TornadoViz: Visualizing Heterogeneous Execution Patterns in Modern Managed Runtime Systems

Michail Papadimitriou
The University of Manchester
United Kingdom

Maria Xekalaki
The University of Manchester
United Kingdom

Athanasios Stratikopoulos
The University of Manchester
United Kingdom

Orion Papadakis
The University of Manchester
United Kingdom

Juan Fumero
The University of Manchester
United Kingdom

Christos Kotselidis
The University of Manchester
United Kingdom

Abstract

With the increasing prevalence of machine learning and large language model (LLM) inference, heterogeneous computing has become essential. Modern JVMs are embracing this transition through projects such as TornadoVM and Babylon, which enable hardware acceleration on diverse hardware resources, including GPUs and FPGAs. However, while performance results are promising, developers currently face a significant tooling gap: traditional profilers excel at CPU-bound execution but become a "black box" when execution transitions to accelerators, providing no visibility into device memory management, execution patterns or cross-device data movement. This gap leaves developers without a unified view of how their Java applications behave across the heterogeneous computing stack.

In this paper, we present TornadoViz, a visual analytics tool that leverages TornadoVM's specialized bytecode system to provide interactive analysis of heterogeneous execution and object lifecycles in managed runtime systems. Unlike existing tools, TornadoViz bridges the managed-native divide by interpreting the bytecode stream that orchestrates heterogeneous execution, hence connecting high-level application logic with low-level hardware utilization patterns. Our tool enables developers to visualize task dependencies, track memory operations across devices, analyze bytecode distribution patterns, and identify performance bottlenecks through interactive dashboards.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**.

ACM Reference Format:
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 (MPLR '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3759426.3760978

MPLR '25, Singapore, Singapore

ACM ISBN 979-8-4007-2149-6/25/10
https://doi.org/10.1145/3759426.3760978

1 Introduction

As computing systems continue to incorporate diverse hardware accelerators such as GPUs, FPGAs, and specialized processors, managed programming languages face the challenge of efficiently utilizing these heterogeneous resources while maintaining the productivity and safety benefits their runtime systems provide. Projects like TornadoVM [7, 9] and Babylon [6] address this challenge by extending Java with the capability to dynamically compile and execute code on heterogeneous hardware devices transparently.

However, developers currently face a significant tooling gap when analyzing modern heterogeneous managed applications. Traditional JVM profiling tools such as Java Flight Recorder [1], VisualVM [2], and JProfiler [8] excel at analyzing CPU-bound execution within the managed runtime but provide no visibility into accelerator utilization, device memory management, or cross-device data movement. In contrast, vendor-specific GPU profiling tools like NVIDIA Nsight [5], Intel VTune [4], and AMD Radeon Profiler [3] offer detailed hardware insights but operate at the native code level, disconnected from the high-level managed application context. This disconnect leaves developers without a unified view of how their Java applications actually behave across the heterogeneous computing stack.

A key innovation of TornadoVM [9, 10] is its specialized bytecode system that serves as an orchestration layer between the JVM and heterogeneous devices. These bytecodes capture the complete execution of heterogeneous applications - from task scheduling, execution and memory allocation to data transfers and device synchronization. Unlike standard Java bytecodes that represent computational operations, TornadoVM bytecodes encode the runtime decisions and resource management operations that govern heterogeneous execution. This bytecode stream provides a view into how managed applications actually utilize heterogeneous

hardware, revealing patterns invisible to traditional profiling approaches.

However as the complexity of applications and hardware configurations increases, developers face significant challenges in understanding:

1. How tasks are scheduled and executed across multiple devices.
2. How data flows between the host and various accelerators.
3. When and why specific bytecode operations occur and what they reveal about runtime behavior.
4. How memory is allocated, transferred, and deallocated across the memory hierarchy.
5. What optimization opportunities exist at the intersection of managed code and heterogeneous hardware.

These challenges become particularly acute when diagnosing performance issues, memory bottlenecks, or unexpected behavior in heterogeneous environments, where the root cause may span both managed runtime decisions and low-level hardware interactions.

In this work, we introduce TornadoViz[1], an open-source interactive tool designed to bridge this tooling gap by leveraging TornadoVM's bytecode execution traces to provide a unified view of heterogeneous managed application behavior. By interpreting and visualizing the bytecode stream that orchestrates heterogeneous execution, our tool connects high-level application logic with low-level hardware utilization patterns. Built on modern web technologies and by leveraging interactive data visualization techniques, the tool enables developers to:

- Visualize task graph dependencies and execution patterns across devices.
- Track memory operations and object lifecycles throughout the heterogeneous memory hierarchy.
- Analyze bytecode distribution patterns to gain insight into runtime behavior.
- Identify performance bottlenecks and optimization opportunities that span managed and native execution.

2 Background

2.1 TornadoVM Overview

TornadoVM [7, 9, 10] is a plug-in to OpenJDK and GraalVM that allows programmers to accelerate Java programs on heterogeneous hardware. It utilizes a task-based programming model where developers identify parallelizable tasks that can be offloaded to accelerators. TornadoVM dynamically compiles these tasks into platform-specific code, such as OpenCL, PTX, and SPIR-V, facilitating execution across a diverse range of hardware, including multi-core CPUs, GPUs, and FPGAs.

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

Table 1. The TornadoVM bytecode operations.

Category	Bytecode	Description
Context	`BEGIN`	Mark the start of execution context for a device
	`END`	Mark the end of execution context
Allocation	`ALLOC`	Allocate memory buffer on target device
	`DEALLOC`	Free memory buffer on device
Data Transfer	`TRANSFER_HOST_TO_DEVICE_ONCE`	Send data from host to device (once, blocking)
	`TRANSFER_HOST_TO_DEVICE_ALWAYS`	Send data from host to device (always, blocking)
	`TRANSFER_DEVICE_TO_HOST_ALWAYS _BLOCKING`	Send data from device to host (always, blocking)
Execution	`LAUNCH`	Execute kernel/task on target device
	`BARRIER`	Synchronize execution across events
State Management	`ON_DEVICE`	Verify object presence on device
	`PERSIST`	Mark object for persistence on device

2.2 TornadoVM Bytecode System

At the core of TornadoVM's execution model is a specialized bytecode system that functions as an intermediary layer between standard Java bytecode and the generation of device-specific machine code. These bytecodes encapsulate high-level operations essential for heterogeneous computing, such as memory allocation, on-device object states, data transfers between the host and the device, kernel execution, and synchronization. Then the TornadoVM bytecode interpreter processes these bytecodes to orchestrate the entire execution flow across the various hardware devices.

Table 1 showcases the full set of TornadoVM bytecodes. These bytecodes are split into five main categories: `Context`, `Allocation`, `Data Transfers`, `Execution` and `State Management`. The analysis of the patterns and interdependencies within these bytecode operations is fundamental to optimize applications for heterogeneous environments. However, the complexity of these patterns escalates significantly with an increase in the number of tasks, the involvement of multiple devices, and intricate data dependencies, rendering manual analysis impractical.

Listing 1 highlights the rich meta-data embedded in the TornadoVM bytecodes through an execution trace of a vector addition application. This example showcases two subsequent `TaskGraphs`, `s0` and `s1` perform vector additions on an NVIDIA GeForce RTX 3070. This execution trace reveals several important aspects of TornadoVM's bytecode richness:

Object Lifecycle Tracking: Each memory object is assigned a unique hash code (e.g., `0x289710d9`) that persists throughout its lifecycle, enabling precise tracking of object allocation, transfers, persistence decisions, and deallocation across multiple execution contexts. **Resource Management Metadata:** Operations include detailed resource information such as memory sizes (`size=88`), batch configurations

```
Interpreter instance running bytecodes for: [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070
bc: BEGIN
bc: ALLOC IntArray@289710d9 on NVIDIA RTX 3070, size=88, batchSize=0
bc: TRANSFER_HOST_TO_DEVICE_ONCE [Hash:0x289710d9] IntArray@289710d9 size=88, offset=0  [Status: Transferred]
bc: LAUNCH task s0.t0 - add on NVIDIA RTX 3070 [event list=0]
bc: PERSIST [0x289710d9] IntArray@289710d9 on NVIDIA RTX 3070
bc: DEALLOC [0x5dd1c9f2] IntArray@5dd1c9f2 [Status: Freed]
bc: BARRIER event-list 3
bc: END
Interpreter instance running bytecodes for: [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070
bc: BEGIN
bc: ON_DEVICE [0x289710d9] IntArray@289710d9 on NVIDIA RTX 3070
bc: LAUNCH task s1.t1 - add on NVIDIA RTX 3070 [event list=0]
bc: TRANSFER_DEVICE_TO_HOST_ALWAYS_BLOCKING [0x324a0017] IntArray@324a0017 size=88, offset=0
bc: DEALLOC [0x324a0017] IntArray@324a0017 [Status: Persisted]
bc: END
```

Figure 1. An example of the TornadoVM bytecode execution trace.

(`batchSize=0`), and memory offsets (`offset=0`), providing complete visibility into resource utilization patterns. **Execution State Information:** Each operation records its execution status (`Status: Transferred`, `Status: Freed`, `Status: Persisted`) and event dependencies (`event list=0`), enabling precise reconstruction of execution flow and synchronization behavior. **Cross-Context Optimization Tracking:** The trace shows how TornadoVM optimizes across execution contexts—objects persisted in the first context (`PERSIST [0x289710d9]`) are efficiently reused in the second context (`ON_DEVICE [0x289710d9]`), avoiding redundant data transfers. **Device-Specific Information:** Each operation includes complete device identification and configuration details, enabling analysis of multi-device execution patterns and load distribution.

This rich metadata embedded in TornadoVM bytecodes provides a comprehensive view of heterogeneous execution that is unavailable through traditional profiling approaches. However, extracting meaningful insights from such complex, multi-dimensional execution traces requires sophisticated visualization and analysis techniques. The temporal relationships between operations, object lifecycle patterns, memory usage trends, and cross-device dependencies create a complex web of information that is difficult to analyze manually, particularly for applications with multiple tasks, large datasets, and complex execution flows.

3 Use Case: Visualizing heterogeneous execution of GPULlama3.java

To demonstrate the effectiveness of TornadoViz, we set out to analyze the traces of a real-world application that uses TornadoVM to accelerate its execution. For this purpose we used GPULlama3.java [11], an open-source implementation of the Llama3.java [12] that accelerates LLM inference in pure Java. GPULlama3.java is a complex AI inference framework that supports the GPU execution of multiple models such as Llama3, Mistral, etc.

Table 2 shows that while the fundamental structure of the GPULlama3.java application remains constant - with the same number of task-graphs and defined tasks - the number of executed tasks skyrockets from 12,950 for a small prompt to 54,131 for a large one. This explosion in dynamic operations directly translates into a more than six-fold increase in the size of the log file. Faced with thousands of lines of dense, multi-dimensional trace data, a developer cannot be expected to manually identify performance bottlenecks, track memory dependencies, or understand the intricate temporal relationships between tasks. This evidence powerfully motivates the argument that sophisticated, automated tools to parse, analyze, and visualize these complex logs are not merely a convenience but an absolute necessity to harness the full potential of the rich performance metadata that TornadoVM provides.

3.1 Memory Objects Timeline Analysis

Listing 1 showcases that the bytecodes include the unique hashes for objects as assigned in the JVM. TornadoViz uses the JVM's `System.identityHashCode` (32-bit) to identify objects. Collisions are rare in typical TornadoVM workloads, as the number of concurrently alive GPU objects is far below problematic thresholds. Therefore, the `Memory Objects Lifecycle analysis` module on TornadoViz utilizes this information, which is already present in the bytecodes, to create a visual timeline of the object lifecycle.

Figure 2 showcases the `Memory Object Timeline` view. The figure shows each of the objects used by any given application, tracks when each object was allocated, deallocated, and transferred to or from the device. At the top part of the figure, the names of each individual `TaskGraph` are listed, while the bottom part lists the names of the executed `Tasks`. Through this layout, the timeline of each object can be tracked based on the status kept while different `TaskGraphs` were executed.

To manage to work with oversized log files, TornadoViz provides interactive analysis to narrow down the scope of the views with the following two components:

Table 2. Key metrics from generated logs for prompts of varying token lengths.

Size	Taskgraphs	Tasks	Objects	Executed Tasks	Executed Bytecodes	Log-file (LOCs)
Small	18	259	31	12950	1230	838
Medium	18	259	31	28490	2583	1732
Large	18	259	31	54131	4907	5743

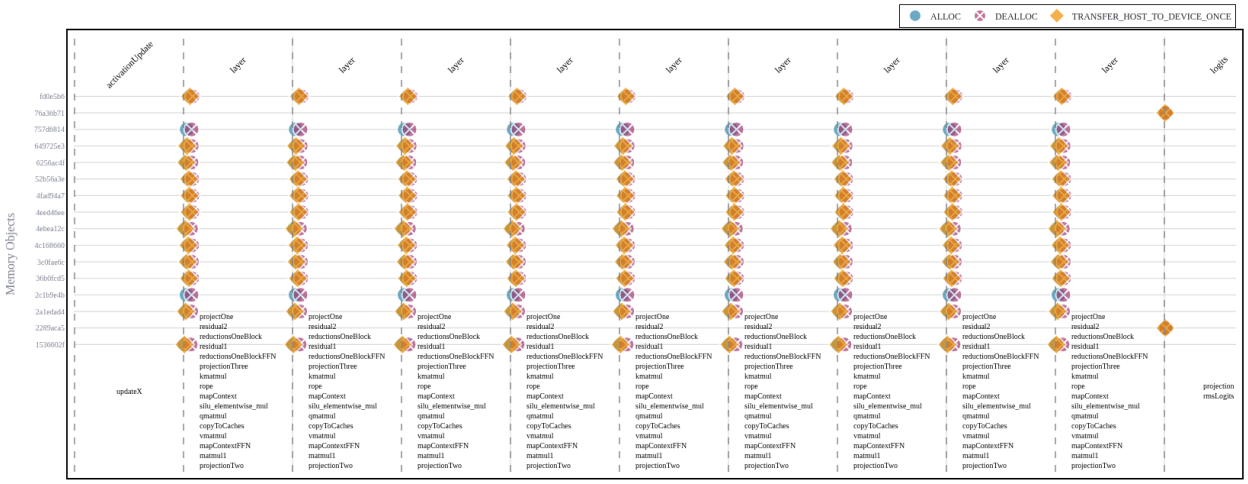

Figure 2. Memory Object Timeline in TornadoViz with the bytecodes obtained when running the GPULlama3.java application.

1. **Visualization Controls**: include several controls for filtering and analysis:
   - Minimum Memory Size: Filter objects below a certain memory threshold.
   - Time Scaling: Adjust temporal resolution (i.e., Linear, Log, Dense scaling shown).
   - Simplified View: Toggle that hides temporary objects and compress repetitive patterns.
   - Show Task Names: Toggle visibility of task names at the bottom.
2. **Subregion Selection Controls**: include options to zoom in/out or select specific regions of the view. One can select a single or multiple `TaskGraphs` to analyze specific behavior of the objects associated with them. For example, Figure 3 depicts the state of object between two different `TaskGraphs`.

3.2 Memory Usage Over Time Analysis

Memory management is a critical aspect of GPU acceleration, particularly when dealing with large-scale applications that may experience varying memory pressure throughout execution. To address this challenge, TornadoViz provides a comprehensive `Memory Usage Over Time` analysis module that enables developers to monitor and understand the memory footprint of their applications on the accelerator device. Figure 4 illustrates the memory usage profile captured during the execution of the GPULlama3.java application. The visualization reveals distinct phases of memory allocation and usage patterns, with the most notable characteristic being the sharp initial increase in memory consumption followed by sustained plateau periods. This pattern suggests an initial allocation phase where the application loads data structures and model parameters onto the GPU memory, followed by stable execution phases where memory usage remains relatively constant. The interactive dashboard provides several key insights for performance optimization. First, it identifies peak memory usage periods, allowing developers to understand the maximum memory pressure their applications exert on the accelerator. In this example, the memory usage rapidly escalates from near-zero to approximately 3.5-4.0 GB during the initial execution phase, then maintains this level throughout the remaining execution time with a slight increase toward the end. Second, the tool enables granular analysis of how different `TaskGraphs` contribute to the overall memory footprint. The timeline visualization shows discrete allocation events (represented by the orange markers) that correspond to specific task execution phases. This granular view allows developers to identify which computational tasks are the most memory-intensive and optimize accordingly. The sustained memory usage pattern observed in Figure 4 is typical of inference workloads where model weights and intermediate tensors remain resident in GPU memory throughout execution. The slight uptick at the end may indicate additional allocations for output processing or

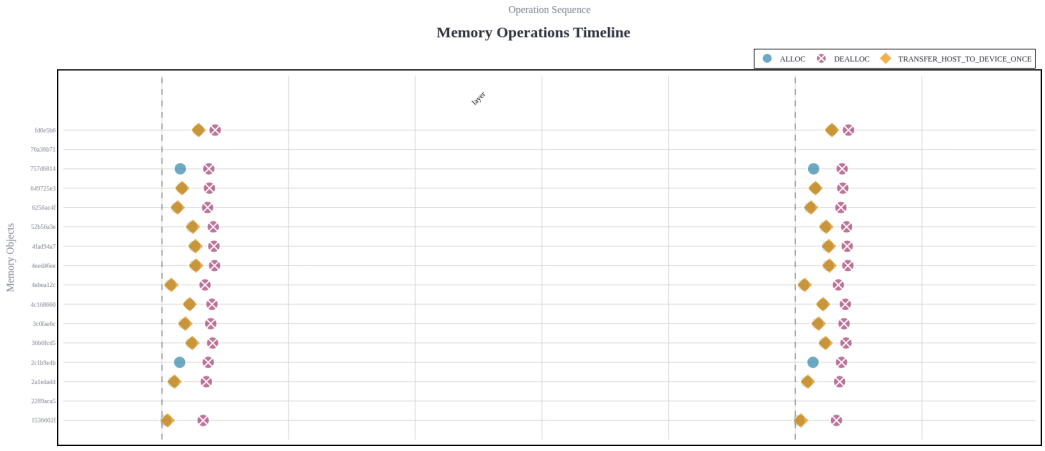

Figure 3. Example of using subregion selection of Figure 2.

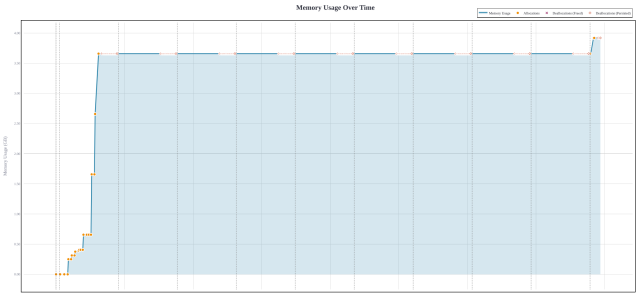

Figure 4. Overview of the `Memory Usage Over Time` dashboard, showing how memory usage on the device grows over time when different `TaskGraphs` execute.

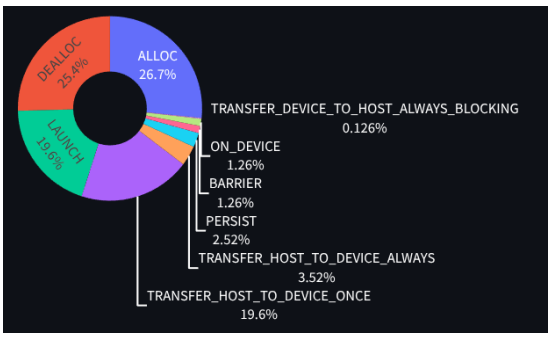

Figure 5. Distribution of bytecode operations from the GPULlama3.java application.

cleanup operations. This type of analysis is invaluable for understanding memory bottlenecks, optimizing batch sizes, and ensuring efficient resource utilization in production deployments.

3.3 Bytecode Detailed Analysis

Understanding the distribution and characteristics of bytecode operations is essential for optimizing GPU-accelerated Java applications. TornadoViz provides comprehensive bytecode analysis capabilities that enable developers to identify performance bottlenecks and memory allocation patterns at the operation level. Also, it provides fine-grained granularity were one can track the history of single object and which bytecodes are associated with it.

3.3.1 Bytecode Allocation. Analyzing the distribution of the bytecode operations can reveal the frequency and types of operations observed during the GPU acceleration. Figure 5 presents the distribution profile for the GPULlama3.java application. The pie chart analysis reveals several key insights about the application's computational characteristics. ALLOC is the most dominant operation of the execution profile, taking 26.7% of the overall bytecode operations. Subsequently, the DEALLOC operations account for approximately 25.0% of all bytecode operations, indicating significant memory management overhead. The small difference between the ALLOC and DEALLOC percentages suggests a balanced allocation/deallocation pattern typical of memory-intensive applications. The presence of LAUNCH operations (visible in smaller percentages) corresponds to kernel launch activities, while TRANSFER operations represent the data movements between host and device memory spaces. This distribution pattern is characteristic of deep learning inference workloads, where frequent memory allocations and deallocations occur for intermediate tensor computations. The relatively high percentage of memory management operations (ALLOC and DEALLOC combined representing over 50% of operations) suggests that memory optimization strategies could yield significant performance improvements.

3.3.2 Bytecode Operation Details. Beyond the high-level distribution analysis, TornadoViz provides granular operation-level insights through its detailed operation viewer. Figure 6 presents the comprehensive operation details interface, which allows developers to examine individual operations within specific `TaskGraphs` and understand their memory

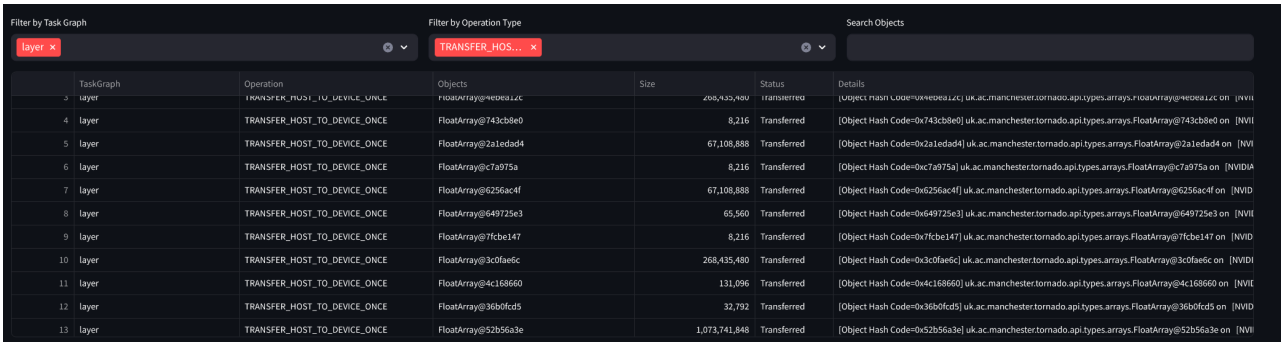

Figure 6. Bytecode operation analysis interface showing individual operations and their objects. The table provides information about TaskGraph assignment, operation types, object details, memory sizes, and GPU device mapping for each bytecode.

footprint and execution characteristics. The detailed analysis interface provides several critical pieces of information for each operation. The `TaskGraph` column identifies which computational graph each operation belongs to, enabling developers to trace operations back to specific algorithmic components. In the GPULlama3.java example, operations are primarily distributed between "activationUpdate" and "layer" `TaskGraphs`, corresponding to neural network forward pass computations. The Operation column specifies the exact bytecode operation type (ALLOC, TRANSFER_HOST_TO_DEVICE_ALW, LAUNCH, PERSIST, BARRIER, ON_DEVICE), providing insight into the computational and memory management patterns. The Objects column reveals the data structures being manipulated, with FloatArray objects of various sizes being the primary data types, consistent with neural network tensor operations. The Size column provides information about the memory footprint of each object. Knowing the size at the object level enables developers to identify memory-intensive operations and optimize their data layout accordingly. The Status column provides execution state information, while the Details column offers comprehensive information about the specific objects and their locations on the GPU device (NVIDIA CUDA - NVIDIA GeForce RTX 3070 in this example). This level of detail is essential for debugging memory-related issues and understanding the mapping of Java objects to GPU memory spaces. The filtering capabilities (Filter by `TaskGraph`, Filter by Operation Type, and Search Objects) enable developers to focus their analysis on specific aspects of the application, making it easier to identify bottlenecks in large-scale applications with thousands of operations.

4 Conclusion & Future Work

This paper presented TornadoViz, an interactive visual analytics tool that addresses the critical tooling gap in heterogeneous managed runtime systems. By leveraging TornadoVM's specialized bytecode system, TornadoViz provides developers with unprecedented visibility into the execution patterns, memory management, and resource utilization of GPU-accelerated Java applications. Our tool successfully bridges the divide between high-level managed application logic and low-level hardware utilization patterns. The evaluation using GPULlama3.java demonstrates the tool's effectiveness in analyzing complex heterogeneous workloads, revealing insights such as memory allocation patterns that dominate execution profiles and cross-device optimization opportunities that would be invisible to traditional profiling approaches. TornadoViz enables developers to identify performance bottlenecks, optimize memory management strategies, and understand the intricate relationships between managed runtime decisions and heterogeneous hardware utilization.

Future work includes real-time monitoring to enable developers to observe heterogeneous applications during execution with live performance analysis and immediate bottleneck identification. Also, integration with development environments, particularly IntelliJ IDEA, would embed heterogeneous performance analysis directly into the coding workflow, making optimization insights accessible during development. Finally, merging TornadoViz with traditional profiling tools would create a unified analysis platform that combines the deep heterogeneous insights of bytecode-level analysis with conventional CPU profiling metrics.

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: 2024-02-15.

[2] 2020. VisualVM: All-in-One Java Troubleshooting Tool. https://visualvm.github.io/. Accessed: 2024-02-15.

[3] 2021. AMD Radeon GPU Profiler. https://gpuopen.com/rgp/. Accessed: 2024-02-15.

[4] 2021. Intel VTune Profiler. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html. Accessed: 2024-02-15.

[5] 2025. NVIDIA Insight Systems. https://developer.nvidia.com/nsight-systems. Accessed: 2025-05-15.

[6] 2025. OpenJDK Project Babylon. https://openjdk.org/projects/babylon/. Accessed: 2025-05-15.

[7] 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

[8] ej-technologies GmbH. 2025. JProfiler. https://www.ej-technologies.com/jprofiler. Accessed: 2025-06-21.

[9] 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

[10] Michail Papadimitriou, Eleni Markou, Juan Fumero, Athanasios Stratikopoulos, Florin Blanaru, and Christos 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* (Virtual, USA) *(VEE 2021)*. Association for Computing Machinery, New York, NY, USA, 125–138. https://doi.org/10.1145/3453933.3454019

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

[12] Alfonso Peterssen. 2024. *llama3.java: A pure Java implementation for Llama3.* https://github.com/mukel/llama3.java

Received 2025-06-23; accepted 2025-07-28