Einfacher COTS Performance-Benchmark

R. Grünert
23. November 2023

CPU: i7 4790k

1 Erster Benchmark

```rust
use clap::Parser;
use std::time::Instant;


/// Simple Benchmark
#[derive(Parser, Debug)]
#[command(author, version, about, long_about = None)]
struct Args {
    /// Number of iterations to perform
    #[arg(short, long)]
    #[clap(value_parser = clap::value_parser!(u32).range(1..=4000000000))]
    num_iterations: u32,
}


fn main() {
    let args = Args::parse();
    let mut result: f32 = 1.0;
    let now = Instant::now();

    for _n in 0..args.num_iterations {
        result = 1.00001 * result;
    }

    let elapsed = now.elapsed();

    println!("done! {} time: {:?}", result, elapsed);
    println!("GFLOPS: {}", (args.num_iterations as f64) / elapsed.as_secs_f64() * 10e-9);
}
```

Abbildung 1: Erstes Benchmarkprogramm.

Für einen ersten Performancetest wurde das in Listing 1 gezeigte Programm in Rust geschrieben. Es berechnet lediglich die durch Aufrufargumente übergebene Anzahl an (32-bit-) Fließpunktmultiplikationen (Zeile 22). In diesem Fall wurde `num_iterations=40000000000` gewählt.

Das Programm wurde mit zwei unterschiedlichen Optimierungsstufen kompiliert [2]:

- Sog. „release“-Modus (`opt-level 3`)
- Keine Optimierung (`opt-level 0`)

opt-level	Zeit / s	Perf. / GFLOPS
0	14.529	2.753
3	4.675	8.557

Tabelle 1: Ergebnisse des ersten Benchmarks.

2 Zweiter Benchmark

```rust
use clap::Parser;
use std::time::Instant;
use rand::prelude::*;

/// Simple Benchmark
#[derive(Parser, Debug)]
#[command(author, version, about, long_about = None)]
struct Args {
    /// Number of iterations to perform
    #[arg(short, long)]
    #[clap(value_parser = clap::value_parser!(u32).range(1..=4000000000))]
    num_iterations: u32,
}


fn main() {
    let args = Args::parse();
    let now = Instant::now();

    let mut rng = rand::thread_rng();

    let array1: [f32; 4] = [rng.gen::<f32>(), rng.gen::<f32>(), rng.gen::<f32>(), rng.gen::<f32>()];
    let mut array2: [f32; 4] = [rng.gen::<f32>(), rng.gen::<f32>(), rng.gen::<f32>(),
    ↪  rng.gen::<f32>()];
    let mut array3: [f32; 4] = [0.0, 0.0, 0.0, 0.0];

    for _n in 0..args.num_iterations {
        array3[0] = array1[0] + array2[0];
        array3[1] = array1[1] + array2[1];
        array3[2] = array1[2] + array2[2];
        array3[3] = array1[3] + array2[3];

        array2[0] = array1[0] * array3[0];
        array2[1] = array1[1] * array3[1];
        array2[2] = array1[2] * array3[2];
        array2[3] = array1[3] * array3[3];
    }

    let elapsed = now.elapsed();
    const NUM_OPS: u32 = 8;

    println!("array: {:?}", array1);
    println!("array2: {:?}", array2);
    println!("array3: {:?}", array3);
    println!("time: {:?}", elapsed);
    println!("GFLOPS: {}", (args.num_iterations as f64) / elapsed.as_secs_f64() * (NUM_OPS as f64) *
    ↪  10e-9);
}
```

Abbildung 2: Zweites Benchmarkprogramm.

In einem weiteren Test wurde das erste Programm modifiziert, um parallele Vektoroperationen zu testen. Erneut wurde mit zwei Optimierungsstufen kompiliert.

Lässt man sich den generierten Assembler-Code dieses Programmes ausgeben[1], findet man im Fall der optimierten Kompilierung die Schleife der Zeilen 26 bis 36 wieder (Abb. 3). Man erkennt partielles Loop-Unrolling sowie die SIMD-Operationen `mulps` und `addps`, welche die Addition bzw. Multiplikation

[1] `cargo rustc --release -- --emit asm`

der 4 Arraywerte in einer einzigen 128-bit Operation ermöglichen (siehe [1]). Dies ist natürlich ein konstruiertes Szenario, welches die SIMD-Operationen bevorzugt, was sich auch in der erhöhten Performance wiederspiegelt (Tabelle 2).

```
.LBB110_207:
  addps  %xmm3, %xmm0
  mulps  %xmm3, %xmm0
  addps  %xmm3, %xmm0
  movaps  %xmm3, %xmm1
  mulps  %xmm0, %xmm1
  movaps  %xmm0, %xmm2
  movaps  %xmm1, %xmm0
  addl  $-2, %eax
  jne  .LBB110_207
.LBB110_208:
```

Abbildung 3: Snippet aus dem optimierten Assembler-Code des zweiten Benchmarkprogrammes.

opt-level	Zeit / s	Perf. / GFLOPS
0	43.741	7.316
3	7.867	40.679

Tabelle 2: Ergebnisse des zweiten Benchmarks.

Literatur

Web

[1] Wikipedia, The Free Encyclopedia. *Streaming SIMD Extensions*. 2023. URL: https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions (besucht am 23. 11. 2023).

[2] The Rust Programming Language. *The Cargo Book: Profiles*. 2023. URL: https://doc.rust-lang.org/cargo/reference/profiles.html#opt-level (besucht am 23. 11. 2023).