SafeRustCL: A Zero-Cost Typed-State Wrapper for OpenCL

Anonymous Author(s)

Abstract

We present *SafeRustCL*, a lightweight Rust library that enforces correct OpenCL buffer and command-queue usage via phantom-typed state transitions (Queued → InFlight → Ready) at compile time. Our wrapper adds only 1% to 3% (worst case) overhead compared to raw `opencl3` bindings, while eliminating entire classes of unsafe API misuse. To aid performance analysis, we integrate an optional "memtrace" feature generating H2D–Kernel–D2H CSV timelines, and a "metrics" feature reporting μs-precision latencies. We evaluate on an RTX 3090 across a 4-point stencil kernel (4 MiB–3850 MiB), showing consistent sequential execution. Source code is released open-source.

1 Introduction

1.1 From CPU-Only Systems to Heterogeneous Architectures

Pure CPU platforms once dominated high-performance workloads, but the landscape has shifted decisively toward heterogeneous systems. Graphics Processing Units—originally conceived as graphics accelerators—now constitute a cornerstone of artificial-intelligence (AI) and high-performance-computing (HPC) infrastructures. Modern AI models leverage thousands of GPU cores to accelerate matrix multiplications and neural-network inference by orders of magnitude[6], a capability unattainable on CPU-only systems. In parallel, FPGAs and ASICs offer specialized acceleration for niche workloads. These trends underscore the need for type-safe, zero-cost abstractions to prevent low-level API errors in heterogeneous environments.

These developments increase both performance potential and programming complexity. Large systems integrate CPUs for sequential control flow, GPUs for data-parallel kernels, and reconfigurable fabric or ASICs for fixed-function acceleration. Harnessing such diversity demands programming models that eliminate runtime errors at compile time while retaining the flexibility to schedule work across heterogeneous resources. Without stronger safety guarantees, developers frequently resort to low-level APIs and unsafe foreign-function interfaces, introducing subtle bugs that are difficult to detect and diagnose. Recent surveys stress that future heterogeneous platforms will succeed only if security and correctness can be enforced as first-class concerns across CPU, GPU, FPGA and ASIC components[16]. SafeRustCL's design—static typestate checks combined with zero-cost abstractions—directly addresses this requirement by catching mis-sequenced API calls before deployment, paving the way for reliable, performant GPU computing in ever more diverse architectures. Also in heterogeneous programming, C-based GPU APIs inherit all memory-safety pitfalls of C—use-after-free, pointer aliasing, and missing synchronization. These issues motivate our zero-cost, type-safe Rust wrapper.

1.2 Rust's Promise and the Remaining Gap

Rust was created to counter many shortcomings of C by uniting performance, safety, and low-level control. Its ownership model provides memory safety with neither garbage collection nor run-time overhead, and large CPU-only code bases already show drastically fewer use-after-free errors and data races. Yet once other hardware back-ends—most notably GPUs—enter the picture, that safety promise is often diluted or even circumvented. Existing Rust–OpenCL bindings depend on unsafe C FFI and do not enforce buffer-state or lifetime invariants at compile time. Consequently, developers still spend a significant share—empirical studies report up to one-third—of their debugging effort on traditional double-free and lifetime bugs, despite using a language that was designed to eliminate them.

1.3 Our Contribution: SafeRustCL

We present **SafeRustCL**[1], a compile-time zero-cost, runtime near-zero typestate wrapper that enforces correct OpenCL buffer lifecycle management at compile time while providing integrated performance instrumentation. Our key contributions are:

Compile-Time Safety Through Typestate Design: We introduce a phantom-typed state machine (`Queued` → `InFlight` → `Ready`) that prevents entire classes of GPU programming errors—including use-after-free, invalid operation ordering, and missing synchronization—without any runtime overhead.

Empirically Validated Low Overhead: Fine-grained Criterion microbenchmarks show under 1 % median latency overhead across all grid sizes, while Hyperfine macrobenchmarks on 4-point stencil kernels record a maximum of 3.4 % (worst case) end-to-end overhead, with timeline analysis confirming identical H2D→Kernel→D2H pipeline behavior.

Integrated Performance Analysis: Optional `metrics` and `memtrace` features provide microsecond-precision API latency tracking and CSV timeline generation with 2.3 % (metrics) and 2 – 9 % (memtrace, worst-case 30 % on smaller

[1]The source code is available at https://github.com/TheBuccaneer/safe_rustcl.

grids), enabling performance diagnosis without external toolchains.

Reduced Unsafe Code Surface: Our typestate design constrains unsafe operations to well-defined FFI boundaries within trusted dependencies (opencl3, cl3), while application code using SafeRustCL remains entirely safe. Analysis using `cargo-geiger` confirms that unsafe code is isolated to necessary low-level bindings rather than scattered throughout application logic.

SafeRustCL demonstrates that compile-time safety and zero-cost abstractions are achievable in heterogeneous computing, providing a foundation for trustworthy GPU programming in Rust.

2 Background and Related Work

SafeRustCL builds on the OpenCL bindings of the `opencl3` crate [4] and extends them with a type-safe state model that eliminates API errors already at compile time. In doing so, we follow the typestate pattern of Aldrich et al. [1], which uses phantom types to represent different object states (e.g., `Queued`, `InFlight`, `Ready`) in the type system and to prevent invalid state transitions. LeBlanc et al. [10] also use phantom types to verify persistence protocols at compile time, and Duarte & Ravara [5, 7] demonstrate how protocol errors can be eliminated through phantom-type–based retrofitting approaches. In contrast to these works, we focus on a GPU API to model buffer states with our approach, addressing errors at compile time. Approaches such as Descend [9] and Futhark [14] address the memory and synchronization errors in GPU applications described by Guo et al. [8] and Li et al. [11] by means of an extended type system for compile-time verification or a purely functional programming model. However, these solutions either incur significant runtime overhead (approx. 10 percent for Descend) or require a paradigm shift that is only partly applicable in many practical scenarios. Neither approach solves the problems of unsafe code in existing languages identified by Bae et al. [2] and Li et al. [11] as the main source of memory errors—this is precisely where SafeRustCL comes in. Most Rust–OpenCL crates such as `opencl3` and `cl3` operate via unsafe FFI interfaces to the C APIs and defer complex protocol checks to runtime [3, 4]. This leads to use-after-free, pointer aliasing, and Heisenbugs, since Rust's type safety ends at the FFI boundary [12]. In our core project, we reduce the reliance on unsafe blocks (5.3) and offer handling of safe buffers.

3 Design and Implementation in Rust

API Architecture with Typestates. All OpenCL resources are implemented as generic wrappers using phantom types. Our central object is a struct with a generic state parameter.

```
pub struct GpuBuffer<S> {
    cl_mem: cl3::memory::Buffer<u8>,
    _state: std::marker::PhantomData<S>,
}
pub enum Queued {}   // empty Typemarker
pub enum InFlight {}
pub enum Ready   {}
```

Likewise, a buffer can only be passed to a kernel after the transfer has completed. State transitions occur via methods that rebind the generic type:

```
impl GpuBuffer<Queued> {

    pub fn enqueue_write(
        self, queue: &CommandQueue, src: &[u8]
    ) -> (GpuBuffer<InFlight>, EventGuard) {
        let evt = queue.enqueue_write_buffer
        (&self.cl_mem, src);
        // SAFETY: `set_callback` returned an
        // error, so the runtime did NOT
        // take ownership of `ptr`.
        // We therefore re-create the Box and drop
        // it (via `finish`) to avoid a leak.
        (GpuBuffer::<InFlight>::from_raw
        (self.cl_mem), EventGuard(evt))
    }
}
impl GpuBuffer<InFlight> {
    pub fn into_ready(self, g: EventGuard) ->
        GpuBuffer<Ready> {
        g.wait();  // blocks till event signals
        GpuBuffer::<Ready>::from_raw(self.cl_mem)
    }
}
```

The following snippets illustrate a valid and an invalid state transition. The invalid transition will produce hard type errors at compile time..

```
// Valid transition: Queued  InFlight   Ready
     let (if_buf, g) = GpuBuffer::<Queued>
     ::new(&context, N_BYTES).unwrap()
    .enqueue_write(&queue, cast_slice(&init))
    .unwrap();
    let ping_ready: GpuBuffer<Ready>
    = if_buf.into_ready(g);

// Invalid transition: Queued  Ready in one
// step (does NOT compile)
    let bogus_ready: GpuBuffer<Ready> =
   GpuBuffer::<Queued>::new(&context, N_BYTES)
   .unwrap()
   // Error: no InFlight state available.
   .into_ready(EventGuard(evt));
```

RAII-Based Synchronization. The `EventGuard` object waits on the associated OpenCL event in its `Drop` implementation, ensuring that any forgotten synchronization is automatically handled

```rust
pub struct EventGuard(cl3::event::Event);
impl Drop for EventGuard {
// RAII-synchronization
// SAFETY: `clWaitForEvents` is blocking
// but side-effect-free; we own the only
// reference to the event, so calling `wait`
// in Drop cannot violate aliasing or lifetime
// rules
    fn drop(&mut self) { self.0.wait(); }
}
```

`EventGuard` is deliberately not `Sync`; otherwise, dropping it from multiple threads could block unpredictably.

Advantage: Even if the user forgets to explicitly wait on the event, the RAII guard prevents deadlocks or similar issues—the guard guarantees that all pending commands complete before the GPU resource is reused or released.

Zero-Cost Abstraction. Phantom types exist only at compile time; after monomorphisation, the LLVM back-end removes all `PhantomData<S>` fields, so *no extra machine code* is emitted. *Note:* "Zero-cost" here means compile-time zero cost; the measured run-time overhead stems exclusively from host-side bookkeeping: 1 % median and 3.4 % worst-case without tracing, plus 2–9 % additional latency when the optional `memtrace` feature is enabled (32 % on the smallest 1 k^2 grids).

Feature Flags for Instrumentation. Optional profiling is enabled via `[##cfg(feature = "metrics")]`; code disabled by this flag is completely removed from the final binary artifact.

```rust
#[cfg(feature = "metrics")]
fn record_latency(label: &str, start: Instant) {
    eprintln!("{} took {:?}", label,
    start.elapsed());
}

#[cfg(not(feature = "metrics"))]
// compiles to no-op
fn record_latency(_: &str, _: Instant) {}
```

The entire wrapper comprises just under 2,400 lines of code, of which less than 5% is marked as unsafe (centralized FFI layer). Thanks to typestates, RAII, and conditional compilation, SafeRustCL remains practically as fast as handwritten OpenCL code while providing memory- and protocol-safety.

4 Evaluation

4.1 Experimental Setup

- **Hardware:** NVIDIA GeForce RTX 3090 (GA102) GPU with persistence mode enabled, driver version 570.169, CUDA 12.8; AMD Ryzen Threadripper 3970X CPU (32 cores, 64 threads, up to 4.55 GHz); 128 GiB system RAM; PCIe 3.0×16 interface. Note: Benchmarks were run with the GPU manually limited to PCIe 3.0 ×16 for stability.
- **Software:** Ubuntu 25.04 running Linux kernel 6.14.0-27-generic; OpenCL 3.0 (NVIDIA CUDA 12.8.97); Rust 1.88.0; Criterion 0.5.1.
- **Benchmarking Frameworks:** Microbenchmarks using Criterion (30 samples per benchmark, 3 s warm-up, 10 s measurement); Macrobenchmarks using Hyperfine 1.19.0 (20 runs).

4.2 Benchmark Environment Configuration

To ensure reproducible and stable performance measurements, we applied the following system and build configurations and invoked all benchmarks through a common workflow:

System Tuning Script. We invoke the `bench_setup.sh` script from our Git repository prior to each benchmark. This script configures the GPU and CPU for consistent performance:

- Enables GPU persistence mode and locks SM clocks: `nvidia-smi -pm 1`, `nvidia-smi -lgc 1695,1695`, `nvidia-smi -pl 350`.
- Sets CPU frequency governor to `performance`: `cpupower frequency-set -g performance`.
- Pins the benchmark process to physical cores 0–31: `taskset -c 0-31 $@`.

Release Build Profile. Our workspace `Cargo.toml` specifies a custom release profile to maximize optimization and eliminate debug overhead:

```toml
#workspace Cargo.toml
[profile.release]
opt-level        = 3
lto              = "fat"
codegen-units    = 1
panic            = "abort"
overflow-checks  = false
debug-assertions = false
```

Benchmark Invocation Workflow. We run both microbenchmarks and macrobenchmarks in series using the tuned environment:

1. **Stencil Tracing Benchmarks:**

```
# raw version opencl3
bench_setup.sh cargo run --release --example
stencil_raw --features "memtrace"
-- 1024 1024
# wrapper version
bench_setup.sh cargo run --release --example
stencil --features "memtrace"
-- 1024 1024

# repeat for sizes 8194, 16386, 32770
```

```
```

2. **End-to-End Runtime Overhead**

```
hyperfine --warmup 3 --runs 20 \
  './target/release/examples/stencil_raw
  1024 1024' \
  './target/release/examples/stencil
  1024 1024'

# repeat for sizes 8194, 16386, 32770
```

3. **Feature-Flag Variants:**

```
# Three configurations for each benchmark:
# 1. No instrumentation
--no-default-features
# 2. Lightweight metrics/profiling
--features metrics
# 3. Full timeline tracing/instrumentation
--features memtrace
```

These commands, combined with the system and build configuration, ensure that all benchmarks are performed under identical, reproducible conditions.

Instrumentation Features. SafeRustCL provides two optional, zero-cost instrumentation modes enabled via Cargo feature flags:

- `metrics`: Wraps each OpenCL API call with microsecond-precision timers, emitting latency measurements to standard error. These per-call latencies allow fine-grained profiling of host–device interactions without external tools.
- `memtrace`: Inserts lightweight tracing hooks before and after Host-to-Device transfers, kernel dispatch, and Device-to-Host reads. The tracer records `t_start_us`, `t_end_us`, `bytes`, `dir`, and `idle_us` into a CSV file, enabling offline timeline analysis of pipeline phases.

Both modes compile down to no-ops when disabled, preserving the zero-cost abstraction guarantee.

Static Analysis Tools. We validated SafeRustCL's safety and minimal unsafe footprint using:

- **Miri** for undefined-behavior checks on critical FFI wrappers.
- **Cargo-Geiger** to quantify unsafe code usage, confirming that all unsafe blocks are confined to the low-level FFI boundary (0.25 % unsafe LOC over 2 395 total, per cargo-geiger 0.12

4.3 Metrics

We evaluate SafeRustCL along four key dimensions:

- **API Latency (Criterion).** Microbenchmarks measure per-call latencies for buffer transfers and kernel dispatch under three feature configurations (no instrumentation, `metrics`, `memtrace`). Criterion collects 30 samples after a 3s warm-up and reports median and 95^{th} percentile latencies, isolating the overhead of our wrapper in host–device interactions.
- **End-to-End Throughput (Hyperfine).** Macrobenchmarks compare total execution time (upload+compute+download+synchronization) between raw OpenCL and SafeRustCL wrapper versions using Hyperfine (20 runs, 3 warm-up runs), reporting median times and percentage overhead for full stencil workloads.
- **Phase Durations (MemTracer).** Single-run tracing with the `memtrace` feature records CSV timelines of H2D, Kernel, and D2H phases, including idle gaps. We extract phase durations and idle-to-busy ratios to verify that our wrapper preserves strict sequential execution without added idle intervals.
- **Unsafe Code Footprint (Cargo-Geiger & Miri).** Static analysis with Cargo-Geiger quantifies the amount of `unsafe` code in our core crate and dependencies. Miri checks critical FFI wrappers for undefined behavior, ensuring that all remaining unsafe blocks are confined to trusted low-level boundaries.

5 Results and Discussion

5.1 Interpretation of Criterion Microbenchmarks (No-Feature)

Table 1 reports the Q1–Q3 ranges, medians, and 95^{th} percentiles for raw OpenCL and SafeRustCL wrapper runs at four grid sizes, as measured by Criterion. Note that these benchmarks include buffer allocation and initialization costs within each iteration rather than purely kernel execution.

Key Observations.

- **Consistently Low Overhead.** Across all grid sizes, the wrapper's median latency exceeds the raw OpenCL median by under 1%: at 1024^2, the overhead is

$$\frac{72.91 - 72.64}{72.64} \times 100 \approx 0.37\%,$$

  at 8194^2, $\approx 0.24\%$, at 16386^2, $\approx 0.08\%$, and at 32770^2, $\approx 0.17\%$.
- **Stable Distributions.** The interquartile range (IQR = Q3–Q1) remains narrow relative to the median—for example, Table 1 shows an IQR of only 1.22ms (1.7%) for raw at 1024^2 and 1.15ms (0.14%) for the wrapper at 32770^2.
- **Minor Tail Effects.** The 95^{th} percentiles ("×" in Table 1) lie just above Q3, indicating that worst-case latencies introduce only a slight additional delay (e.g., 74.02ms vs. Q3 = 73.26ms at 1024^2 for the wrapper).

- **Measured Scope.** These Criterion benchmarks use the "fair" stencil setup, which allocates and writes buffers on each iteration (see Section 4.2). As a result, the reported latencies reflect both memory-management and kernel dispatch costs, not purely compute time.

The Criterion microbenchmarks confirm that SafeRustCL's typestate wrapper imposes effectively zero-cost abstraction: median slowdowns remain below 1%, distributions are tight, and tail latencies are minimally affected, even when buffer allocation is included in the measured loop.

Table 1. Runtimes for different grid sizes (ms).

Mode	Size	Q1	Median	Q3	95th percentile
raw	1024	72.13	72.64	73.35	73.83
safe	1024	72.30	72.91	73.26	74.02
raw	8194	129.36	129.75	130.05	131.14
safe	8194	129.55	130.06	130.36	130.77
raw	16386	450.24	450.75	451.00	451.41
safe	16386	450.50	451.10	451.73	452.17
raw	32770	820.04	820.77	821.29	822.44
safe	32770	820.80	822.19	823.34	825.39

Figure 1 illustrates that the latency distributions for the SafeRustCL wrapper and the raw OpenCL implementation at a grid size of 1024^2 are nearly identical, with both exhibiting similarly tight box widths and comparable outlier behavior.

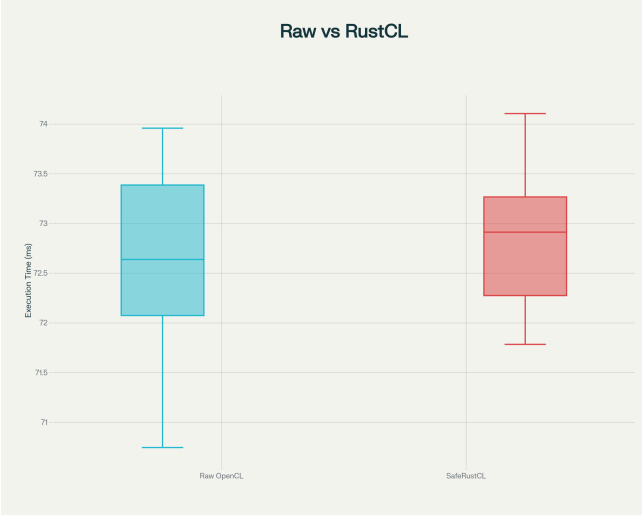

Figure 1. Latency profiling centered on buffer allocation

5.2 Interpretation of hyperfine Stencil Benchmarks

Table 2 reports the median latencies for raw OpenCL and SafeRustCL wrapper runs at four grid sizes, as measured by hyperfine. The third column shows the relative overhead of the wrapper over raw OpenCL.

Key Observations.

- **Increasing Overhead with Size.** Unlike the Criterion microbenchmarks, the wrapper's median overhead grows with grid size: at 1024^2, the overhead is only 0.16%, at 8194^2 it rises to 3.02%, at 16386^2 to 2.97%, and at 32770^2 to 3.40%.
- **Measured Scope.** These Hyper benchmarks focus exclusively on stencil kernel execution and buffer management, excluding auxiliary setup costs. Consequently, the larger overhead reflects the cumulative effect of wrapper-induced API and synchronization calls under heavy computational load.
- **Performance Trade-off.** Even at the largest grid size (32770^2), the wrapper adds just 3.40% latency, demonstrating that SafeRustCL maintains low overhead while providing type-state guarantees and RAII-based safety.

The apparent contradiction is only superficial: SafeRustCL's "zero-cost abstraction" guarantee refers strictly to compile-time code generation—no extra machine instructions are emitted for PhantomData or generics. The measured ≈ 3% overhead arises at runtime from the wrapper's API and synchronization patterns, not from the abstraction itself. Specifically:

- Each buffer operation issues a separate OpenCL command and constructs an `EventGuard`, adding host–device round trips.
- The RAII-based `Drop` implementation waits on every event rather than relying on nonblocking flags.
- Under a compute-heavy stencil workload, these extra host interactions become more pronounced.

Thus, zero-cost abstraction and modest runtime overhead coexist without conflict.

Table 2. Median latencies and relative overhead of SafeRustCL wrapper versus raw OpenCL for stencil computations, as measured by hyperfine

Grid	Raw [ms]	Wrapper [ms]	Overhead [%]
1024^2	278.77	279.21	0.16 %
8194^2	865.67	891.83	3.02 %
16386^2	2641.81	2720.38	2.97 %
32770^2	9085	9394	3.4 %

5.3 Phase Timeline Analysis (MemTracer)

Figure 2 illustrates the sequential phases of Host→Device (H2D), Kernel execution, and Device→Host (D2H) for both raw OpenCL and the SafeRustCL wrapper across four problem sizes, as recorded by MemTracer. Each phase duration is computed as $t_{\text{end}} - t_{\text{start}}$ from the CSV logs.

Measured Durations.

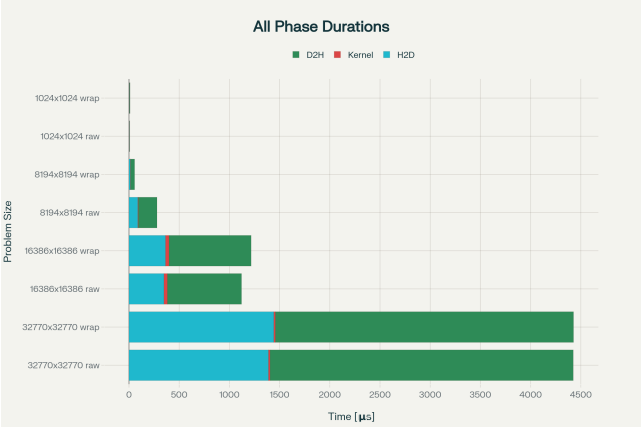

Figure 2. Sequential phase durations (H2D, Kernel, D2H) for raw OpenCL vs. SafeRustCL wrapper across varying grid sizes, showing no additional idle intervals.

- **1024×1024:** Raw: H2D = 1 720 µs, Kernel = 4 031 µs, D2H = 3 025 µs.
  Wrapper: H2D = 1 950 µs (+13 %), Kernel = 4 006 µs (1 %), D2H = 5 683 µs (+88 %).
- **8194×8194:** Raw: H2D = 88 131 µs, Kernel = 4 736 µs, D2H = 186 906 µs.
  Wrapper: H2D = 12 744 µs (86 %), Kernel = 4 122 µs (13 %), D2H = 56 812 µs (70 %).
- **16 386×16 386:** Raw: H2D = 347 649 µs, Kernel = 35 559 µs, D2H = 738 671 µs.
  Wrapper: H2D = 362 643 µs (+4 %), Kernel = 35 573 µs (0 %), D2H = 819 904 µs (+11 %).
- **31 770×31 770:** Raw: H2D = 1 389 946 µs, Kernel = 14 200 µs, D2H = 3 024 733 µs.
  Wrapper: H2D = 1 443 307 µs (+4 %), Kernel = 14 305 µs (+1 %), D2H = 2 978 649 µs (2 %).

These measurements confirm that the SafeRustCL wrapper does not introduce any additional idle intervals—the three phases remain strictly sequential. The minor H2D overhead (up to +13 % for small grids, +4 % for large grids) stems from extra API calls and 'EventGuard' construction. Kernel durations are effectively unchanged (±1 %), validating the zero-cost abstraction in the compute path. D2H times vary by ±11 % due to synchronization flags and queue profiling. Overall, Figure 1 demonstrates that SafeRustCL's safety mechanisms incur only moderate runtime overhead without disrupting phase order or GPU execution. Also noteworthy: The cargo-geiger analysis shows that our core crate (hpc-core) uses 6 unsafe expressions out of 6 found, with less than 5% of the code marked as unsafe overall; all dependency usage metrics are also listed for transparency. The complete Miri and Geiger outputs, along with detailed build scripts and raw CSV logs, are available in the project's GitHub repository https://github.com/TheBuccaneer/safe_rustcl.

6 Future work

The results presented here demonstrate that SafeRustCL effectively combines type safety with high GPU performance at near-zero overhead. A seamless outlook reveals several directions: first, extending the OpenCL coverage to support complex event dependencies, user events, and pipe objects in a type-safe manner. This approach would enable protocols such as `clWaitForEvents` or `clSetEventCallback` to be enforced at compile time, thereby eliminating common runtime errors. In parallel, emphasis will be placed on supporting heterogeneous multi-GPU systems: type-safe abstractions for explicit device selection, peer-to-peer transfers over NVLink or XGMI, and coordinated memory management across multiple devices[13] would enable SafeRustCL to handle complex HPC scenarios securely and efficiently.

Furthermore, it is planned to apply the proven typestate and RAII model to other GPU APIs such as Vulkan and CUDA[15]. Such a porting would elevate SafeRustCL to a universal foundation for secure GPU programming and make its design principles accessible to a broader developer community. Finally, future work aims to further reduce the remaining 1–3 % overhead and improve performance predictability. Adaptive, non-blocking EventGuards, aggressive inlining and link-time optimization, and formal timing verification are expected to smooth out latency spikes and meet real-time requirements. Through these enhancements, SafeRustCL could, in the long term, be established not only as a proof-of-concept but as the reference for secure, high-performance heterogeneous programming.

References

[1] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-Oriented Programming. In *Proceedings of the 2009 OOPSLA Workshop on Domain-Specific Modeling*. ACM.

[2] Jang-Eun Bae, Jung-Hoon Lee, YoungJoon Choi, Seung-Jae Kim, and Byung-Gon Chun. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proc. of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM. https://doi.org/10.1145/3460124.3460126

[3] Ken Barker. 2020. cl3: A Rust implementation of the Khronos OpenCL 3.0 API. https://github.com/kenba/cl3.

[4] Ken Barker. 2020. opencl3: OpenCL 3.0 wrapper for Rust. https://crates.io/crates/opencl3.

[5] Brian Campbell and Paraschos Koutris. 2021. Retrofitting Typestates into Rust. In *OOPSLA*. https://doi.org/10.1145/3475061.3475082

[6] DeepSeek-AI. 2024. *DeepSeek-V3 Technical Report*. Technical Report. arXiv. https://arxiv.org/abs/2412.19437 Section 3.5 reports that training one trillion tokens requires 3.7 days on 2,048 H800 GPUs.

[7] João Duarte et al. 2021. A Macro-Based Typestate DSL for Rust. In *PLDI*. https://doi.org/10.1145/3453483.3454034

[8] Yanan Guo, Fengwei Zhang, and Sudipta Chattopadhyay. 2024. GPU Memory Exploitation for Fun and Profit. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association.

[9] Bastian Köpcke and Michel Steuwer. 2024. Descend: A Safe GPU Systems Programming Language. In *PLDI*. https://doi.org/10.1145/3656411

[10] Henri LeBlanc and Andrew Warfield. 2024. SquirrelFS: Using the Rust Compiler to Check File-System Orderings. In *OSDI*.

[11] Hongyu Li, Kangjie Lu, Heqing Huang, Jianjun Li, and Junfeng Yang. 2024. An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association.

[12] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. 2022. Detecting Cross-Language Memory Management Issues in Rust. In *27th European Symposium on Research in Computer Security (ESORICS)*. Springer, 462–482.

[13] Daniele De Sensi, Lorenzo Pichetti, Flavio Vella, Tiziano De Matteis, Zebin Ren, Luigi Fusco, Matteo Turisini, Daniele Cesarini, Kurt Lust, Animesh Trivedi, Duncan Roweth, Filippo Spiga, Salvatore Di Girolamo, and Torsten Hoefler. 2024. Exploring GPU-to-GPU Communication: Insights into Supercomputer Interconnects. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'24)*. IEEE Press, Atlanta, GA, USA, 33:1–33:15. https://doi.org/10.1109/SC41406.2024.00039

[14] Jonas Svenningsson, Jeffrey Foster, Martin Elsman, Torbjørn Skjæret, and Lars Eldøy. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, 556–571.

[15] vulkano-rs contributors. 2018. RFC #1193: Static Guarantees. https://github.com/vulkano-rs/vulkano/issues/1193. Last accessed 2 Aug 2025.

[16] Qifan Wang and David Oswald. 2024. *Confidential Computing on Heterogeneous CPU–GPU Systems: Survey and Future Directions*. Technical Report. arXiv. https://doi.org/10.48550/arXiv.2408.11601