HPS
lquenti.de

Lars Quentin

Rust for HPC Applications

An Practical Introduction in Rust Performance Engineering

29.06.2023

Recent Trends in High-Performance Data Analytics

Overview

Learning Objectives

- Why Rust is a good fit for HPC.
- How to do the following things in Rust:
    - Microbenchmarking
    - Full Application Benchmarking
    - Analyze generated Assembly
    - Compiler Optimizations
    - Statistical Profiling
    - CI benchmarking
    - Parallelism

Why Rust is a good fit for HPC

Why Rust is a good fit for HPC

- Its like modern C++ enforced by the compiler
  - RAII-based memory management
  - References are like `std::unique_ptr`

Why Rust is a good fit for HPC

- Its like modern C++ enforced by the compiler
    - RAII-based memory management
    - References are like `std::unique_ptr`
- Great Python / C++ interoperability

Why Rust is a good fit for HPC

- Its like modern C++ enforced by the compiler
    - RAII-based memory management
    - References are like `std::unique_ptr`
- Great Python / C++ interoperability
- Allows for very low level control; even supports bare metal deployment.

Why Rust is a good fit for HPC

- Its like modern C++ enforced by the compiler
  - RAII-based memory management
  - References are like `std::unique_ptr`
- Great Python / C++ interoperability
- Allows for very low level control; even supports bare metal deployment.
- Mature compiler optimizations through LLVM backend

Why Rust is a good fit for HPC

- Its like modern C++ enforced by the compiler
  - RAII-based memory management
  - References are like `std::unique_ptr`
- Great Python / C++ interoperability
- Allows for very low level control; even supports bare metal deployment.
- Mature compiler optimizations through LLVM backend
- Many modern concepts from functional programming
  - immutability by default
  - Traits/typeclasses instead of inheritance
  - exhaustive pattern matching
  - Algebraic data types
  - No Nullability

Why Rust is a good fit for HPC

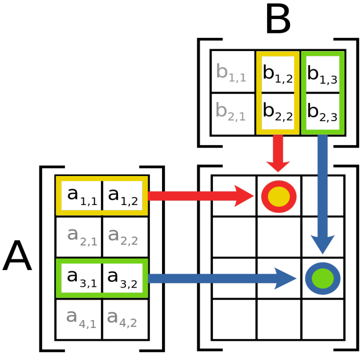
- Its like modern C++ enforced by the compiler
  - RAII-based memory management
  - References are like `std::unique_ptr`
- Great Python / C++ interoperability
- Allows for very low level control; even supports bare metal deployment.
- Mature compiler optimizations through LLVM backend
- Many modern concepts from functional programming
  - immutability by default
  - Traits/typeclasses instead of inheritance
  - exhaustive pattern matching
  - Algebraic data types
  - No Nullability
- Developers’ most loved language for the 7th year according to StackOverflow [1]

Problem: 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

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

i.e. C_{ij} is the dot product of the i-th row of A and the j-th column of B.



[2]

Simplified Problem: 3 × 3 Matrix

First Implementation

```
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
}
```

Microbenchmarking

Native Benchmarking: `cargo bench` [3]

- Not stable (nightly only)
- No regression testing or visualizations
- No clear roadmap to become stable [4]
- `cargo-benchcmp` [5] for comparing benchmarks

Microbenchmarking

Native Benchmarking: `cargo bench` [3]

- Not stable (nightly only)
- No regression testing or visualizations
- No clear roadmap to become stable [4]
- `cargo-benchcmp` [5] for comparing benchmarks

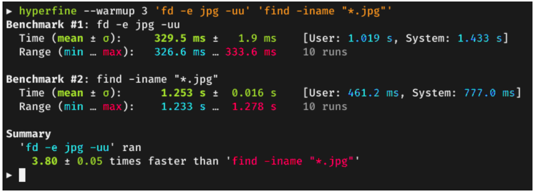
`criterion.rs` [6]

- Uses statistical analysis for regression significance
- Blocks constant folding
- HTML report with plotting through `gnuplot` [7]
- `cargo-critcmp` for comparing benchmarks [8]

Benchmarking Full Applications

Hyperfine [9]

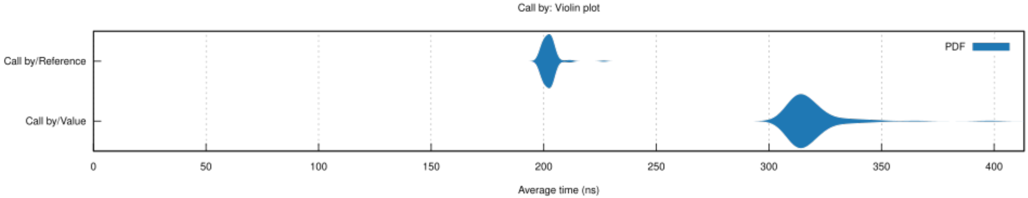
- Statistical analysis / outlier detection
- Warmup runs
- Cache clearing commands available
- Export to different formats such as JSON or CSV
- Supports parametrized benchmarks
- Various Pythonscripts for visualization



```
▶ hyperfine --warmup 3 'fd -e jpg -uu' 'find -iname "*.jpg"'
Benchmark #1: fd -e jpg -uu
  Time (mean ± σ):     329.5 ms ±   1.9 ms    [User: 1.019 s, System: 1.433 s]
  Range (min … max):   326.6 ms … 333.6 ms    10 runs

Benchmark #2: find -iname "*.jpg"
  Time (mean ± σ):      1.253 s ±  0.016 s    [User: 461.2 ms, System: 777.0 ms]
  Range (min … max):    1.233 s …  1.278 s    10 runs

Summary
  'fd -e jpg -uu' ran
    3.80 ± 0.05 times faster than 'find -iname "*.jpg"'
▶ █
```

[9]

Benchmarking Results (-O3)



	Mean	**Std. Dev**	**Median**
Call By Reference	202.31 ns	3.5063 ns	201.96 ns
Call By Value	318.48 ns	12.173 ns	314.59 ns

Total Improvements:

Mean: 57.42%

Median: 55.77%

Next Improvement: Static Stack Arrays

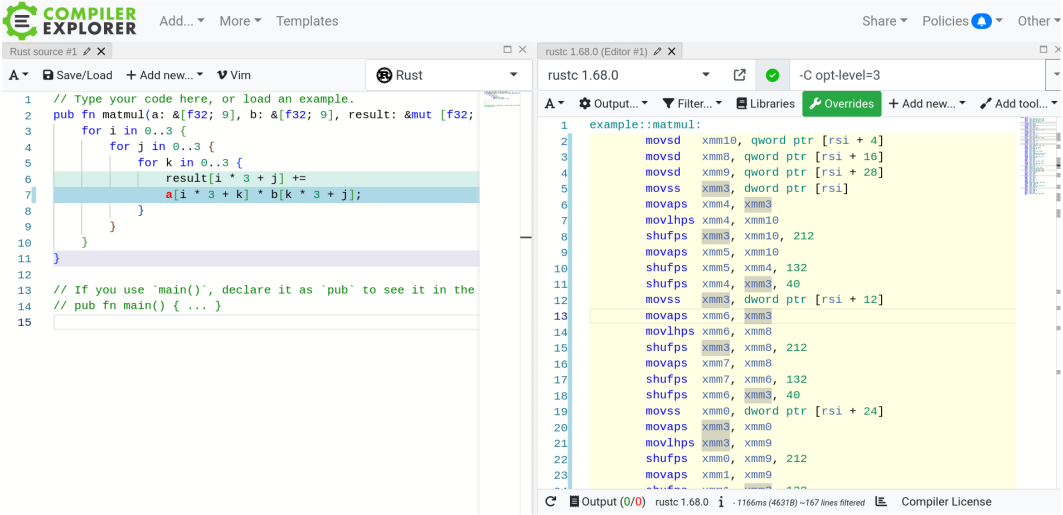
Static Stack Arrays

```
pub fn matmul(a: &[[f32; 3]; 3], b: &[[f32; 3]; 3],
              result: &mut [ [f32; 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];
            }
}}}
```

	Mean	**Std. Dev**	**Median**
Call By Value	318.48 ns	12.173 ns	314.59 ns
Static Arrays	8.0685 ns	254.42 ps	8.0121 ns

Total Improvements:
Mean: 3847.2%
Median: 3826.44%

Assembly 1: Compiler Explorer [10]



rustc 1.68.0

-C opt-level=3

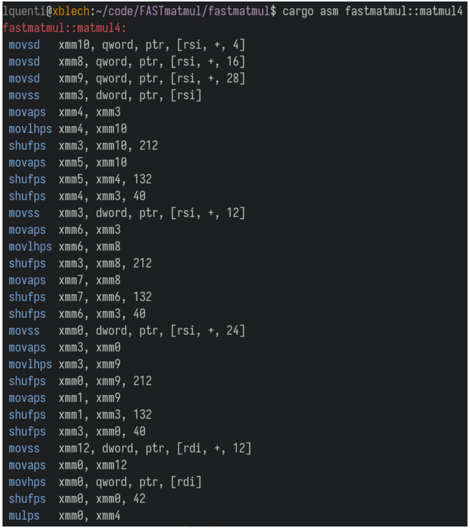
```rust
// Type your code here, or load an example.
pub fn matmul(a: &[f32; 9], b: &[f32; 9], result: &mut [f32;
    for i in 0..3 {
        for j in 0..3 {
            for k in 0..3 {
                result[i * 3 + j] +=
                    a[i * 3 + k] * b[k * 3 + j];
            }
        }
    }
}

// If you use `main()`, declare it as `pub` to see it in the
// pub fn main() { ... }
```

```
example::matmul:
        movsd   xmm10, qword ptr [rsi + 4]
        movsd   xmm8, qword ptr [rsi + 16]
        movsd   xmm9, qword ptr [rsi + 28]
        movss   xmm3, dword ptr [rsi]
        movaps  xmm4, xmm3
        movlhps xmm4, xmm10
        shufps  xmm3, xmm10, 212
        movaps  xmm5, xmm10
        shufps  xmm5, xmm4, 132
        shufps  xmm4, xmm3, 40
        movss   xmm3, dword ptr [rsi + 12]
        movaps  xmm6, xmm3
        movlhps xmm6, xmm8
        shufps  xmm3, xmm8, 212
        movaps  xmm7, xmm8
        shufps  xmm7, xmm6, 132
        shufps  xmm6, xmm3, 40
        movss   xmm0, dword ptr [rsi + 24]
        movaps  xmm3, xmm0
        movlhps xmm3, xmm9
        shufps  xmm0, xmm9, 212
        movaps  xmm1, xmm9
```

Output (0/0) rustc 1.68.0 - 1166ms (4631B) ~167 lines filtered

Assembly 2: `cargo-show-asm` [11]

- Allows to view Assembly or LLVM-IR
- Can query single functions
- Can also resolve trait implementations



```
lquenti@xblech:~/code/FASTmatmul/fastmatmul$ cargo asm fastmatmul::matmul4
fastmatmul::matmul4:
 movsd   xmm10, qword, ptr, [rsi, +, 4]
 movsd   xmm8, qword, ptr, [rsi, +, 16]
 movsd   xmm9, qword, ptr, [rsi, +, 28]
 movss   xmm3, dword, ptr, [rsi]
 movaps  xmm4, xmm3
 movlhps xmm4, xmm10
 shufps  xmm3, xmm10, 212
 movaps  xmm5, xmm10
 shufps  xmm5, xmm4, 132
 shufps  xmm4, xmm3, 40
 movss   xmm3, dword, ptr, [rsi, +, 12]
 movaps  xmm6, xmm3
 movlhps xmm6, xmm8
 shufps  xmm3, xmm8, 212
 movaps  xmm7, xmm8
 shufps  xmm7, xmm6, 132
 shufps  xmm6, xmm3, 40
 movss   xmm0, dword, ptr, [rsi, +, 24]
 movaps  xmm3, xmm0
 movlhps xmm3, xmm9
 shufps  xmm0, xmm9, 212
 movaps  xmm1, xmm9
 shufps  xmm1, xmm3, 132
 shufps  xmm3, xmm0, 40
 movss   xmm12, dword, ptr, [rdi, +, 12]
 movaps  xmm0, xmm12
 movhps  xmm0, qword, ptr, [rdi]
 shufps  xmm0, xmm0, 42
 mulps   xmm0, xmm4
```

Assembly 3: Loop Unrolling and Function Inlining

Loop Unrolling

- Was already applied in our case
- Tooling: `unroll` [12] provides a macro for creating unrolled rust code.
- For dynamic length loops: `-C llvm-args="-unroll-threshold=N"`
    - Do not apply without benchmarking!

Assembly 3: Loop Unrolling and Function Inlining

Loop Unrolling

- Was already applied in our case
- Tooling: `unroll` [12] provides a macro for creating unrolled rust code.
- For dynamic length loops: `-C llvm-args="-unroll-threshold=N"`
    - Do not apply without benchmarking!

Function Inlining

- Was not applied in our case
- But compiler hints exist: `#[inline(/always/never)]` [13]

Introduction Real Problem

- **Task:** You get introduced to a scientific problem which is too slow.

Introduction Real Problem

- **Task:** You get introduced to a scientific problem which is too slow.
- Why is this so slow?

Introduction Real Problem

- **Task:** You get introduced to a scientific problem which is too slow.
- Why is this so slow?
- How do I figure this out?

Introduction Real Problem

- **Task:** You get introduced to a scientific problem which is too slow.
- Why is this so slow?
- How do I figure this out?
- **Solution: Profiling**

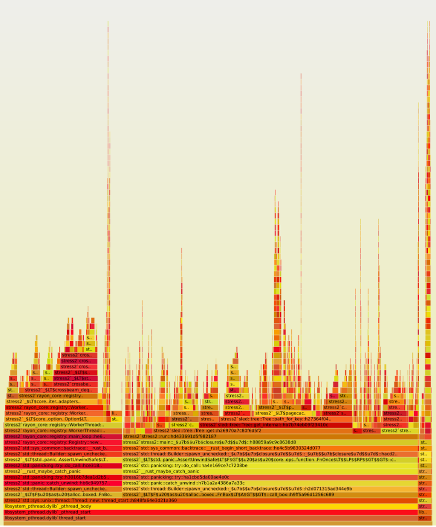
Profiling

- Since Rust produces normal binaries, most profilers just work.
  - Including:
    - `perf` [14]
    - `cachegrind` [15]
    - ...
  - `rustfilt` [16] can demangle all symbols.

Profiling

- Since Rust produces normal binaries, most profilers just work.
  - Including:
    - `perf` [14]
    - `cachegrind` [15]
    - ...
  - `rustfilt` [16] can demangle all symbols.
- Here, we will use `cargo-flamegraph` [17] and later `iai` [18].

Cargo flamegraph



- Statistical Profiler
  - Interrupts randomly
  - Looks at the stack
  - Then it can approximate how much time is spent in each function
- Uses `perf` internally

Our Result: Lets assume the problem was a quadratic $n \times n$ matrix multiplication!

For the benchmarks, we assume $n = 1024$.

Unoptimized code

Unoptimized Version

```rust
fn matmul1(a: Vec<Vec<f32>>, b: Vec<Vec<f32>>) -> Vec<Vec<f32>> {
    let n = a.len();
    let mut result = vec![vec![0.0; n]; n];
    for i in 0..n {
        for j in 0..n {
            for k in 0..n {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    result
}
```

Applying our previous knowledge

First optimized Version

```
fn matmul2(a: &[f32], b: &[f32]) -> Vec<f32> {
    let n = (a.len() as f32).sqrt() as usize;
    let mut result = vec![0.0; n * n];
    for i in 0..n {
        for j in 0..n {
            for k in 0..n {
                result[i * n + j] += a[i * n + k] * b[k * n + j];
            }
        }
    }
    result
}
```

Compiler Optimizations!

- Using a Release build (-03)
- LLVM Link Time Optimization (LTO)
- Using the native Architecture
- LLVM Single Code Unit

Out of scope:

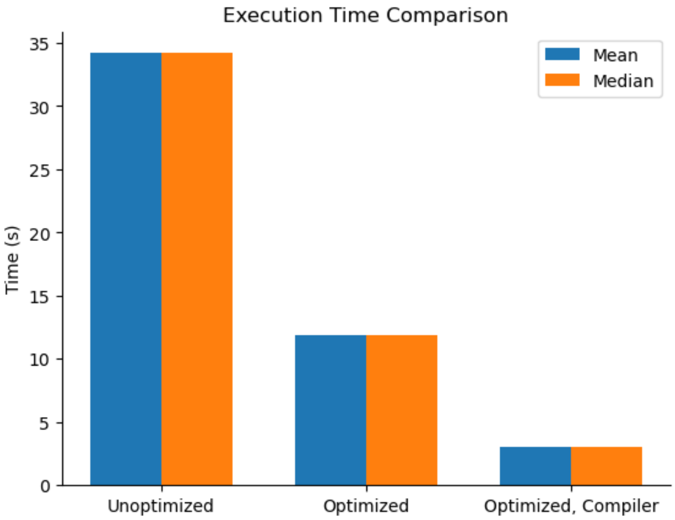
- Profile Guided Optimization (PGO)

```toml
[profile.release]
opt-level = 3
lto = true
codegen-units = 1
```

First Results

	Mean	**Std. Dev**	**Median**	**Mean Improvement**
Unoptimized	$34.192s$	$76.206ms$	$34.177s$	
Optimized	$11.875s$	$96.287ms$	$11.856s$	187.93%
Compiler	$3.0426s$	$13.525ms$	$3.0450s$	1023.78%

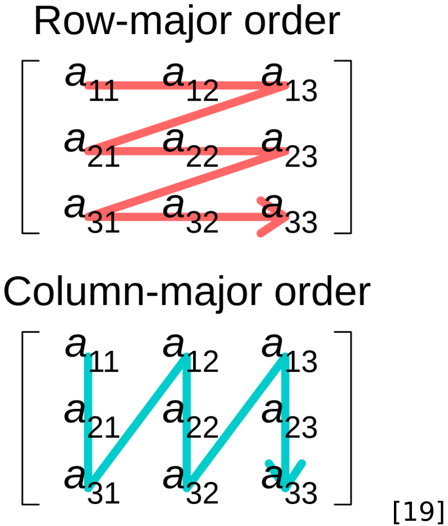
First Results (cont.)



Cache-oblivious algorithms [20]

Standard Matrix Multiplication $A \cdot B$

- Traverses A row-major order
- Traverses B column-major order
  - Every step of B we get a cache miss
- Solution: Transpose B
- C_{ij} becomes row A_i times **row** B_j
- Requires $\Theta(n^2)$ precompute.
  - Does it improve speed?
  - Does it reduce cache misses?



Cache-oblivious algorithms (cont.)

Does it improve Speed?

- Lets benchmark it:

	Mean	**Median**
Row-major	2.9668s	2.9661s
Col-major	2.1689s	2.1686s

Cache-oblivious algorithms (cont.)

Does it improve Speed?

- Lets benchmark it:

	Mean	**Median**
Row-major	2.9668s	2.9661s
Col-major	2.1689s	2.1686s

Does it reduce cache misses?

- This is more complex
- For this, we have to simulate the caches
- This can be done using cachegrind [15].

Iai [18]

- Based on cachegrind
- Emulating the CPU and its caches
- Precise single-shot measurements
- Main usecase in CI systems

Iai [18]

- Based on cachegrind
- Emulating the CPU and its caches
- Precise single-shot measurements
- Main usecase in CI systems

```
iai_normal
  Instructions:          13970975862
  L1 Accesses:           17192372607
  L2 Accesses:            1074884737
  RAM Accesses:               262191
  Estimated Cycles:      22575972977

iai_transpose
  Instructions:           9144912377
  L1 Accesses:           12838193034
  L2 Accesses:              68158189
  RAM Accesses:               328137
  Estimated Cycles:      13190468774
```

- Unclear, requires further investigation.

SIMD

Old API

- Experimental only; **Unsafe**
- Low level Platform-Specific structs
- Direct Intrincics translation
- Intel Documentation [21]:
  - `__mmask32 _kadd_mask32 (__mmask32 a, __mmask32 b)`
- Rust port [22]:
  - `unsafe fn _kadd_mask32(a: __mmask32, b:  __mmask32) -> __mmask32`

SIMD

Old API

- Experimental only; **Unsafe**
- Low level Platform-Specific structs
- Direct Intrincics translation
- Intel Documentation [21]:
  - `__mmask32 _kadd_mask32 (__mmask32 a, __mmask32 b)`
- Rust port [22]:
  - `unsafe fn _kadd_mask32(a: __mmask32, b:  __mmask32) -> __mmask32`

Portable SIMD

- Experimental only, **Safe**
- Generalized on **bit width level**
  - `std::simd::{f32x8, f64x4, i32x8}`
- Conditional Compilation [23] with
  - `#[cfg(target_arch="x86_64")]`
  - `#[cfg(target_feature="aes")]`
- Conditional Execution [24] with `std::is_x86_feature_detected` (runtime)

SIMD

Old API

- Experimental only; **Unsafe**
- Low level Platform-Specific structs
- Direct Intrincics translation
- Intel Documentation [21]:
  - `__mmask32 _kadd_mask32 (__mmask32 a, __mmask32 b)`
- Rust port [22]:
  - `unsafe fn _kadd_mask32(a: __mmask32, b: __mmask32) -> __mmask32`

Portable SIMD

- Experimental only, **Safe**
- Generalized on **bit width level**
  - `std::simd::{f32x8, f64x4, i32x8}`
- Conditional Compilation [23] with
  - `#[cfg(target_arch="x86_64")]`
  - `#[cfg(target_feature="aes")]`
- Conditional Execution [24] with `std::is_x86_feature_detected` (runtime)

Unable to port due to missing documentation

Rayon [25]

- High-Level Parallelism Library
- Gurantees **data-race freedom**
- Main Feature: Parallel Iterators
  - Just replace `.iter()` with `.par_iter()`
  - Same functionality as sequential **if** the iterator has no side effects
  - Support for High-Level functions
    - `.map()`, `.filter()`, `.reduce()` ...
  - Low level primitives such as `.join()`:
    - `.join(|| a(), || b())`
    - **May** run in parallel
    - Based on if idle cores are available

Rayon (cond.)

Unported Code (no transpose)

```rust
fn matmul3(a: &[f32], b: &[f32], result: &mut [f32], n: usize) {
    for i in 0..n {
        for j in 0..n {
            for k in 0..n {
                result[i * n + j] += a[i * n + k] * b[k * n + j];
            }
        }
    }
}
```

Rayon (cond.)

Ported to Iterators

```
fn matmul3(a: &[f32], b: &[f32], result: &mut [f32], n: usize) {
    result.iter_mut().enumerate().for_each(|(idx, res)| {
        let i = idx / n;
        let j = idx % n;
        *res = (0..n).map(|k| a[i * n + k] * b[k * n + j]).sum();
    });
}
```

Rayon (cond.)

Ported to Iterators **and Parallelized!**

```
fn matmul3(a: &[f32], b: &[f32], result: &mut [f32], n: usize) {
    result.par_iter_mut().enumerate().for_each(|(idx, res)| {
        let i = idx / n;
        let j = idx % n;
        *res = (0..n).map(|k| a[i * n + k] * b[k * n + j]).sum();
    });
}
```

Further Ressources

- The Rust Performance Book [13]
  - Bounds checking
  - I/O
  - Perf linter clippy
  - Type sizes

Further Ressources

- The Rust Performance Book [13]
    - Bounds checking
    - I/O
    - Perf linter clippy
    - Type sizes
- Algorithmica: Algorithms for Modern Hardware [26]

Further Ressources

- The Rust Performance Book [13]
  - Bounds checking
  - I/O
  - Perf linter clippy
  - Type sizes
- Algorithmica: Algorithms for Modern Hardware [26]
- rsmpi [27]
  - Pure Rust implementation
  - Compatible with
    - OpenMPI
    - MPICH
    - MS-MPI (Windows)

Summary

- Rust is viable for HPC, although still experimental

Summary

- Rust is viable for HPC, although still experimental
- There are many flags for compiler tuning

Summary

- Rust is viable for HPC, although still experimental
- There are many flags for compiler tuning
- The following tools are available for HPC:

Summary

- Rust is viable for HPC, although still experimental
- There are many flags for compiler tuning
- The following tools are available for HPC:

Topic	**Tool**
Microbenchmarking	Criterion
Application Benchmarking	Hyperfine
Assembly Generation	Compiler Explorer, cargo show-asm
Loop Unrolling	`unroll`, Compiler Arguments
Function Inlining	`#[inline]`
Statistical Profiling	`cargo-flamegraph`
CI benchmarking	`iai`
SIMD	`std::simd`, `core::arch`
Intra-Node parallelism	rayon

References I

Stack Overflow Developer Survey 2022. Stack Overflow. URL: `https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022` (visited on 06/08/2023).

File:Matrix multiplication diagram svg:User:BilouSee below. *Schematic depiction of the matrix product AB of two matrices A and B.* Oct. 4, 2010. URL: `https://commons.wikimedia.org/wiki/File:Matrix_multiplication_diagram_2.svg` (visited on 06/08/2023).

cargo bench - The Cargo Book. URL: `https://doc.rust-lang.org/cargo/commands/cargo-bench.html` (visited on 06/08/2023).

Stabilize #[bench] and Bencher? · Issue #66287 · rust-lang/rust. GitHub. URL: `https://github.com/rust-lang/rust/issues/66287` (visited on 06/08/2023).

Andrew Gallant. *cargo benchcmp*. May 14, 2023. URL: `https://github.com/BurntSushi/cargo-benchcmp` (visited on 06/08/2023).

Brook Heisler. *Criterion.rs*. original-date: 2014-05-26T14:14:22Z. June 8, 2023. URL: `https://github.com/bheisler/criterion.rs` (visited on 06/08/2023).

References II

gnuplot. URL: `http://www.gnuplot.info/` (visited on 06/08/2023).

Andrew Gallant. *critcmp*. May 19, 2023. URL: `https://github.com/BurntSushi/critcmp` (visited on 06/08/2023).

David Peter. *hyperfine*. Version 1.16.1. Mar. 2023. URL: `https://github.com/sharkdp/hyperfine` (visited on 06/08/2023).

Matt Godbolt. *Compiler Explorer*. URL: `https://godbolt.org/` (visited on 06/08/2023).

gnzlbg. *cargo-asm*. original-date: 2018-02-13T19:38:49Z. June 6, 2023. URL: `https://github.com/gnzlbg/cargo-asm` (visited on 06/08/2023).

unroll. GitLab. June 6, 2022. URL: `https://gitlab.com/elrnv/unroll` (visited on 06/08/2023).

Nicholas Nethercote. *The Rust Performance Book*. URL: `https://nnethercote.github.io/perf-book/` (visited on 06/08/2023).

Perf Wiki. URL: `https://perf.wiki.kernel.org/index.php/Main_Page` (visited on 06/08/2023).

Valgrind. URL: `https://valgrind.org/` (visited on 06/08/2023).

References III

Ted Mielczarek. *luser/rustfilt*. original-date: 2016-05-13T17:00:31Z. May 19, 2023. URL: `https://github.com/luser/rustfilt` (visited on 06/08/2023).

[cargo-]flamegraph. original-date: 2019-03-07T16:31:30Z. June 8, 2023. URL: `https://github.com/flamegraph-rs/flamegraph` (visited on 06/08/2023).

Brook Heisler. *Iai*. original-date: 2021-01-02T20:54:31Z. June 7, 2023. URL: `https://github.com/bheisler/iai` (visited on 06/08/2023).

Cmglee. *English: Illustration of row- and column-major order by CMG Lee.* URL: `https://commons.wikimedia.org/wiki/File:Row_and_column_major_order.svg` (visited on 06/13/2023).

Matteo Frigo et al. "Cache-Oblivious Algorithms". In: *ACM Transactions on Algorithms* 8.1 ().

Intel Intrinsics Guide. Intel. URL: `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html` (visited on 06/19/2023).

_kadd_mask32 in core::arch::x86_64 - Rust. URL: `https://doc.rust-lang.org/core/arch/x86_64/fn._kadd_mask32.html` (visited on 06/19/2023).

References IV

Conditional compilation - The Rust Reference. URL: `https://doc.rust-lang.org/reference/conditional-compilation.html` (visited on 06/19/2023).

is_x86_feature_detected in std - Rust. URL: `https://doc.rust-lang.org/std/macro.is_x86_feature_detected.html` (visited on 06/19/2023).

Rayon. original-date: 2014-10-02T15:38:05Z. June 19, 2023. URL: `https://github.com/rayon-rs/rayon` (visited on 06/19/2023).

Sergey Slotin. *Algorithmica*. URL: `https://en.algorithmica.org/` (visited on 06/19/2023).

MPI bindings for Rust. original-date: 2015-07-21T20:51:28Z. June 15, 2023. URL: `https://github.com/rsmpi/rsmpi` (visited on 06/19/2023).