Benchmarking SoftPosit in Eigen: Evaluating Posit16 vs Float in Matrix Arithmetic

The goal of this assignment is to identify a C/C++ library that could benefit from posit number representations over IEEE-754 floating point, implementing posit support using the softposit-cpp library and evaluate performance and numerical accuracy

IEEE-754 floats have well known accuracy issues (rounding, underflow/overflow)

Posits Offer:

- Higher accuracy near 1.0
- Graceful underflow
- Better dynamic range in fewer bits

Softposit is a software implementation of the posit standard therefore it can be slower than hardware floats due to optimizations, but may offer better accuracy or compactness.

Selected Library: Eigen

Eigen is a popular C++ linear algebra library that allows custom types through NumTraits and templating, Posits were implemented using `NumTraits<posit32>` template specialization

Implementation

The `posit32` (32-bit posit) type from SoftPosit-cpp was used to represent a comparison between `float` (32-bit IEEE-754 on Linux). Template specialization for `NumTraits<posit32>` was used to integrate a posit32 into Matrices.

```
// Posit NumTraits specialization for Eigen
namespace Eigen
{
    template<>
    struct NumTraits<posit32> {
        using Self = posit32;
        using Real = posit32;
        using NonInteger = posit32;
        using Nested = posit32;
        using Literal = float;

        enum {
            IsComplex = 0,
            IsInteger = 0,
            IsSigned = 1,
            RequireInitialization = 1,
            ReadCost = 1,
            AddCost = 2,
            MulCost = 2
        };
```

```
        static inline Real epsilon() { return p32(0.00001f); }
        static inline Real dummy_precision() { return p32(0.00001f); }
        static inline int digits10() { return 3; }  // arbitrary safe num
    };
}
```

The benchmark function takes in integers for the rows and columns of the matrix to create, a number of repititions to perform the arithmetic operations and the values for the two matrices to be filled with. Dynamic Eigen matrices are used to fill the matrices with the numbers, the double matrix is there to be used as a baseline for testing the mean absolute error of the matrix multiplications of the float and posit16 matrices

```
// Benchmarking template
template<typename A, typename B>
void benchmark(int r, int c, int repetitions, A&& numa, B&& numb)
{
  Matrix<posit32, Dynamic, Dynamic> pa(r, c);
  Matrix<posit32, Dynamic, Dynamic> pb(r, c);

  Matrix<float, Dynamic, Dynamic> fa(r, c);
  Matrix<float, Dynamic, Dynamic> fb(r, c);

  Matrix<double, Dynamic, Dynamic> da(r, c);
  Matrix<double, Dynamic, Dynamic> db(r, c);
  // fill, run, measure...
}
```

Addition, Subtraction and Multiplication of the two matrices are performed for each of the float and posit16 types, `volatile` is used to prevent the compiler from optimzing out the arithmetic.

```
auto pstart = high_resolution_clock::now();
auto pmul = pa * pb;
volatile auto padd = pa + pb;
volatile auto pmin = pa - pb;
auto pend = high_resolution_clock::now();
pelapsed += pend - pstart;

auto fstart = high_resolution_clock::now();
auto fmul = fa * fb;
volatile auto fadd = fa + fb;
volatile auto fmin = fa - fb;
auto fend = high_resolution_clock::now();
felapsed += fend - fstart;
```

Benchmarking

Matrix Ops Measured: Multiplication, Addition, Subtraction
Measurement: Aggregated time taken to perform addition, subtraction and multiplication of matrices Matrix Sizes: From 10×10 to 50×50 in 10-step increments
Repetitions: 5 per measurement
Values:

- Baseline values `1.0, 2.0`
- Small differences `1.00001, 0.99999` (precision test)
- Very small numbers `1e-5, 2e-5` (underflow test)
- Very large numbers `1e4, 1e4` (overflow test)

Platform: Linux-x86-64
Compiler: g++ with -O3 with the following makefile

```
run: main.cpp
    g++ -std=gnu++20 -o main \
 main.cpp \
 /root/softposit/soft-posit-cpp/build/libsoftposit.a  \
 -I/root/softposit/soft-posit-cpp/include  \
 -I/root/eigen-3.4.0 \
 -O3 && ./main
```

Eigen utilizes SIMD optimizations for arithmetic operations on floats, this checks to see which are enabled. For the benchmark only SSE is enabled.

```
    #ifdef EIGEN_VECTORIZE_SSE
        std::cout << "SSE enabled\n";
    #endif
    #ifdef EIGEN_VECTORIZE_AVX
        std::cout << "AVX enabled\n";
    #endif
    #ifdef EIGEN_VECTORIZE_AVX512
        std::cout << "AVX-512 enabled\n";
    #endif
    #ifdef EIGEN_VECTORIZE_NEON
        std::cout << "NEON enabled (ARM)\n";
    #endif
    #ifndef EIGEN_VECTORIZE
        std::cout << "No SIMD vectorization\n";
    #endif
```

Results

Baseline values `1.0, 2.0`

Matrix Size	Posit Time	Float Time	Posit Mean Absolute Error	Float Mean Absolute Error

Matrix Size	Posit Time	Float Time	Posit Mean Absolute Error	Float Mean Absolute Error
10x10	0.0372 μs	0.0218 μs	0	0
20x20	0.1812 μs	0.1088 μs	0	0
30x30	0.3228 μs	0.1852 μs	0	0
40x40	0.4716 μs	0.2624 μs	0	0
50x50	0.6162 μs	0.3460 μs	0	0

Small differences `1.00001, 0.99999` (precision test)

Matrix Size	Posit Time	Float Time	Posit Mean Absolute Error	Float Mean Absolute Error
10x10	0.0706 μs	0.0412 μs	9.99998×10^{-10}	9.99998×10^{-10}
20x20	0.2162 μs	0.1280 μs	1.99999×10^{-9}	1.99999×10^{-9}
30x30	0.3646 μs	0.2044 μs	2.99998×10^{-9}	2.99998×10^{-9}
40x40	0.5076 μs	0.2834 μs	3.99999×10^{-9}	3.99999×10^{-9}
50x50	0.6506 μs	0.3654 μs	5.0×10^{-9}	5.0×10^{-9}

Very small numbers `1e-5, 2e-5` (underflow test)

Matrix Size	Posit Time	Float Time	Posit Mean Absolute Error	Float Mean Absolute Error
10x10	0.1118 μs	0.0652 μs	1.65481×10^{-16}	5.65639×10^{-17}
20x20	0.2506 μs	0.1470 μs	3.30961×10^{-16}	1.13128×10^{-16}
30x30	0.3972 μs	0.2234 μs	4.96442×10^{-16}	5.23530×10^{-17}
40x40	0.5450 μs	0.3052 μs	6.61923×10^{-16}	2.26255×10^{-16}
50x50	0.6866 μs	0.3846 μs	8.27404×10^{-16}	6.07747×10^{-17}

Very large numbers `1e4, 1e4` (overflow test)

Matrix Size	Posit Time	Float Time	Posit Mean Absolute Error	Float Mean Absolute Error
10x10	0.1434 μs	0.0870 μs	512	0
20x20	0.2868 μs	0.1660 μs	3072	0
30x30	0.4344 μs	0.2430 μs	5632	0
40x40	0.5816 μs	0.3246 μs	8192	0
50x50	0.7214 μs	0.4040 μs	12800	1496.68

Analysis

Performance:

- IEEE Float outperforms Posit32 consistently across all tested matrix sizes and input ranges, typically by a factor of ~1.8–2x. This is largely due to native hardware acceleration for floats (via SIMD), while Posit operations are software-emulated.

Accuracy:

- For typical ranges (e.g., values near 1.0), both Posit and Float achieved perfect or near-perfect accuracy.
- In the small difference test (e.g., 1.00001 vs. 0.99999), both types maintained equal absolute error magnitudes.
- For very small values (e.g., 1e-5), Float showed slightly better precision, maintaining errors in the 1e-17 range vs. Posit in the 1e-16 range.
- For large values (e.g., 1e4), Float was drastically more accurate, with Posit32 producing significant errors (e.g., 12800) compared to Float (1496.68 or even 0 in some cases), suggesting numeric instability or overflow effects.

Conclusion

The benchmarks demonstrate that Posit32 is not yet a viable alternative to IEEE 32-bit floats for general-purpose matrix arithmetic, particularly in environments like Eigen without hardware-level posit support.

While Posit32 maintained competitive accuracy in standard cases, it suffered:

- Severe numerical errors under large magnitude inputs (up to 12800 mean absolute error)
- Slightly worse precision with very small inputs,
- Consistently slower performance across all scenarios (by 50–80%).

These results confirm that IEEE floats remain superior in:

- Performance-critical applications due to hardware acceleration,
- Numerical stability across wide dynamic ranges,
- Precision retention under extreme input scenarios.

Posits may still offer benefits in niche domains such as:

- Custom FPGA/ASIC environments
- Domains with restricted dynamic ranges centered near 1.0,
- Or in cases prioritizing bit-efficiency over performance and accuracy.

But under current conditions and software libraries, IEEE float remains the more practical and performant choice. This is why a HAL library for posits on RISC-V where hardware support for posits is growing is a very crucial step for posits implementation in technologies.