Using modern CPU instructions to improve LLVM's libc math library

Tue Ly

Overview of LLVM's libc math library

- Goals: Re-implement C99 (C23) standards math functions focusing on:
  - Accuracy
  - Performance
  - Multi-platform support
  - Other use-cases support: code size + memory usage for embedded systems
- Implemented most functions for single precision (float) + few for double
  - Accuracy: correctly rounded for all 4 rounding modes
    - x86-64: linux, windows
    - aarch64: linux, macos
  - Performance: comparable to current glibc (2.35) implementations
- For more information: https://libc.llvm.org/math

Priority: Accuracy

- As accurate as possible, with the ultimate goal to be correctly rounded for all rounding modes:
  - FE_TONEAREST, FE_UPWARD, FE_DOWNWARD, FE_TOWARDZERO
  - Rounding mode is decided by the floating point environment from <fenv.h>
- Correct rounding → Consistency:
  - Bit-identical outputs across platforms, library versions
  - Reduce the toil of updating / integrating libc
  - Floating-point golden tests make sense again
- Testing:
  - Compared outputs with mpfr and other accurate math libraries
  - Exhaustive testings for single precision
  - Hard-to-round cases + fuzz tests for double precision or higher

Priority: Performance

- Pitfalls of IBM Accurate Mathematical Library (link)
  - Extremely high latency for worst-cases
  - Use up to 800 bits of precisions for double precision accurate passes.
- Recent advances in floating point arithmetic
  - Worst-case analysis reduced the required precisions needed down to < 200 bits for double precision.
- Modern CPUs' instructions
  - rounding instructions
  - fused-multiply-adds (FMAs)

Multiplatform support - Compiler Builtins vs. Asm

- Compiler built-ins:
    - Participate in compiler optimizations
    - Circular dependent: __builtin_nan, __builtin_fma, …
    - Platform-dependent: <immintrin.h> on x86-64, <arm_acle.h> on aarch64
    - Limited: no FMA builtins in arm_acle.h

- Assembly:
    - Very low dependency
    - Do not participate in compiler optimizations: fma(a, b, -c) with aarch64
    - Instruction choices

Overview: a math function implementation - sin(x)

- Step 1: range reduction
  - Find k and y such that x = (k + y) * pi/32
  - k = round(x * (32/pi)) → rounding instruction
  - y = x * (32/pi) - k → y = fma(x, 32/pi, -k)

- Step 2: polynomial approximation
  - Approximate: $\sin(y * pi/32) \sim c_1 * y + c_3 * y^3 + c_5 * y^5 + c_7 * y^7$
    $= y * (c_1 + y^2 * (c_3 + y^2 * (c_5 + y^2 * c_7)))$
    $= y * fma(fma(fma(c_7, y^2, c_5), y^2, c_3), y^2, c_1)$
  - Approximate: $\cos(y * pi/32) \sim 1 + c_2 * y^2 + c_4 * y^4 + c_6 * y^6$

- Step 3: combine
  - sin(x) = sin(k * pi/32) * cos(y * pi/32) + cos(k * pi/32) * sin(y * pi/32)
    = fma (...)

Performance summary (single precision vs glibc 2.35)

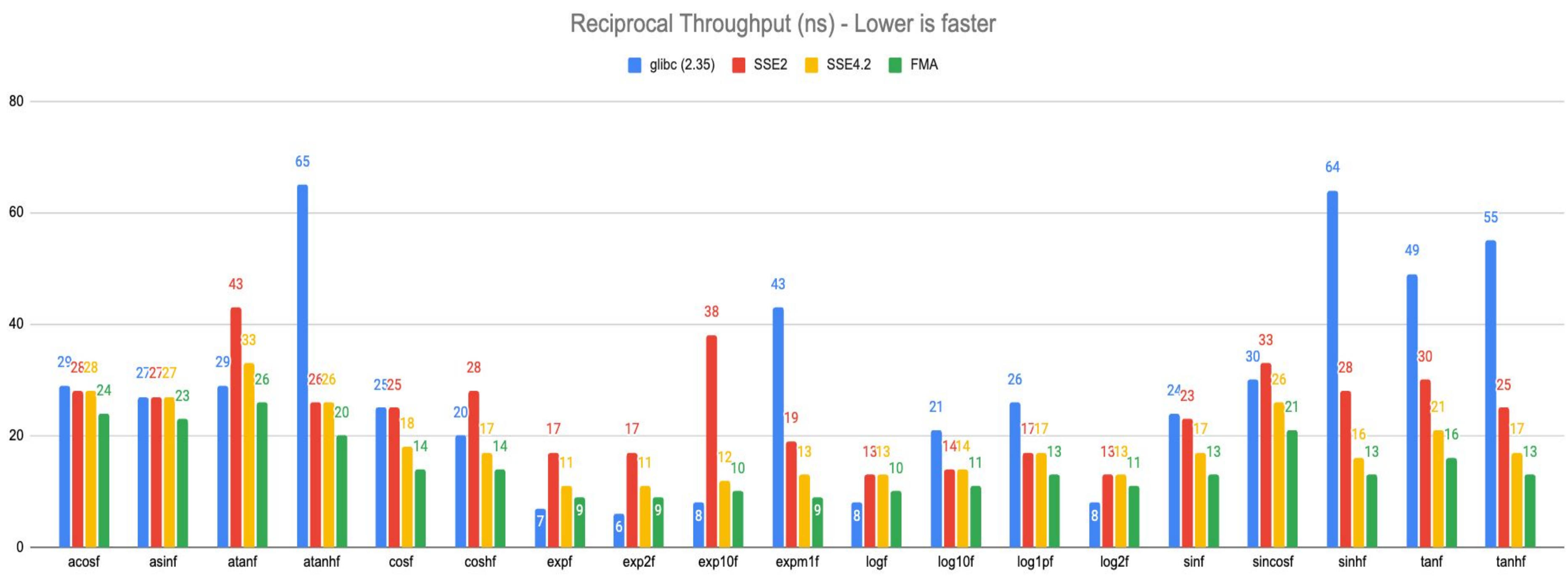
- 19 single precision transcendental functions implemented for float
  - Use `perf.sh` scripts from the CORE-MATH projects (link)
  - Input ranges: link
  - Throughput (reciprocal throughput) - # ops / time in batch
  - Latency - time / single op in isolation
- Testing config:
  - Ryzen 1700 - Ubuntu 22.04 - Clang 14.0
  - Compare SSE2 (default) vs SSE4.2 (rounding) vs AVX2 (rounding + FMA)
- For aarch64:
  - Working on porting CORE-MATH’s `perf.sh` tool to aarch64
  - Utilizing armv8, other extensions are to be explored
  - Expected similar results as x86-64

Performance summary (single precision vs glibc 2.35)

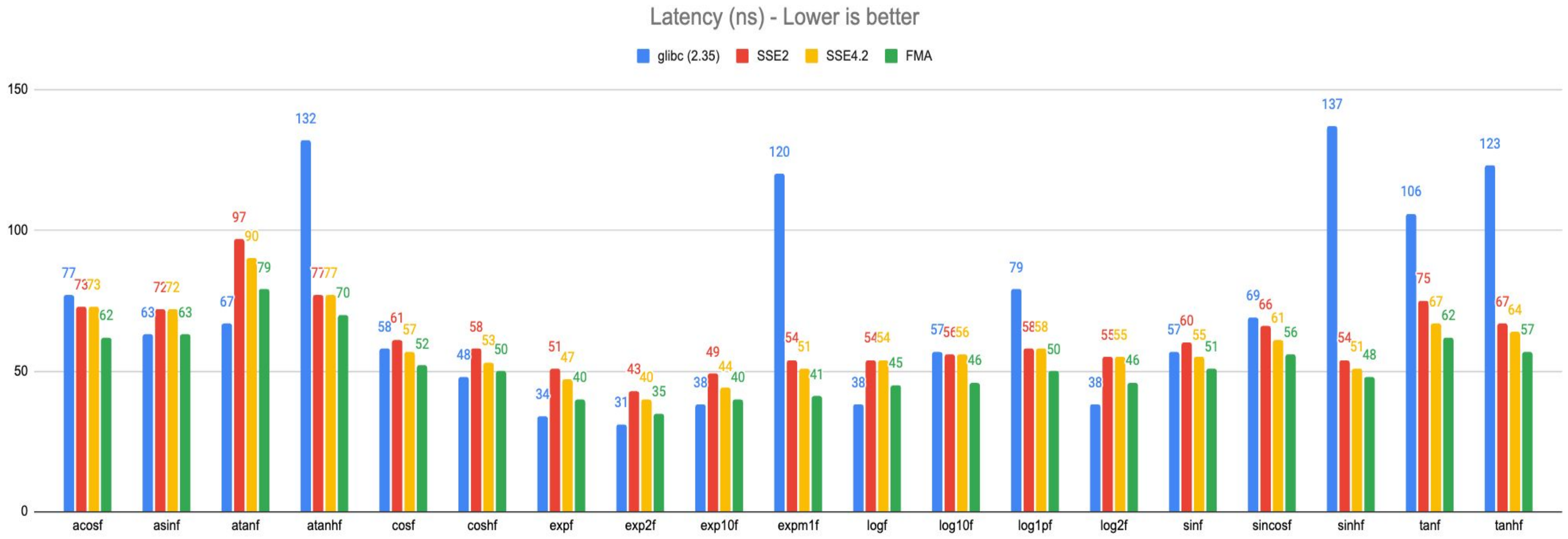
Results - On average (with equal weight for each function):

- On SSE2 (default x86-64): throughput **+18.4%**, latency **+1.3%**
- On SSE4.2 (rounding instructions): throughput **+55.4%**, latency **-3.6%**
- On AVX2 (rounding + FMAs): throughput **+99.5%**, latency **-15.3%**

Performance: Reciprocal Throughput



Performance: Latency

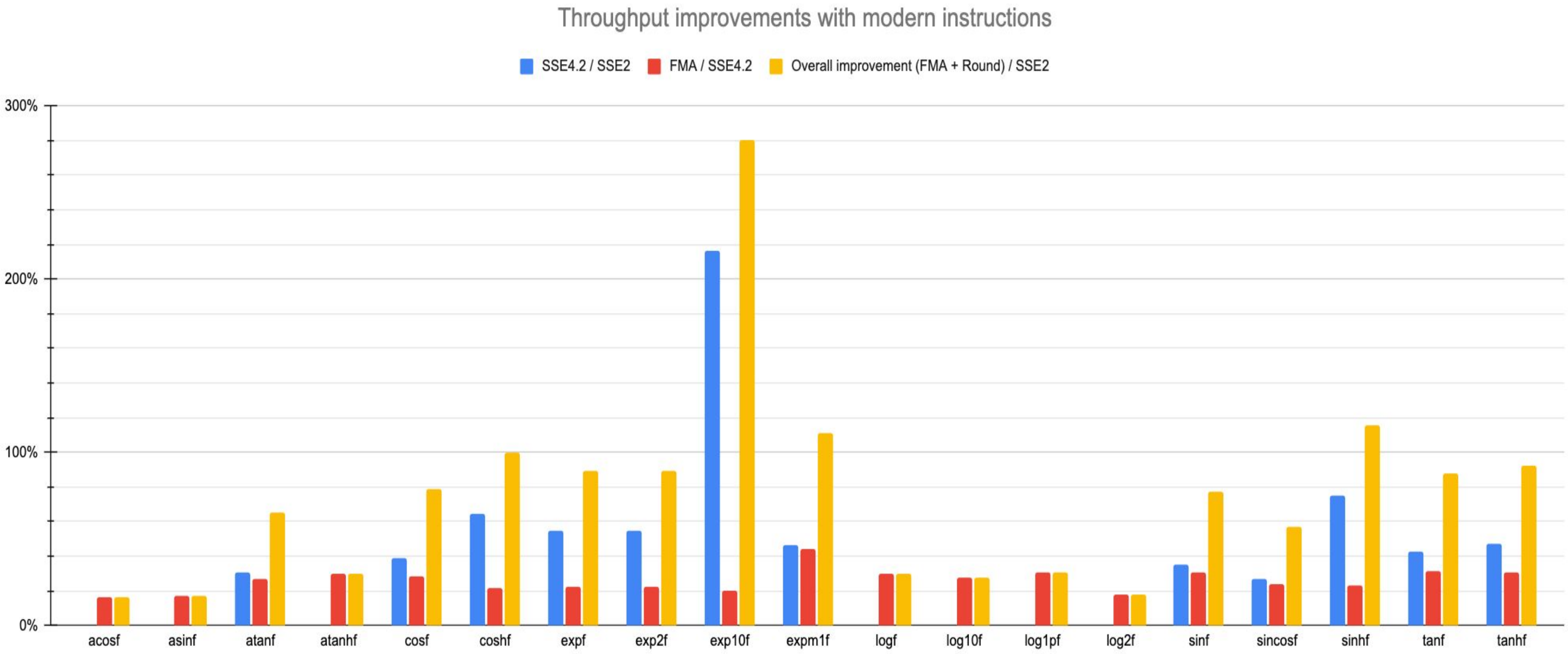


Effects of rounding and FMA instructions

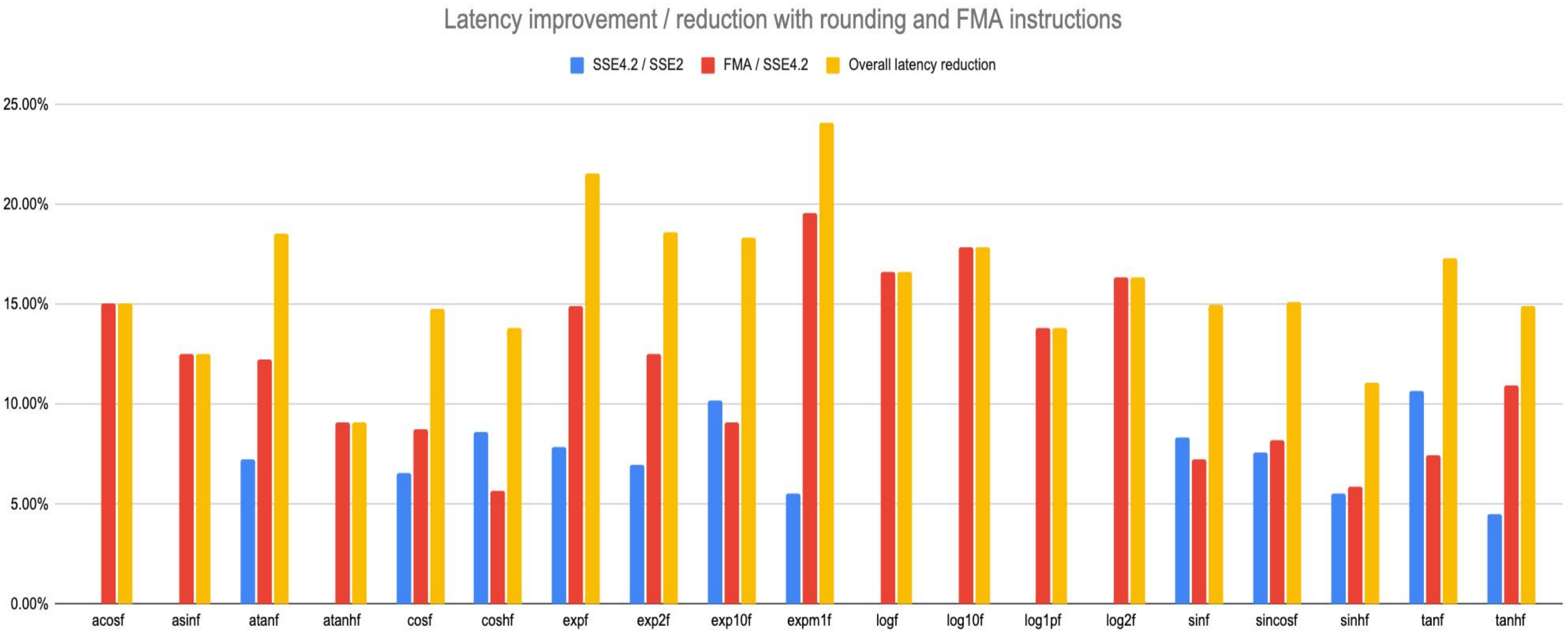
Improvements when applicable, on average:

- Using rounding instructions: throughput **+61%**, latency **-7%**
- Using FMA instructions: throughput **+26%**, latency **-12%**
- Using both: throughput **+74%**, latency **-16%**

Effects of rounding and FMA instructions



Effects of rounding and FMA instructions



Conclusion

- Modern instructions such as floating point rounding or fused-multiply-add significantly increase the throughputs and reduce the latencies of math functions when applicable.
- Comprehensive testings allow us to try various performance optimizations without sacrificing accuracy.
- Beware of dependencies with compiler builtins.
- Assembly might be needed, but it would be better to put them into low dependency compiler builtins.

References

- IBM Accurate Mathematical Library
  - Correctly rounded double precision math library (default rounding mode)
  - https://github.com/dreal-deps/mathlib
- CR-LIBM
  - Correctly rounded double precision math library (all rounding modes, default rounding environment)
  - https://github.com/taschini/crlibm
- RLIBM
  - Correctly rounded single precision math library (all rounding modes)
  - https://people.cs.rutgers.edu/~sn349/rlibm/
- The CORE-MATH project
  - Correctly rounded single, double*, long double* math library (all rounding modes) (*-in development)
  - https://core-math.gitlabpages.inria.fr/

THANKS!