Advanced Systems Lab

Assignment 2

Albert Cerfeda (20-980-272)

Contents

14.03.2024
Eidgenössische Technische Hochschule Zürich
D-INFK Department
Switzerland

2.1 Short project info (5pts)

2.2 Optimization blockers (30pts)

a)

Figure 1 shows the plot of the runtime in cycles of the various implementations of the original function. These benchmarks were performed on an *Intel Xeon Silver 4210 @ 2.20 GHz Cascade Lake* and the code was compiled using *GCC 11.2.1* with flags `-O3`, `-march=skylake`, `-mno-fma` and `-fno-tree-vectorize`.

Implementation	Baseline	V1	V2	V3
Runtime (cycles)	876724.0	485556.0	488395.0	45202.6

Table 1: Comparison of the function implementations

Baseline
The original provided baseline implementation. It is the slowest of all the implementations.

V1

1. Uses $2x$ loop unrolling on the outer loop, $6x$ for the inner loop, and $4x$ for the final loop.
2. All function invocations are removed, and replaced with direct access to the `data` matrix.
3. This implementation also makes use of the assumption that $n = 100$ for every square $n * n$ matrix, therefore it uses the numeric constant 100 for the matrix bound check.

We notice how removing the function invocation overhead and adding loop unrolling already provides a significant speedup of almost $2x$ over the baseline.

V2

1. In this implementation we simplify the data fetching that uses the modulo operator. Since we are performing $6x$ loop unrolling, we can simplify the modulo 3 and modulo 6 expressions by replacing them with constant values in the range $[0, 5]$.
2. Simplified the `abs(fmax(...))+1` statements in the last loop with the ternary operator.
3. Introduced precomputation of all independent `sqrt` statements outside of the loop body.
4. Each division by a `sqrt` is replaced with a multiplication by the inverse of the `sqrt` value, also precomputed. This was done to avoid costly divisions in the loop body.

V3

1. In this last version we exploit the periodicity of the cosine function to precompute the values of the `cos` function outside of the loop body.
2. Analogously to the previous version, we also precompute the reciprocal of the `cos` values to avoid costly divisions in the loop body.
3. We also merge the ternary operations of the last loop into the first loop.

We notice how precomputing the cosine values yields by far the biggest performance improvement. The last implementation is clearly the fastest, yielding a speedup of **22.2** over the baseline.

b)

The `V3` implementation performs 50 iterations in outer loop and 16 in the inner loop, therefore $W = 19 + 50 * (8 + 16 * (72) + 1 * (40)) = 60019$ flops. The performance is therefore $\pi = \frac{60019}{45202.6} = 1.328$ flops/cycle. **Note:** When evaluating the flops, the `abs` and `cos` functions have been assumed to be 1 floating point operation each.

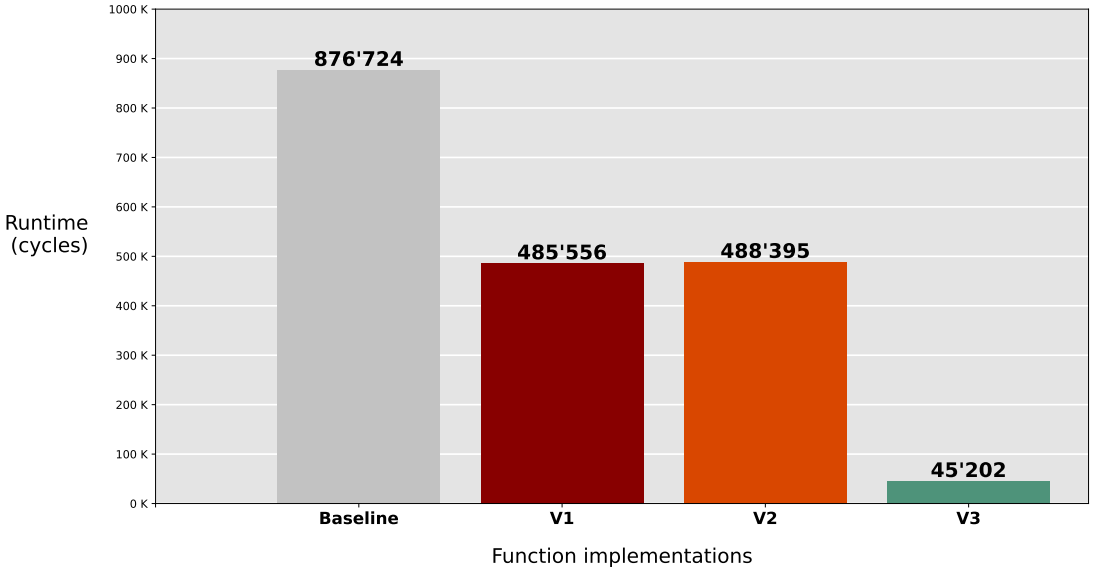

Figure 1: Runtime comparison of the function implementations

c)

The theoretical peak performance is 2 flops/cycles. The performance of the `V3` implementation is 1.328 flops/cycle, which is 66.39% of the theoretical peak performance.

2.3 Microbenchmarks (40 pts)

a)

Yes, the Intel Optimization manual reports that the FP ADD instructions takes 4 cycles with a throughput of 0.5, and indeed our microbenchmark confirms it.
Furhtermore, a latency of 14 cycles and a throughput of 6 cycles for the square root instruction is reported, also confirmed by our microbenchmarks.

b)

The function *foo()* consists of 2 FP multiplication and 1 square root operation. The reported (and mesured) latencies are 4 cycles for mults and 18 for the sqrt operation. This results in a theoretical latency of $4(2) + 18(1) = 26$ cycles, which is consistent with our microbenchmarks. The reciprocal throughput is bottlenecked by the troughput of the square root operation, therefore resulting in an overall reciprocal throughput of 6 cycles.

c)

Let us examine function $f_2(a) = a \cdot \sqrt{a}$.
As confirmed by our microbenchmarks, the latency of the multiplication operation is 4 cycles, and the latency of the square root operation is 18 cycles, resulting in an total latency of 22. The reciprocal throughput instead does not change from the previous case, and is still 6 cycles.