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<posit16>` template specialization

Implementation

The `posit16` (16-bit posit) type from SoftPosit-cpp was used to represent a comparison between `float`. Template specialization for `NumTraits<posit16>` was used to integrate a posit16 into Matrices.

```
// Posit NumTraits specialization for Eigen
namespace Eigen
{
    template<>
    struct NumTraits<posit16> {
        using Self = posit16;
        using Real = posit16;
        using NonInteger = posit16;
        using Nested = posit16;
        using Literal = float;

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

```
        static inline Real epsilon() { return p16(0.00001f); }
        static inline Real dummy_precision() { return p16(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<posit16, Dynamic, Dynamic> pa(r, c);
  Matrix<posit16, 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
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 (µs)	Float Time (µs)	Posit Mean Abs. Error	Float Mean Abs. Error

Matrix Size	Posit Time (µs)	Float Time (µs)	Posit Mean Abs. Error	Float Mean Abs. Error
10x10	0.0416	0.02	0	0
20x20	0.0814	0.0418	0	0
30x30	0.1192	0.0618	0	0
40x40	0.1588	0.0858	0	0
50x50	0.1988	0.106	0	0

Small differences `1.00001, 0.99999` (precision test)

Matrix Size	Posit Time (µs)	Float Time (µs)	Posit Mean Abs. Error	Float Mean Abs. Error
10x10	0.0414	0.0200	9.99998e-10	9.99998e-10
20x20	0.0812	0.0400	1.99999e-09	1.99999e-09
30x30	0.1208	0.0600	2.99998e-09	2.99998e-09
40x40	0.1606	0.0800	3.99999e-09	3.99999e-09
50x50	0.1966	0.1000	5.00000e-09	5.00000e-09

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

Matrix Size	Posit Time (µs)	Float Time (µs)	Posit Mean Abs. Error	Float Mean Abs. Error
10x10	0.0460	0.0198	1.29012e-08	5.65639e-17
20x20	0.0818	0.0396	1.09012e-08	1.13128e-16
30x30	0.1196	0.0596	8.90116e-09	5.23530e-17
40x40	0.1618	0.0796	6.90116e-09	2.26255e-16
50x50	0.2000	0.0996	4.90116e-09	6.07747e-17

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

Matrix Size	Posit Time (µs)	Float Time (µs)	Posit Mean Abs. Error	Float Mean Abs. Error
10x10	0.0434	0.0198	9.32891e+08	0
20x20	0.0824	0.0420	1.93289e+09	0
30x30	0.1214	0.0622	2.93289e+09	0
40x40	0.1640	0.0844	3.93289e+09	0
50x50	0.2018	0.1044	4.93289e+09	1496.68

Analysis

- Performance: Float is consistently faster (about 2x) due to hardware acceleration (SIMD, native float units).
- Accuracy: Equal for typical values and small deltas. Float outperforms posit with both very small and very large numbers in this setup.
- Posit16 limitations: Precision and dynamic range are not sufficient to outperform IEEE 32-bit floats in these scenarios.

Conclusion

The benchmark shows that posit16 is not currently competitive with float (IEEE-754 32-bit) for general-purpose matrix arithmetic in Eigen. While posits offer theoretical advantages such as better accuracy near 1.0 and graceful underflow, these are not realized in practice due to:

- Lack of hardware acceleration (float uses SIMD, posit is software-emulated)
- Limited dynamic range and precision in posit16

In all test cases, float outperformed posit in both speed (~2x faster) and accuracy especially in extreme value scenarios. Float maintained high precision (~1e-16) for very small numbers and resisted accuracy loss under large values, while posit incurred significant error (up to 1e9) due to cumulative rounding. Posits may become more viable in domains with:

- Narrower dynamic range centered near 1.0
- Custom hardware support (e.g., FPGA or ASIC)
- Storage-constrained environments (where fewer bits matter)

However, for high-precision or performance-critical workloads using standard hardware and libraries like Eigen, IEEE floats remain superior under current conditions.