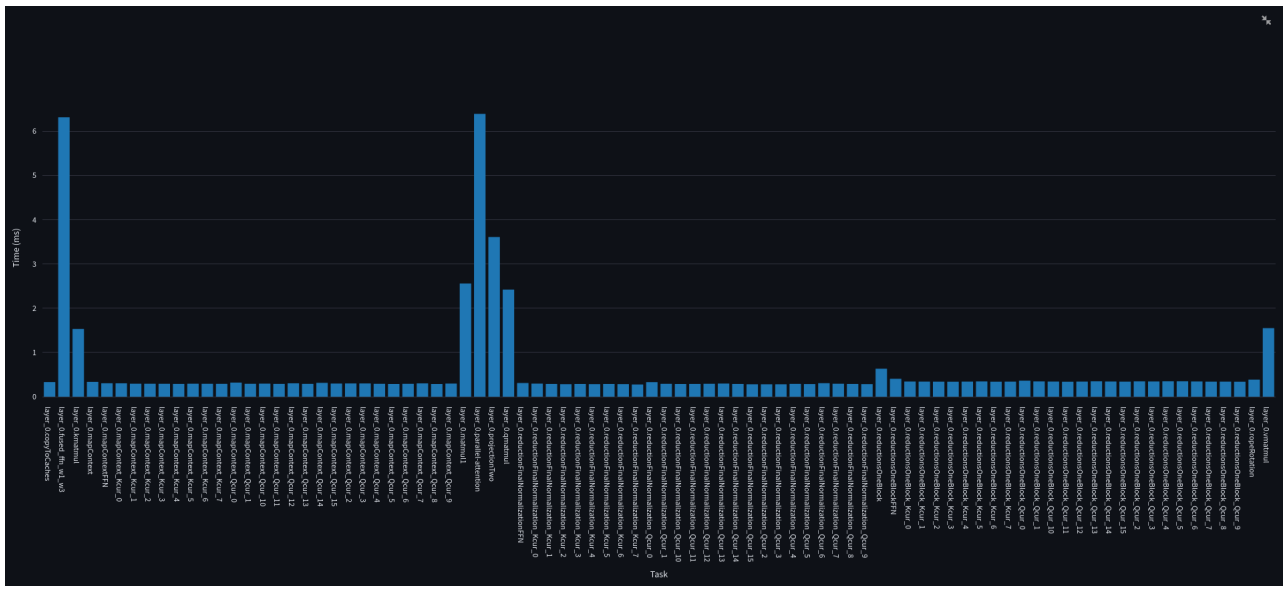

Figure 9: Fine-grained execution with many short-lived GPU kernels, resulting in high dispatch overhead and poor utilization.

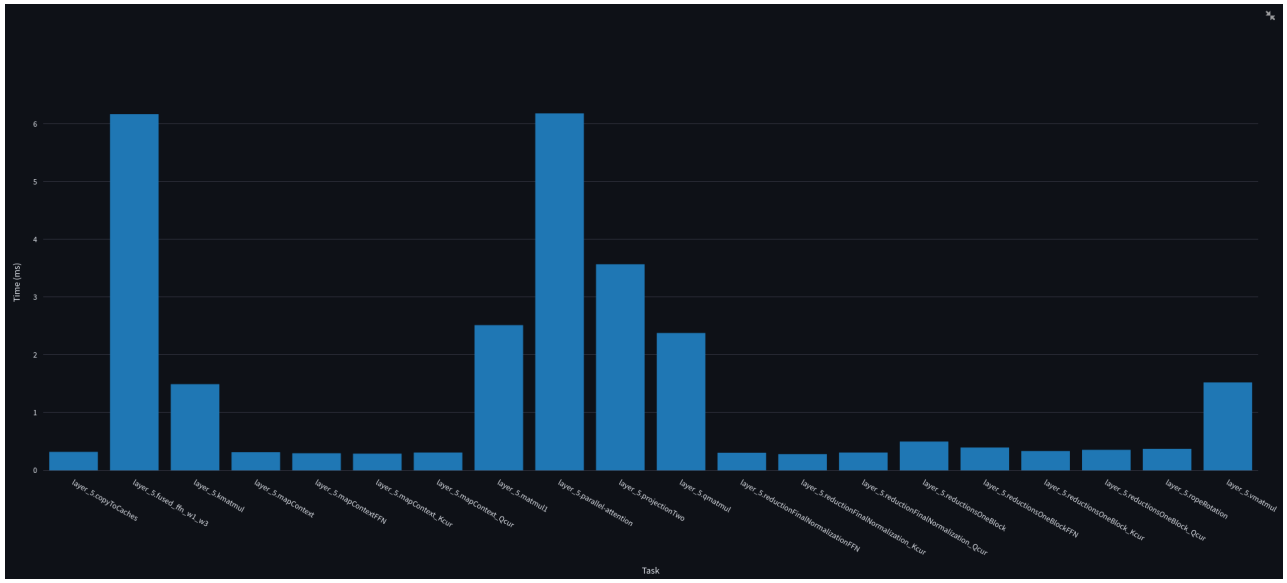

Figure 10: Optimized execution after kernel fusion, showing fewer, longer-running kernels and a more balanced execution profile.

NVIDIA Nsight Systems/Compute [8], AMD Radeon GPU Profiler [4], and Intel VTune/Graphics Performance Analyzers [2], provide detailed insights into GPU and CPU behavior. Cross-platform or API-specific tools, such as RenderDoc [12] and Apitrace [5], focus primarily on graphics or driver-level tracing. In the managed-language domain, several profiling tools target the JVM or high-level concurrency frameworks (e.g., Intel VTune for Java [15], AntTracks [21], and AkkaProf [28]). Other research has examined profiling for heterogeneous systems from perspectives, such as power-aware optimization [30] and low-overhead tracing for streaming applications [20], offering complementary insights to the goals of Pulse.

Despite this rich ecosystem, very few tools operate at the intersection of heterogeneous execution and managed programming languages. Historically, developers requiring extreme performance - particularly in the High Performance Computing domain - have favored native languages such as C/C++, reducing demand for such tools in the managed ecosystems. However, the rapid adoption of

```
TaskGraph layer = new TaskGraph("transformer_layer")
    // Normalization and projections
    .task("rmsnorm", rmsnorm)
    .task("qmatmul", matmul)
    .task("kmatmul", matmul)
    .task("vmatmul", matmul)

    // Attention computation
    .task("rope", ropeRotation)
    .task("copyToCaches", copyToCache)
    .task("parallelAttention", processHeadsParallel)

    // Output projection and residual
    .task("matmul1", matmul)
    .task("residual1", addInPlace)

    // Feed-forward network
    .task("rmsnorm_ffn", rmsnorm)
    .task("w1_project", matmul)
    .task("w3_project", matmul)
    .task("silu_activation", siluElementwiseMul)
    .task("w2_project", matmul)
    .task("residual2", addInPlace);
```

Listing 4: Baseline TaskGraph layout for a Transformer layer using fine-grained operators.

```
TaskGraph optimizedLayer = new
     ↪ TaskGraph("transformer_layer_optimized")
    // Integrated normalization
    .task("reduce_norm", reductionBlock)
    .task("map_context", normalizationMap)

    // Attention and projection
    .task("qkv_matmul", matrixVectorGeneric)
    .task("rope", ropeRotation)
    .task("copyToCaches", copyToCache)
    .task("attention", processHeadsParallel)
    .task("output_projection",
         ↪ matrixVectorGenericWithResidual)

    // Fused feed-forward block
    .task("reduce_ffn_norm", reductionBlock)
    .task("map_ffn_context", normalizationMap)
    .task("fused_ffn",
         ↪ fusedFeedForwardWithSiLUAndGLUActivation)
    .task("projection_out", matrixVectorGenericWithResidual);
```

Listing 5: Optimized TaskGraph layout with fused normalization and feed-forward operations.

LLMs has shifted this landscape. GPU-accelerated computing is now pervasive across nearly all programming languages, including managed ones, due to the substantial parallelism required for both training and inference. Regardless of the mechanism used to invoke GPU execution (native bindings, JIT compilation frameworks, or domain-specific languages), developers increasingly require profiling tools capable of bridging high-level language semantics with heterogeneous execution models. Only recently, such tools (e.g., Triton-Viz [27]) have begun to emerge.

In the context of developer tooling for heterogeneous managed runtimes, two complementary tools have recently been proposed. TornadoViz [26] is a visual analytics tool that leverages TornadoVM's internal bytecode system to provide interactive analysis of heterogeneous execution patterns and object lifecycles. Unlike Pulse, which focuses on profiling performance metrics such as kernel timing, data-transfer costs, and power consumption, TornadoViz targets the visualization of bytecode-level execution flows, task dependencies, and memory operation tracking across devices. TornadoInsight [29] addresses a different stage of the development lifecycle by providing static and dynamic code analysis for Java programs targeting heterogeneous hardware. Implemented as an IntelliJ IDEA plugin, TornadoInsight detects incompatible Java constructs before execution, offering source-linked feedback within the developer's IDE.

Pulse distinguishes itself from prior work in several important ways. First, it provides targeted support for heterogeneous managed runtime systems, addressing profiling challenges that arise from the interaction between JIT compilation, runtime orchestration, and device-level execution—an area largely unsupported by existing profilers. Second, Pulse can be parameterized to integrate with custom programming models, allowing it to become context-aware with respect to the abstractions exposed by the runtime it instruments. In this paper, we demonstrated this capability through its tight integration with TornadoVM, where concepts such as TaskGraphs, Graal-based compilation stages, and multiple backend code generators are reflected in the profiling output. At the same time, the infrastructure is sufficiently general to support emerging heterogeneous runtimes with different composability models, including systems like Project Babylon [11]. Finally, Pulse offers comprehensive support for hardware heterogeneity. While most existing profilers focus primarily on GPUs, Pulse provides a unified profiling interface for multi-core CPUs, different GPU architectures, and FPGAs, enabling cross-device analysis within a single toolchain.

6 Conclusion and Future Work

This paper presented Pulse, an integrated profiling and visualization infrastructure for managed applications running on heterogeneous systems. Pulse comprises two tightly connected components: a profiler infrastructure and an interactive visualization layer. The profiler captures a rich set of performance metrics - including execution times, data-transfer costs, compilation breakdowns, and power consumption - across TornadoVM's backends (OpenCL, PTX, SPIR-V) and hardware targets (CPUs, GPUs, FPGAs). The Pulse visualization layer complements this by transforming raw profiler output into an intuitive web-based dashboard, offering high-level summaries and detailed views that help developers identify bottlenecks, reason about resource utilization, and perform informed optimization. Together, these components make the performance behavior of complex workloads that target hardware acceleration substantially more transparent and actionable.

Looking ahead, several directions will guide the evolution of Pulse:

- **Profiler Enhancements:** Future work will expand the profiler with more granular metrics, including optional

hardware-counter information where supported by backend APIs, to provide deeper architectural insights. We also plan to explore techniques for reducing profiling overhead, particularly for workloads dominated by short-lived kernels. A major goal is to achieve tighter correlation between device-side metrics and JVM-level events (e.g., from Java Flight Recorder [1] or AsyncProfiler [6]), enabling a unified end-to-end view that captures interactions between the managed runtime and heterogeneous hardware.

- **Visualizer Enhancements:** The visualization layer will be extended to support comparative analysis across multiple profiles, enabling developers to contrast performance across code revisions, hardware configurations, or tuning scenarios. Additional planned features include automated insight generation (e.g., anomaly detection or bottleneck suggestions), integration with development environments, and improved customization through user-defined visualizations or plugins.

Acknowledgments

This work is supported by the European Union's Horizon Europe programme under grant agreements No 101070670 (ENCRYPT), No 101092850 (AERO), No 101093069 (P2CODE), and No 101070052 (TANGO). This work is also funded by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee (ENCRYPT: 10039809; AERO: 10048318; P2CODE: 10048316; TANGO: 10039107).

References

[1] 2020. Java Flight Recorder. https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm. Accessed: 2026-02-20.

[2] 2021. Intel VTune Profiler. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html. Accessed: 2026-02-20.

[3] 2024. OpenJDK. https://openjdk.org/. Open-source implementation of the Java Platform, Standard Edition.

[4] 2025. AMD Radeon GPU Profiler. https://gpuopen.com/rgp/. Accessed: 2026-02-20.

[5] 2025. Apitrace. https://apitrace.github.io. Accessed: 2026-02-20.

[6] 2025. Async Profiler. https://github.com/async-profiler/async-profiler. Accessed: 2026-02-20.

[7] 2025. Enable/Disable the Profiler using the TornadoExecutionPlan. https://tornadovm.readthedocs.io/en/latest/profiler.html#enable-disable-the-profiler-using-the-tornadoexecutionplan. Accessed: 2026-02-20.

[8] 2025. NVIDIA Insight Systems. https://developer.nvidia.com/nsight-systems. Accessed: 2026-02-20.

[9] 2025. NVIDIA Management Library, NNML. https://developer.nvidia.com/management-library-nvml. Accessed: 2026-02-20.

[10] 2025. OneAPI, Level Zero Sysman API. https://oneapi-src.github.io/level-zero-spec/level-zero/1.11/sysman/api.html. Accessed: 2026-02-20.

[11] 2025. OpenJDK Project Babylon. https://openjdk.org/projects/babylon/. Accessed: 2026-02-20.

[12] 2025. Renderdoc. https://renderdoc.org. Accessed: 2026-02-20.

[13] 2025. Streamlit.io. https://streamlit.io. Accessed: 2026-02-20.

[14] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Luján. 2018. Exploiting High-Performance Heterogeneous Hardware for Java Programs Using Graal. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang '18)*. Association for Computing Machinery, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/3237009.3237016

[15] Jack Donnell. 2022. Java Performance Profiling using the VTune™ Performance Analyzer. https://www.intel.com/content/dam/develop/external/us/en/documents/219355-vtune-performance-profiling-178112.pdf.

[16] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. 2019. Dynamic Application Reconfiguration on Heterogeneous Hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19)*. Association for Computing Machinery, New York, NY, USA, 165–178. https://doi.org/10.1145/3313808.3313819

[17] Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. 2024. Programming Heterogeneous Hardware Via Managed Runtime Systems. In *Programming Heterogeneous Hardware via Managed Runtime Systems*. Springer, 71–125.

[18] Wilhelm Hasselbring and André van Hoorn. 2020. Kieker: A monitoring framework for software engineering research. *Software Impacts* 5 (2020). https://doi.org/10.1016/j.simpa.2020.100019

[19] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. 2017. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*. Association for Computing Machinery, New York, NY, USA, 74–82. https://doi.org/10.1145/3050748.3050764

[20] Joseph Lancaster. 2011. *Low-Impact Profiling of Streaming, Heterogeneous Applications*. Ph. D. Dissertation. Washington University in St. Louis. https://doi.org/10.7936/K7K935KF

[21] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) *(ICPE '15)*. Association for Computing Machinery, New York, NY, USA, 51–62. https://doi.org/10.1145/2668930.2688037

[22] OpenTelemetry Authors. 2019. OpenTelemetry Documentation. https://opentelemetry.io. Accessed: 2026-02-20.

[23] Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. 2019. Towards Prototyping and Acceleration of Java Programs onto Intel FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 310–310. https://doi.org/10.1109/FCCM.2019.00051

[24] Michail Papadimitriou, Eleni Markou, Juan Fumero, Athanasios Stratikopoulos, Florin-Gabriel Blanaru, and Christos-Efthymios Kotselidis. 2021. Multiple-Tasks on Multiple-Devices (MTMD): Exploiting Concurrency in Heterogeneous Managed Runtimes. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '21)*. Association for Computing Machinery, New York, NY, USA, 125–138. https://doi.org/10.1145/3453933.3454019

[25] Michail Papadimitriou, Mary Xekalaki, Juan Fumero, Athanasios Stratikopolos, Orion Papadakis, and Christos Kotselidis. 2025. *GPULlama3.java*. https://github.com/beehive-lab/GPULlama3.java

[26] Michail Papadimitriou, Maria Xekalaki, Athanasios Stratikopoulos, Orion Papadakis, Juan Fumero, and Christos Kotselidis. 2025. TornadoViz: Visualizing Heterogeneous Execution Patterns in Modern Managed Runtime Systems. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Singapore, Singapore) *(MPLR '25)*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/3759426.3760978

[27] Tejas Ramesh, Alexander Rush, Xu Liu, Binqian Yin, Keren Zhou, and Shuyin Jiao. 2025. Triton-Viz: Visualizing GPU Programming in AI Courses. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1* (Pittsburgh, PA, USA) *(SIGCSETS 2025)*. Association for Computing Machinery, New York, NY, USA, 952–958. https://doi.org/10.1145/3641554.3701795

[28] Andrea Rosà, Lydia Y. Chen, and Walter Binder. 2016. AkkaProf: A Profiler for Akka Actors in Parallel and Distributed Applications. In *Programming Languages and Systems*, Atsushi Igarashi (Ed.). Springer International Publishing, Cham, 139–147.

[29] Athanasios Stratikopoulos, Tianyu Zuo, Umut Sarp Harbalioglu, Juan Fumero, Michail Papadimitriou, Orion Papadakis, Maria Xekalaki, and Christos Kotselidis. 2025. Dynamic and Static Code Analysis for Java Programs on Heterogeneous Hardware. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Singapore, Singapore) *(MPLR '25)*. Association for Computing Machinery, New York, NY, USA, 107–113. https://doi.org/10.1145/3759426.3760984

[30] Kuen Hung Tsoi and Wayne Luk. 2011. Power profiling and optimization for heterogeneous multi-core systems. *SIGARCH Comput. Archit. News* 39, 4 (Dec. 2011), 8–13. https://doi.org/10.1145/2082156.2082159

[31] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd Joint ACM/SPEC International Conference on Performance Engineering (ICPE '12)*. ACM, 247–248. https://doi.org/10.1145/2188286.2188326

[32] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581