GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Introduction to Performance Engineering in Rust

Lars Quentin

MatrNr: 21774184

Supervisor: Dr. Artur Wachtel

Georg-August-Universität Göttingen
Campus-Institut Data Science / GWDG

August 29, 2023

Abstract

[Put your abstract here.] Note that we typically publish your report as PDF on our webpage. Let us know if you disagree.]

General structure of an abstract, write 1 to 2 sentences per section

1. A general statement introducing the broad research area of the particular topic being investigated.
2. An explanation of the specific problem (difficulty, obstacle, challenge) to be solved.
3. A review of existing or standard solutions to this problem and their limitations.
4. An outline of the proposed new solution.
5. A summary of how the solution was evaluated and what the outcomes of the evaluation were.

You may find the following resources useful:

- `https://www.grammarly.com/blog/write-an-abstract/`
- `https://www.editage.com/insights/manuscript-structure-how-to-convey-your-most-im`
- More useful links: `https://hps.vi4io.org/teaching/ressources/start`

Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work I have used ChatGPT or a similar AI-system as follows:

- ☑ Not at all
- ☐ In brainstorming
- ☐ In the creation of the outline
- ☐ To create individual passages, altogether to the extent of 0% of the whole text
- ☐ For proofreading
- ☐ Other, namely: -

I assure that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

ADT Algebraic Data Types

HPC High-Performance Computing

IR Intermediate Representation

LLVM Low Level Virtual Machine

PBL Problem-Based Learning

RAII Resource acquisition is initialization

1 Introduction

1.1 Motivation

From a programming language perspective, High-Performance Computing (HPC) is dominated by code written C, C++ or Fortran as these provide the low level control and optimization capablilites common in tightly-optimized code. However, as Rust was initially designed as a modern, memory-safe C++ replacement, it could be a valid choice for any kind of performance-critical code.

Instead of just providing yet another taxonomy of successful HPC projects in Rust, this report will rather provide an introduction to the topic on performance engineering in Rust, implicitly covering the current state of the ecosystem while providing an short explaination of each of the common concepts. Instead of just providing an enumeration of techniques, it rather uses Problem-Based Learning (PBL) to introduce the techniques just-in-time when they are relevant, providing a more coherent learning progression.

Problem-Based Learning can be difficult. The problem has to be

- Small enough to fit the scope of a report
- Complex enough to cover most of the concepts of real-life performance engineering
- Interesting enough to keep readers engaged in the topic

For this report, we decided to analyze matrix multiplications. More than just a toy-problem, the matrix multiplication is at the core of all deep learning frameworks. As the parameter count steadily increases into the trillions [1], fast matrix multiplications become evermore important for todays frameworks.

1.2 Rust

Rust [2] is a systems programming language initially released by Mozilla Research in 2015. It was designed as a memory safe alternative for C++ in Servo [3], which is the web rendering engine used in Firefox. Rusts main goal is to provide memory safety while having an on-par performance with other systems languages such as C or C++.

Having memory safety is paramount, as most security issues in traditional C/C++ codebases are a result of using a memory-unsafe language. To quote the overview by Alex Gaynor [4]

- Android [5]: "Our data shows that issues like use-after-free, double-free, and heap buffer overflows generally constitute more than *65%* of High & Critical security bugs in Chrome and Android."
- Android's bluetooth and media components [6]: "Use-after-free (UAF), integer overflows, and out of bounds (OOB) reads/writes comprise *90%* of vulnerabilities with OOB being the most common."

- iOS and macOS [7]: "Across the entirety of iOS 12 Apple has fixed 261 CVEs, 173 of which were memory unsafety. That's *66.3%* of all vulnerabilities." and "Across the entirety of Mojave Apple has fixed 298 CVEs, 213 of which were memory unsafety. That's *71.5%* of all vulnerabilities."
- Chrome [8]: "The Chromium project finds that around *70%* of our serious security bugs are memory safety problems."
- Microsoft [9]: *"∼70%* of the vulnerabilities Microsoft assigns a CVE each year continue to be memory safety issues"
- Firefox's CSS subsystem [10]: "If we'd had a time machine and could have written this component in Rust from the start, 51 (*73.9%*) of these bugs would not have been possible."
- Ubuntu's Linux kernel [11]: "*65%* of CVEs behind the last six months of Ubuntu security updates to the Linux kernel have been memory unsafety."

Furthermore, it is now adapted by many big tech firms such as Amazon [12], Google [13], Meta [14], and Microsoft [15]. Lastly, in december 2022, it became the first language other than C and Assembly supported for Linux kernel development [16].

1.2.1 Why Rust is a good fit for HPC

Basically, one can think of Rust as a modern dialect of C++ enforced by the compiler. It uses Resource acquisition is initialization (RAII) internally to ensure memory safety, while references are roughly equivalent to `std::unique_ptr`.

Especially relevant is the great interoperability with other languages. It supports easy integration with C++ using `bindgen` [17], which is developed by the Rust core team. Rust also allows for easy embedding into Python code using `PyO3` [18], allowing for high-performant native extensions.

Furthermore, it allows for very low level control, even to the extend of bare metal deployment support. Due to Rusts aforementioned RAII-alike memory management model, the runtime has no need for a garbage collector. One can even bring their own memory allocator and do raw pointer arithmetic if required. Lastly, Rust supports architecture based conditional compilation which makes it possible to write fast programs leveraging modern CPU instructions while providing portable alternatives. To support bare metal, OS-less development, Rust's standard library is split into 3 tiers:

- `core`: The `core` library provides essential types and functionality that do not require heap memory allocation.
- `alloc`: The `alloc` library builds upon the `core` library but expects heap-allocations, thus supporting things such as dynamically sized vectors.
- `std`: The `std` library is the highest-level tier, requiring not only a memory allocator but also several OS capablilites such as I/O management.

Although Rust itself is a relatively new language, its compiler supports most modern compiler optimizations. This is possible through Low Level Virtual Machine (LLVM). Instead of producing native assembly for all architectures, the compiler just provides a LLVM frontend generating LLVM Intermediate Representation (IR) which then gets translated to native code by LLVM.

Lastly, it supports many modern functional concepts such as immutability by default, flat traits instead of deep inheritance, exhaustive pattern matching with Algebraic Data Typess (ADTs) sum types as well as providing alternatives to nullability, which is commonly known as the billion dollar mistake [19]. Its language design is in fact so popular that according to the yearly StackOverflow surveys it was voted as the most loved language for the 7th year in the row [20].

1.3 Quadratic Matrix Multiplication

Let $A, B \in \mathbb{R}^{n \times n}, n \in \mathbb{N}$. Then $C \in \mathbb{R}^{n \times n}$ is defined as

$$C_{ij} := \sum_{k=1}^{n} A_{ik} \cdot B_{kj}.$$

One can think of C_{ij} as the dot product of the i-th row of A and the j-th column of B.

1.4 Structure

This report is structured as follows: Section 2 will explore a simplified version of the matrix multiplication problem where the dimension is fixed. Here, the focus will be set on microbenchmarking, full application benchmarking, and assembly analysis. Section 3 will then explore the full matrix multiplication, exploring the topics of profiling, compiler optimizations, cache oblivious as well as how to benchmark in noisy environments. Section 4 will provide a short introduction of parallelism. Lastly, section 5 concludes this report by providing an overview of all shown tools as well as further ressources.

2 Fixed Size Matrix Multiplication

To start off with a simplified problem, this section focuses on a fixed quadratic matrix size of $n = 3$. Using the mathematical definition, this can be trivially implemented:

```rust
fn matmul(a: Vec<Vec<f32>>, b: Vec<Vec<f32>>) -> Vec<Vec<f32>> {
    let mut result = vec![vec![0.0; 3]; 3];
    for i in 0..3 {
        for j in 0..3 {
            for k in 0..3 {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    result
}
fn driver_code(a: Vec<Vec<f32>>, b: Vec<Vec<f32>>, c: Vec<Vec<f32>>)
    -> Vec<Vec<f32>> {
    matmul(matmul(a, b), c) // D := A * B * C
}
```

Listing 1: Naive implementation of a 3×3 matrix multiplication.

The `Vec` arguments are currently passed as a call-by-value, which means that the whole vector gets copied onto the function's stack. Intuitively, this could be improved by using call-by-reference semantics, which just copies the pointer instead of the underlying data. Theoretically, this should result in a performance improvement. In reality, it is very hard to predict actual performance. Thus, some benchmarking is required. In order to measure the performance, either microbenchmarking or full application benchmarking can be used.

2.1 Microbenchmarking

Microbenchmarking is the performance evaluation of small isolated functions. In the Rust ecosystem, there are two obvious solutions for microbenchmarking: Rust's native `cargo bench` as well as `criterion.rs`, which is the modern canonical benchmark library.

Native Benchmarking Cargo, Rust's package manager, supports benchmarking natively through the `cargo bench` [21] subcommand. Unfortunately, this is still experimental, thus only part of the unstable nightly Rust versions. Furthermore, no clear roadmap to stability exists [22].

`cargo bench` is a very lightweight microbenchmarking solution. It provides no integrated regression testing nor any kind of visualization or plotting. The 3rd-party `cargo-benchcmp` [23] utility can be used to compare different benchmarks.

Criterion The other solution is `criterion.rs` [24], which is also available in stable Rust. It uses basic statistical outlier detection to measure regressions and their significance. Furthermore, it blocks constant folding using the `criterion::black_box`, which is described as a "function that is opaque to the optimizer, used to prevent the compiler from optimizing away computations in a benchmark" [25]. It automatically generates HTML reports with plots using `gnuplot`. For benchmark comparisons, the `cargo-critcmp` [26]

program can be used.

As there is currently no active development in `cargo bench`, criterion should always be the prefered solution for microbenchmarking.

2.2 Full Application Benchmarking

There are several solutions for benchmarking whole applications, especially as they are usually agnostic to the application's programming language. But to stick to the modern Rust ecosystem, this report will focus on Hyperfine [27], a very actively developed command-line benchmarking tool written in Rust.

From a simplified perspective, full application benchmarking is quite trivial. First, take a timestamp of the current time. Then, run the command to be benchmarked. Afterwards, take a new timestamp. The time delta is the benchmark time. But beyond this core functionality, Hyperfine supports many important and fundamental features for proper benchmarking and analysis.

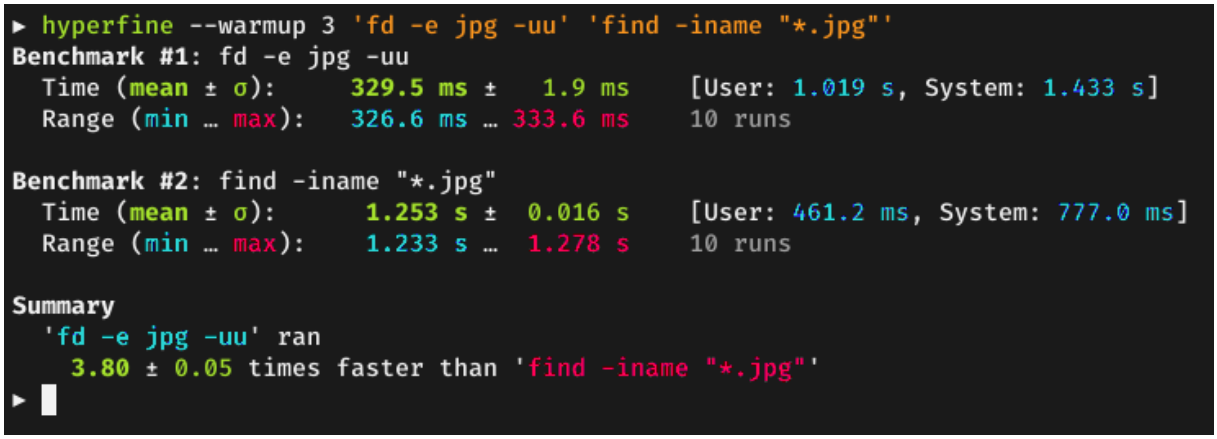

Figure 1: An example picture of Hyperfines output comparing `fd` and `find` [27]

Firstly, Hyperfine supports out of the box statistical analysis and outlier detection. Since it can assume that the program run times are approximately equal, benchmark times are normal distributed. Thus, by fitting a normal distribution over all runs and computing its confidence interval, it can detect any outliers. Secondly, it allows for warmup runs and cache-clearing commands [1] between each run. Warmup runs are useful to fill caches such as the page cache for disk I/O. Lastly, it supports further analysis by providing an export to various formats, such as CSV, JSON, Markdown or AsciiDoc, which can then be analyzed programmatically. Hyperfines repository contains several python scripts for basic visualization [29], which can be used as a starting point for further analysis.

[1] such as `echo 1 > /proc/sys/vm/drop_caches` to free the page cache [28].

References

[1] Vitalii Shevchuk. *GPT-4 Parameters Explained: Everything You Need to Know*. Medium. July 17, 2023. URL: https://levelup.gitconnected.com/gpt-4-parameters-explained-everything-you-need-to-know-e210c20576ca (visited on 08/15/2023).

[2] *Rust Programming Language*. URL: https://www.rust-lang.org/ (visited on 08/15/2023).

[3] The Servo Project Developers. *Servo, the parallel browser engine*. Servo. URL: https://servo.org/ (visited on 08/15/2023).

[4] Alex Gaynor. *What science can tell us about C and C++'s security · Alex Gaynor*. URL: https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/ (visited on 08/15/2023).

[5] *Detecting Memory Corruption Bugs With HWASan*. Android Developers Blog. URL: https://android-developers.googleblog.com/2020/02/detecting-memory-corruption-bugs-with-hwasan.html (visited on 08/15/2023).

[6] *Queue the Hardening Enhancements*. Google Online Security Blog. URL: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html (visited on 08/15/2023).

[7] *Memory Unsafety in Apple's Operating Systems*. URL: https://langui.sh/2019/07/23/apple-memory-safety/ (visited on 08/15/2023).

[8] *Memory safety*. URL: https://www.chromium.org/Home/chromium-security/memory-safety/ (visited on 08/15/2023).

[9] *A proactive approach to more secure code | MSRC Blog | Microsoft Security Response Center*. URL: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (visited on 08/15/2023).

[10] *Implications of Rewriting a Browser Component in Rust – Mozilla Hacks - the Web developer blog*. Mozilla Hacks – the Web developer blog. URL: https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust (visited on 08/15/2023).

[11] Geoffrey Thomas [@geofft]. *Some unofficial @LazyFishBarrel stats from @alex_gaynor and myself: 65% of CVEs behind the last six months of Ubuntu security updates to the Linux kernel have been memory unsafety.* Twitter. May 26, 2019. URL: https://twitter.com/geofft/status/1132739184060489729 (visited on 08/15/2023).

[12] *Why AWS loves Rust, and how we'd like to help | AWS Open Source Blog*. Section: Announcements. Nov. 24, 2020. URL: https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/ (visited on 08/15/2023).

[13] *Welcome to Comprehensive Rust - Comprehensive Rust*. URL: https://google.github.io/comprehensive-rust/ (visited on 08/15/2023).

[14] *Programming languages endorsed for server-side use at Meta*. Engineering at Meta. July 27, 2022. URL: https://engineering.fb.com/2022/07/27/developer-tools/programming-languages-endorsed-for-server-side-use-at-meta/ (visited on 08/15/2023).

[15] jirehl. *Microsoft Azure CTO Wants to Replace C and C++ With Rust | The Software Report*. Oct. 21, 2022. URL: https://www.thesoftwarereport.com/microsoft-azure-cto-wants-to-replace-c-and-c-with-rust/ (visited on 08/15/2023).

[16] Thomas Claburn. *Linus Torvalds says Rust is coming to the Linux kernel*. URL: https://www.theregister.com/2022/06/23/linus_torvalds_rust_linux_kernel/ (visited on 08/15/2023).

[17] *bindgen*. original-date: 2016-06-22T15:05:51Z. Aug. 15, 2023. URL: https://github.com/rust-lang/rust-bindgen (visited on 08/15/2023).

[18] PyO3 Project and Contributors. *PyO3*. original-date: 2017-05-13T05:22:06Z. Aug. 15, 2023. URL: https://github.com/PyO3/pyo3 (visited on 08/15/2023).

[19] Tony Hoare. *Null References: The Billion Dollar Mistake*. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (visited on 08/28/2023).

[20] *Stack Overflow Developer Survey 2023*. Stack Overflow. URL: https://survey.stackoverflow.co/2023 (visited on 08/28/2023).

[21] *cargo bench - The Cargo Book*. URL: https://doc.rust-lang.org/cargo/commands/cargo-bench.html (visited on 08/28/2023).

[22] *When will benchmark testing be stable?* The Rust Programming Language Forum. Feb. 20, 2019. URL: https://users.rust-lang.org/t/when-will-benchmark-testing-be-stable/25482 (visited on 08/28/2023).

[23] Andrew Gallant. *cargo benchcmp*. original-date: 2016-02-15T01:05:58Z. Aug. 23, 2023. URL: https://github.com/BurntSushi/cargo-benchcmp (visited on 08/28/2023).

[24] *bheisler/criterion.rs: Statistics-driven benchmarking library for Rust*. URL: https://github.com/bheisler/criterion.rs (visited on 08/28/2023).

[25] *black_box in criterion - Rust*. URL: https://docs.rs/criterion/latest/criterion/fn.black_box.html (visited on 08/28/2023).

[26] Andrew Gallant. *critcmp*. original-date: 2018-09-18T21:42:02Z. Aug. 22, 2023. URL: https://github.com/BurntSushi/critcmp (visited on 08/28/2023).

[27] David Peter. *hyperfine*. Version 1.16.1. original-date: 2018-01-13T15:49:54Z. Mar. 2023. URL: https://github.com/sharkdp/hyperfine (visited on 08/29/2023).

[28] *Documentation for /proc/sys/vm/*. URL: https://www.kernel.org/doc/Documentation/sysctl/vm.txt (visited on 08/29/2023).

[29] David Peter. *hyperfine visualization scripts*. Version 1.16.1. original-date: 2018-01-13T15:49:54Z. Mar. 2023. URL: https://github.com/sharkdp/hyperfine/tree/master/scripts (visited on 08/29/2023).

A Work sharing

If you worked in a group, describe here how you distributed the work and the actual contributions of each peer.

A.1 Hans

...

A.2 Peter

...

B Code samples

This is part of the appendix...