Exploring GPU Acceleration for Primitive Operations in Substrate-based Blockchain Frameworks

QF Foundation Vas Soshnikov* Max Pestov†

Abstract

This paper presents an exploratory study on the potential of GPU acceleration for executing primitive mathematical operations within the Substrate blockchain framework. As part of QuantumFusion's broader mission to revolutionize blockchain technology, we investigate the performance gains of offloading basic arithmetic operations to GPUs compared to traditional CPU execution. Our research, conducted by the QF Foundation Team, aims to contribute to the ongoing efforts to enhance blockchain performance and scalability in the near term, while complementing efforts to integrate more advanced parallel processing techniques. While our initial scope focused on simple operations, this work lays the groundwork for future investigations into more complex smart contract executions on GPUs within blockchain environments.

1 Introduction

Blockchain technology continues to evolve, with performance and scalability remaining key areas for improvement. QuantumFusion emerged from a vision to create a new paradigm in blockchain technology, inspired by innovative approaches to parallel computing. This study, conducted by the QF Foundation Team, explores the immediate potential of integrating GPU processing capabilities within the Substrate blockchain framework [4], specifically for executing primitive mathematical operations.

In the broader context of QuantumFusion's research initiatives, it's important to note the ongoing development of advanced parallel processing techniques, including the exploration of Higher Order Virtual Machine 2 (HVM2) [9]. HVM2, a massively parallel Interaction Combinator evaluator, shows great promise for future blockchain optimizations. While a dedicated team within QuantumFusion is actively developing solutions using HVM2 technology, our current study focuses on more immediately implementable GPU acceleration techniques. This dual approach allows us to pursue both short-term performance enhancements and long-term revolutionary changes in blockchain architecture.

As part of QuantumFusion's commitment to pushing the boundaries of blockchain technology, our team embarked on this research to explore immediate performance enhancements. While the full realization of QuantumFusion's vision involves various advanced techniques currently under development, our study on GPU acceleration aims to provide tangible improvements that can be implemented in the shorter term, contributing to the overall goal of creating a high-performance blockchain ecosystem.

We explored various approaches to achieve our performance goals, ultimately focusing on a WebAssembly (Wasm) based approach using the arrayfire-rust module [1] for GPU computations. This aligns with our goal of delivering exceptional performance within our project timeline.

This paper outlines our methodology, presents initial findings, and discusses the implications for future research in this domain. We aim to provide a transparent account of our exploration, including both successes and challenges encountered, to contribute valuable insights to the blockchain development community and set the stage for future advancements in parallel processing within blockchain environments.

2 Objectives

Building on QuantumFusion's multifaceted approach to blockchain innovation, this study focuses on the potential of GPU acceleration within the Substrate framework. Our research objectives are designed to explore immediate performance enhancements while complementing the organization's long-term goals. We established the following objectives:

1. Implement methodologies for executing primitive operations on the GPU for Substrate with WebAssembly.
2. Conduct a comparative analysis of performance metrics between CPU and GPU execution of primitives according to the methodology outlined in Section III.
3. Lay the groundwork for future, more comprehensive research.

3 Methodology

The methodology for testing and comparative analysis consisted of the following steps:

1. For collecting information about code execution, we used RUST's criterion package [7] (version installed with RUST). The hardware used was: M1 Pro, SSD, 16GB RAM, MacOS Sonoma 14.5. Software versions were: ArrayFire 3.8.0 [1], RUST 1.80.0 [6], and Substrate [4].
2. We introduced a function performing addition and division of 1,000,000 random float numbers.
3. The same function was used for both CPU WebAssembly Virtual Machine and GPU WebAssembly Virtual Machine (the first and second testing phases, respectively).

3.1 Implementation Details

The implementation of our GPU acceleration approach is available on GitHub. The main branch of our work can be found at:

`https://github.com/QuantumFusion-Foundation/polkadot-sdk/tree/develop`

For those interested in reproducing our results or building upon our work, we have provided a starting point with instructions:

`https://github.com/QuantumFusion-Foundation/polkadot-sdk/blob/develop/substrate/BENCH_GPU.md`

The specific changes made for this research can be viewed in the repository history:

`https://github.com/paritytech/polkadot-sdk/compare/master...QuantumFusion-Foundation:polkadot-sdk:develop`

We encourage the community to explore these resources and build upon our findings.

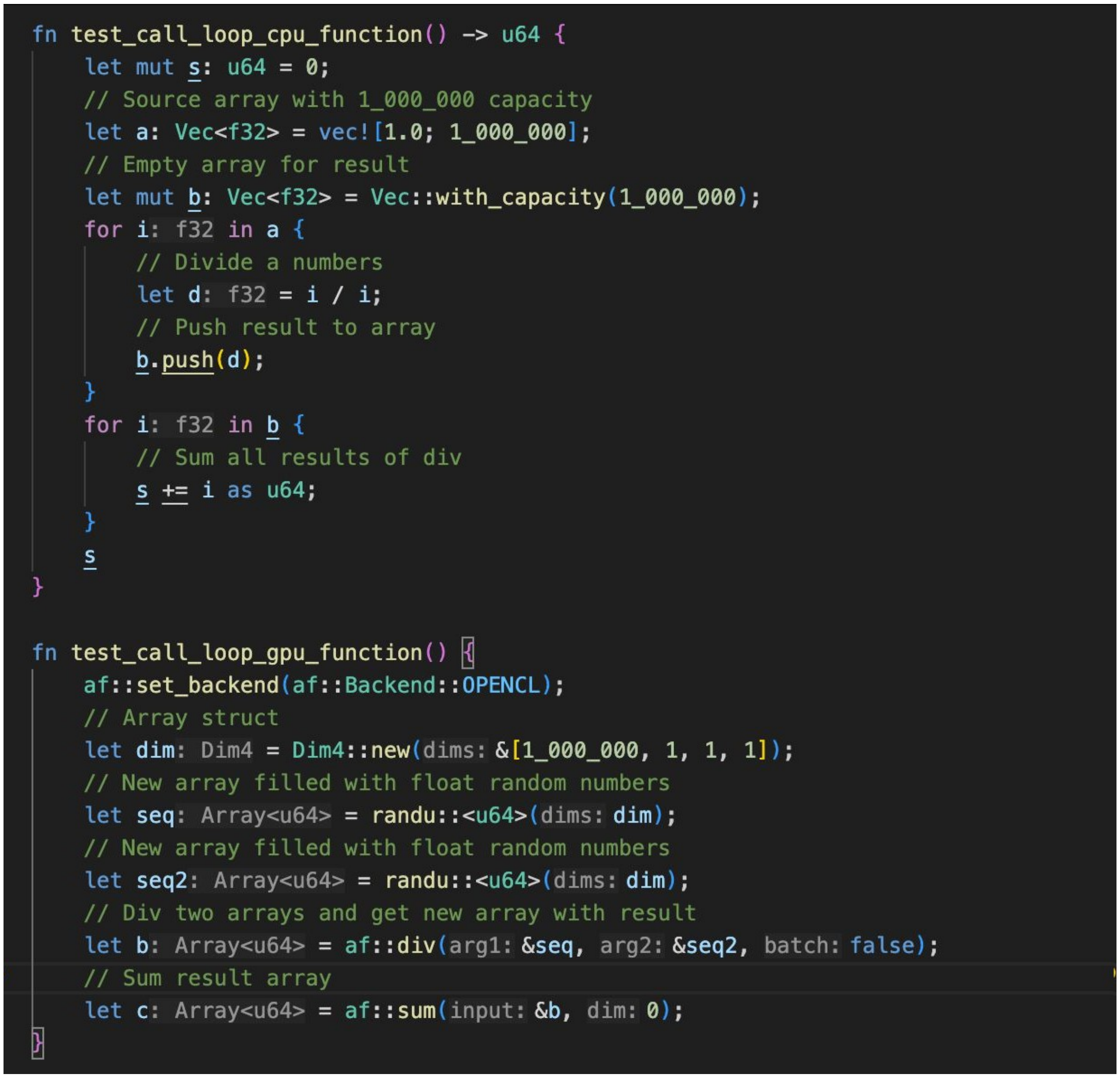
3.2 First Testing Phase

The initial phase of testing involved the standard execution of primitive operations within the Substrate framework using the WebAssembly Virtual Machine (VM) on the CPU. During this phase, execution times were recorded to establish a baseline reference for subsequent performance comparisons. This provided a foundational dataset against which the impact of GPU acceleration could be measured.

3.3 Second Testing Phase

In the second phase, we extended the existing Substrate framework without modifying the core codebase. By leveraging Substrate externalities [2], we integrated the execution of primitive operations at the entry point using the arrayfire-rust module [1]. This enabled offloading computations from the CPU to the GPU, facilitating the parallel execution of operations.

The implementation of the described phases involved the creation of two distinct functions for handling division and addition operations. For CPU execution, memory was allocated for an array of values, and the corresponding operations were performed sequentially on this array. In contrast, for GPU execution, a similar memory allocation process was employed; however, the calculations were distributed across the available GPU cores to leverage parallel processing. This approach aimed to exploit the inherent parallelism of the GPU architecture, potentially reducing execution time for large datasets.



```rust
fn test_call_loop_cpu_function() -> u64 {
    let mut s: u64 = 0;
    // Source array with 1_000_000 capacity
    let a: Vec<f32> = vec![1.0; 1_000_000];
    // Empty array for result
    let mut b: Vec<f32> = Vec::with_capacity(1_000_000);
    for i: f32 in a {
        // Divide a numbers
        let d: f32 = i / i;
        // Push result to array
        b.push(d);
    }
    for i: f32 in b {
        // Sum all results of div
        s += i as u64;
    }
    s
}

fn test_call_loop_gpu_function() {
    af::set_backend(af::Backend::OPENCL);
    // Array struct
    let dim: Dim4 = Dim4::new(dims: &[1_000_000, 1, 1, 1]);
    // New array filled with float random numbers
    let seq: Array<u64> = randu::<u64>(dims: dim);
    // New array filled with float random numbers
    let seq2: Array<u64> = randu::<u64>(dims: dim);
    // Div two arrays and get new array with result
    let b: Array<u64> = af::div(arg1: &seq, arg2: &seq2, batch: false);
    // Sum result array
    let c: Array<u64> = af::sum(input: &b, dim: 0);
}
```

Figure 1: An example of two functions that use GPU.

3.4 Metrics to Evaluate Benchmark Results

- Average time to complete an operation: Measures the mean duration required to perform a single instance of the arithmetic operations under evaluation. It provides a quantifiable measure of the efficiency and responsiveness of the computational execution on both CPU and GPU. This metric assesses the performance impact of GPU acceleration on individual operations, offering insight into how quickly each operation can be processed.

- Mathematical mean

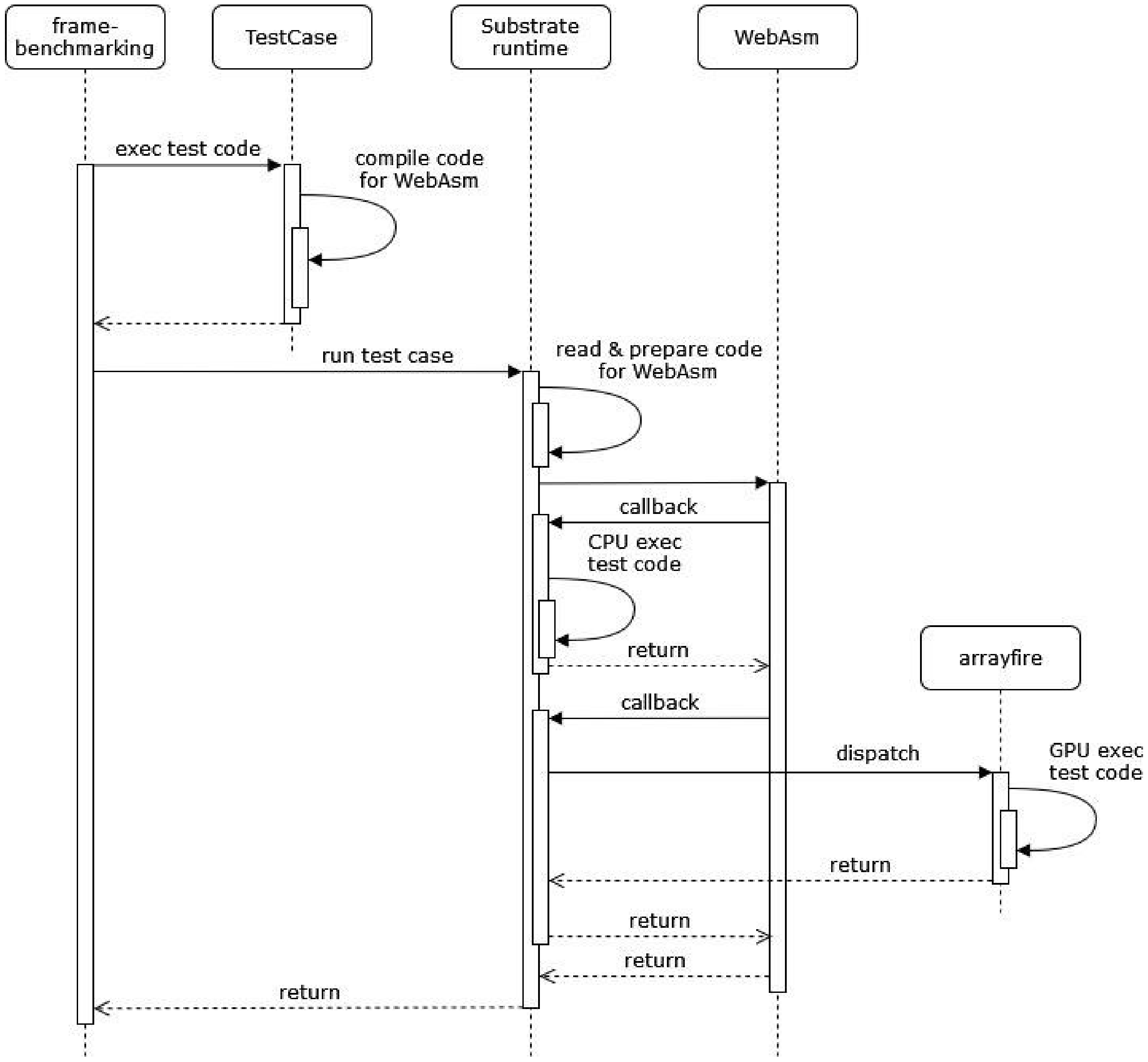


Figure 2: Execution flow diagram illustrating the process of GPU acceleration within the Substrate framework.

It's important to note that we did not introduce any operation balancer that performs batching of operations for optimal GPU execution. This is a limitation of our current research that other teams might explore in the future.

4 Results

Our benchmarking results demonstrate a significant performance improvement when utilizing GPU acceleration for primitive operations.

Metric	CPU	GPU
Average time	1.15 - 1.85 ms	0.38 - 0.50 ms
Mean time	1.3167 ms	0.4139 ms

Table 1: Execution time for primitive operations on 1,000,000 random float numbers

This chart shows the relationship between function and iteration time. The thickness of the shaded region indicates the probability that a measurement of the given function would take a particular length of time.

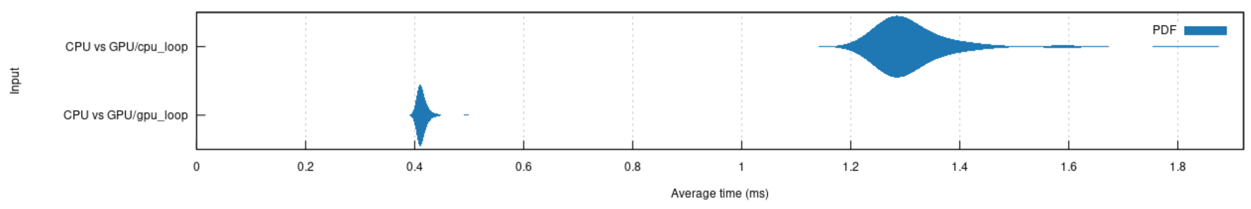


Figure 3: Relationship between function and iteration time.

4.1 CPU execution time for the primitive operations of 1,000,000 random float numbers

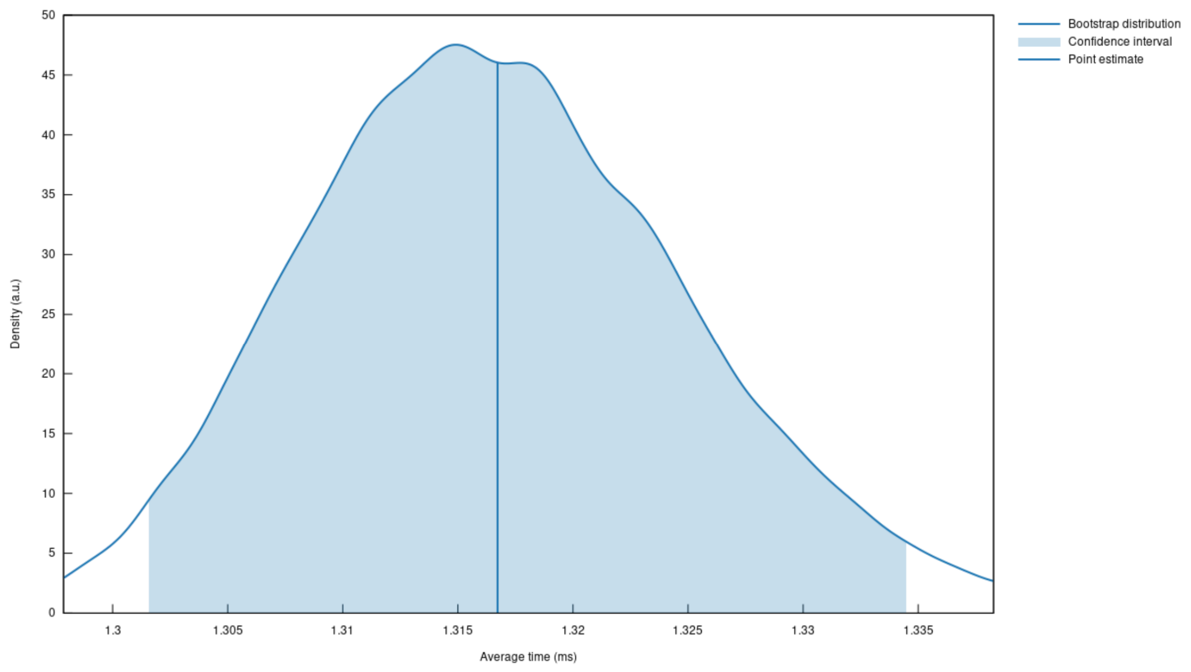


Figure 4: CPU execution time distribution

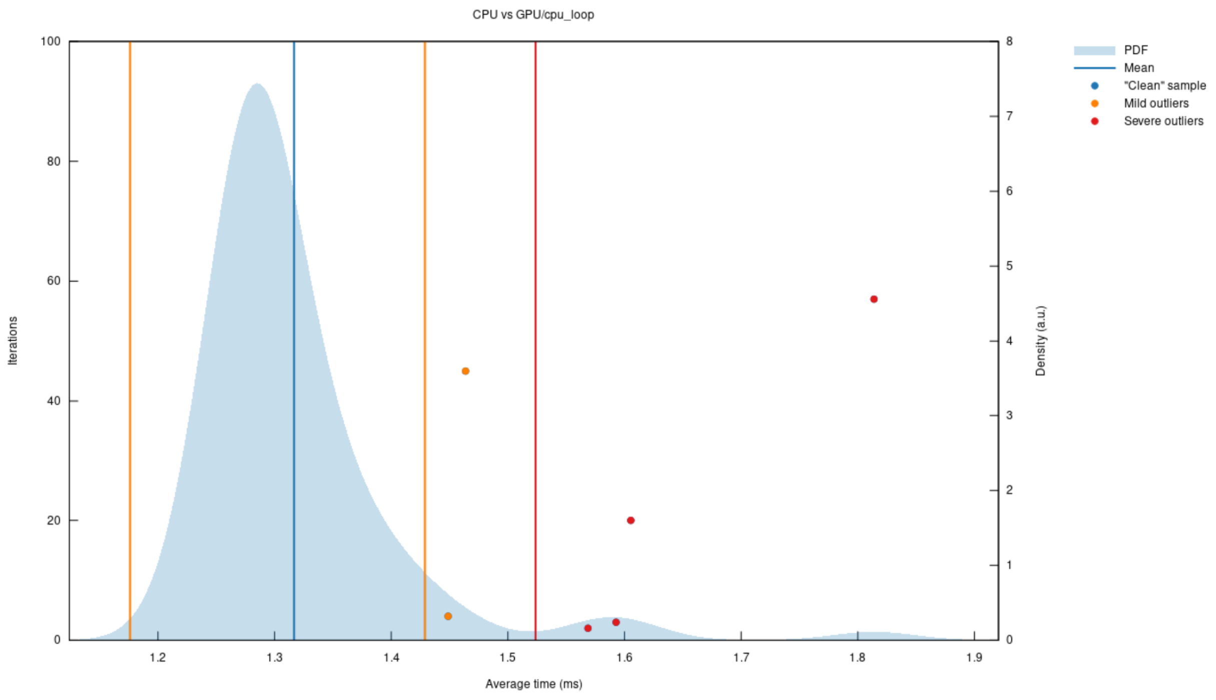


Figure 5: CPU mean time chart

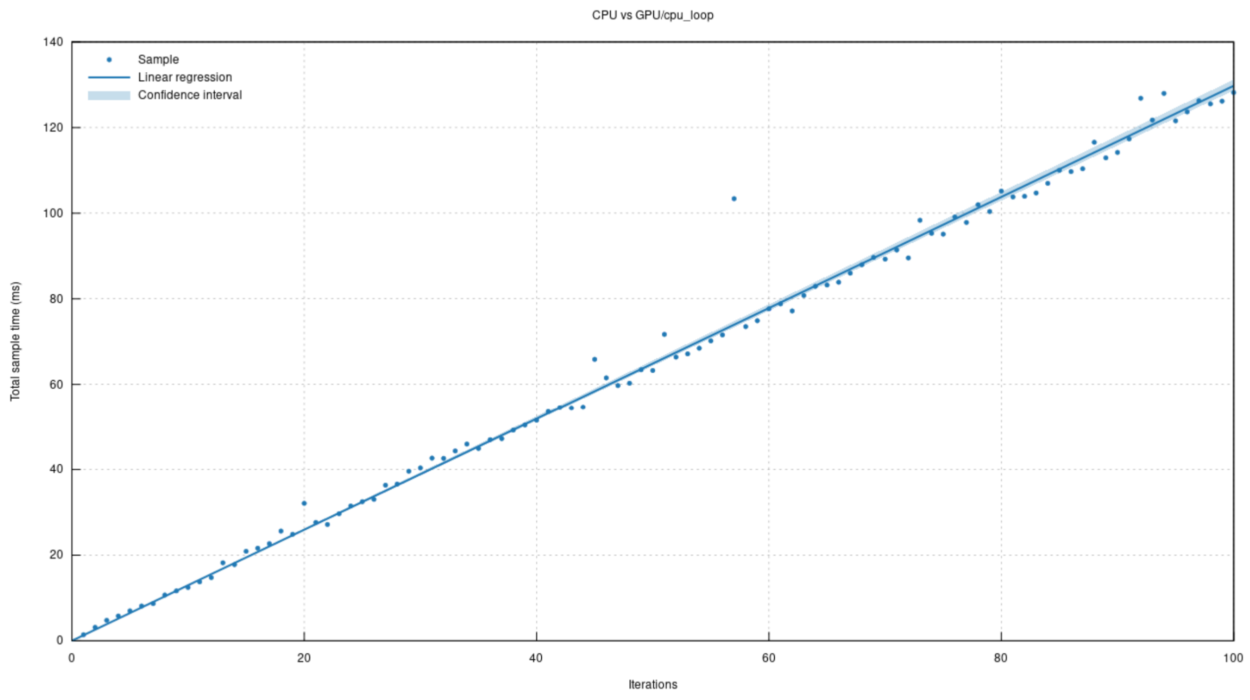


Figure 6: CPU linear regression

4.2 GPU execution time for the primitive operations of 1,000,000 random float numbers

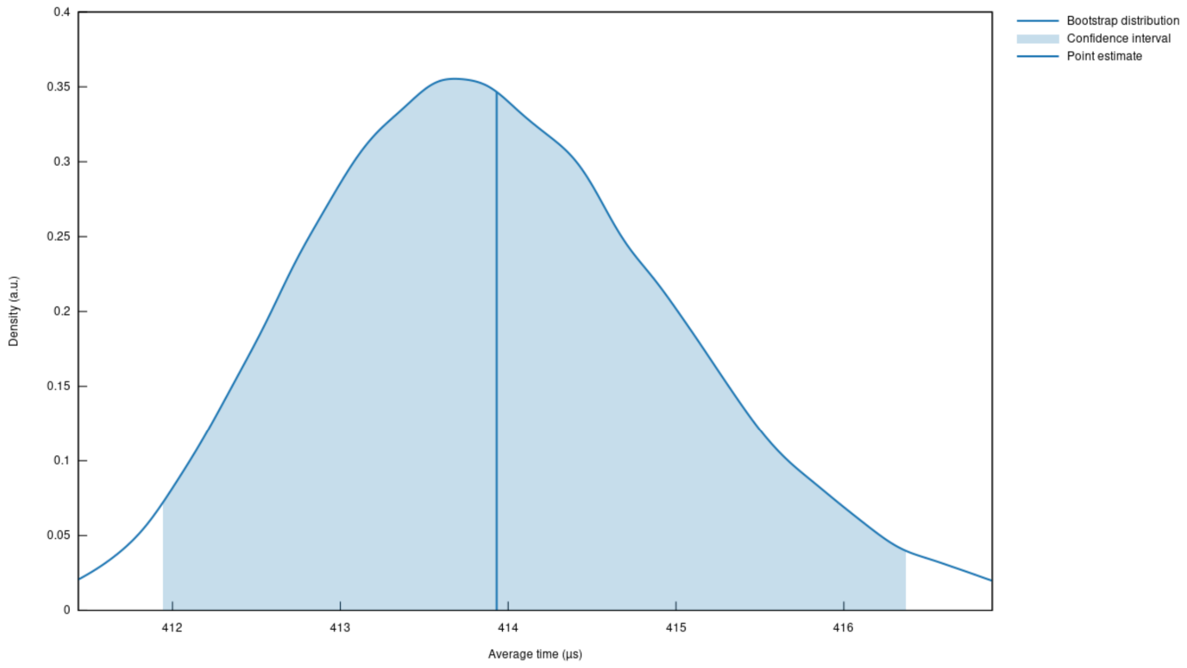


Figure 7: GPU execution time distribution

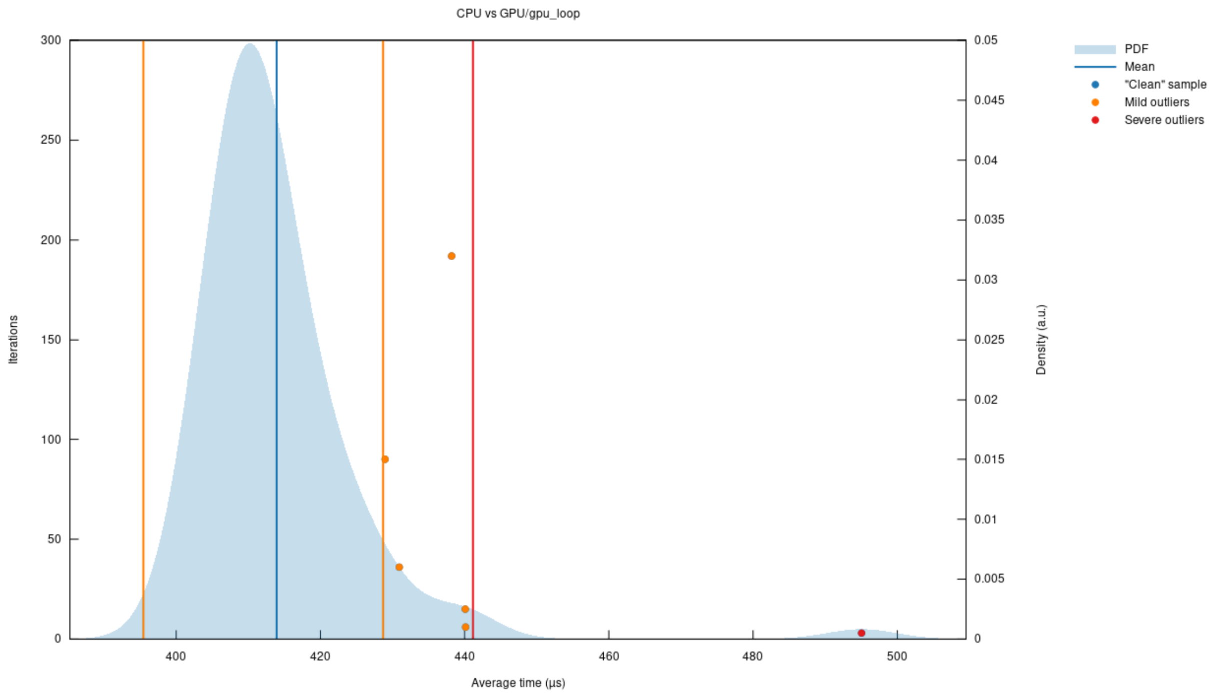


Figure 8: GPU mean time chart

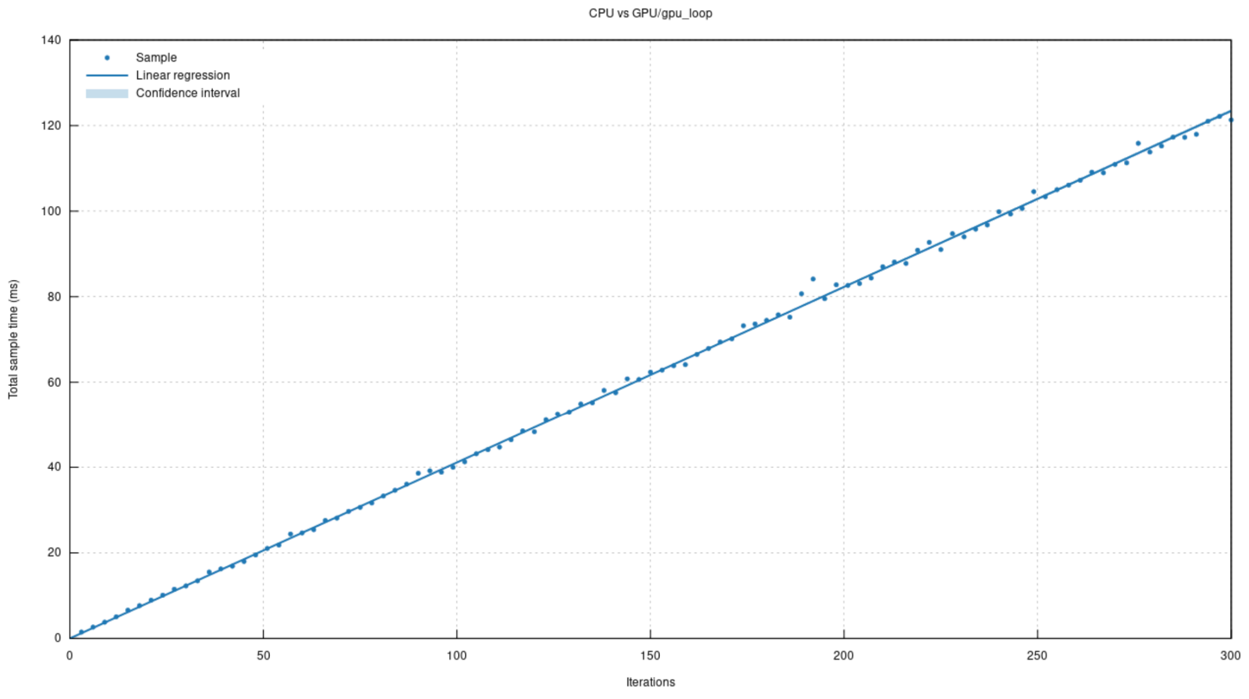


Figure 9: GPU linear regression

5 Conclusion

As we have demonstrated, there is potential for using GPU acceleration in blockchain's VM, at least for the case we studied. Would it be beneficial for other cases? We can't say for certain, but we believe that other teams could continue working in this direction based on our results and share their findings with the community.

Potential architecture might look like this:

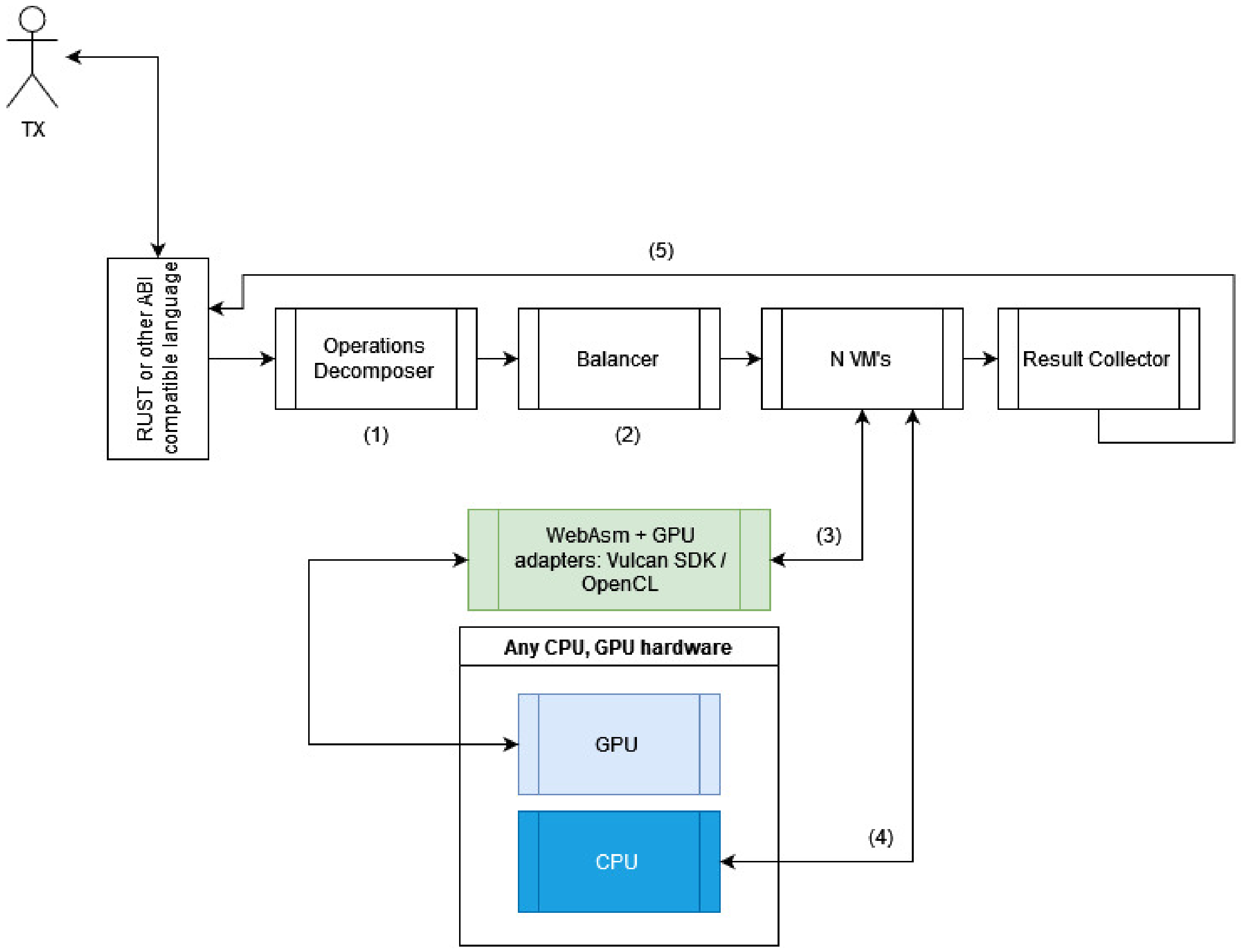


Figure 10: Conceptual architecture for a GPU-accelerated Substrate implementation with operation balancing.

The simple explanation of the process illustrated in the diagram is as follows: based on incoming required operations and the possibility of batching, the balancer uses either GPU or CPU (or both) to obtain the result.

For further exploration of this topic, we can point to some great references that might provide food for thought:

1. The logic of Linux kernel scheduling: `https://docs.kernel.org/scheduler/index.html`
2. Research that might improve balancing/scheduling operations for GPU, for instance: `https://arxiv.org/abs/2212.08964`
3. Range Schedulers (sharding) implementations, Cost Schedulers implementations, etc.,

from databases and distributed systems areas.

References

[1] Arrayfire-rust. Module which allows for GPU computations on the host at runtime with WebAssembly. `https://github.com/arrayfire/arrayfire-rust`

[2] Externalities. Substrate interface between the runtime logic (blockchain's core logic) and the outside world, including the operating system and other runtimes.

[3] Primitives. Fundamental computation operations: addition and division.

[4] Substrate. Modular framework for building customized blockchains. `https://substrate.io/`, `https://github.com/paritytech/polkadot-sdk/tree/master/substrate`

[5] WebAssembly (aka Wasm, WebAsm). Portable, binary instruction format designed for high-performance execution and cross-platform compatibility.

[6] RUST. A language for development. `https://www.rust-lang.org/tools/install`

[7] Criterion. Rust library on crates.io that provides statistical benchmarking tools for precise measurement and analysis of Rust code performance. `https://crates.io/crates/criterion`

[8] WebAssembly System Interface (WASI). WASI for the WebGPU API, enabling high-performance graphics and computation capabilities in WebAssembly applications. `https://github.com/WebAssembly/wasi-webgpu`

[9] HVM 2.0. Higher Order Virtual Machine 2 is a massively parallel Interaction Combinator evaluator. `https://github.com/HigherOrderCO/HVM`

*GitHub: @dedok, Email: vasiliy.soshnikov@gmail.com

†GitHub: @ealataur, Email: ealataur@gmail.com