Week3

Hao Mao

Troubles and Solutions

This week presented several challenges; some were resolved while others continue to perplex.

Problem 1: Function Parameterization

I wanted to optimize a testing function to accept a variable number of function names as parameters, with a form resembling:

```
performance_test(test_func1, test_func2, ..., int test_nums = 1);
```

However, integrating an indefinite number of function names as parameters along with a default parameter proved difficult. Initially, I attempted:

```
template<typename... Funcs>
void performance_test(Funcs... funcs) {/* body */}
```

This approach was successful in accepting 1 to n function names as parameters. Yet, I needed to set a default parameter, leading me to adjust its position to the following:

```
template<typename... Funcs>
void performance_test(int test_nums, Funcs... funcs) {
    // run test
}

template<typename... Funcs>
void performance_test(Funcs... funcs) {
    int test_nums = 1;
    performance_test(test_nums, funcs...);
}
```

It functioned well except for the default test_nums being the first parameter, which was not ideal. Soon after, changing test_nums' type to uint caused an infinite recursion error at compile time.

I deduced that the confusion between uint and function names (function pointers) types led to this issue, as the compiler could not distinguish their types, preventing recursion termination.

To differentiate between uint and Funcs, I considered using a helper structure or type wrapper to clearly separate the test_nums parameter from function pointer parameters

```
struct TestNum { int value; };

template<typename... Funcs>
void performance_test(TestNum test_nums, Funcs... funcs) {
    // run test
}

template<typename... Funcs>
void performance_test(Funcs... funcs) {
    performance_test(TestNum{1}, funcs...);
}
```

Then, I recalled SFINAE (Substitution Failure Is Not An Error), which allows the compiler to bypass errors during template instantiation, allowing other potential template instantiations:

```
template<typename T, typename... Funcs>
auto performance_test(T test_nums, Funcs... funcs) -> std::enable_if_t<std::i
    // run test
}

template<typename... Funcs>
void performance_test(Funcs... funcs) {
    uint test_nums = 1;
    performance_test(test_nums, funcs...);
}
```

Here, std::enable_if_t and std::is_integral_v implement SFINAE, allowing template instantiation to depend on whether T is an integral type.

Continued Challenges

On the topic of GEMM optimization, I faced numerous unresolved issues. A notable realization concerned the fundamental matrix multiplication process, typically explained through matrix inner and outer products, giving it a deeper mathematical significance.

Efforts to optimize the algorithms led to significant performance drops when compiler optimizations were set to -Ofast, overshadowing any manual optimizations.

SIMD Exploration

Exploring SIMD on ARM architecture macOS, specifically NEON instead of AVX, yielded suboptimal results compared to compiler optimizations. A simple NEON-optimized algorithm did not outperform the compiler's version:

```
for (size_t k = 0; k < A.num_cols(); ++k) {
    for (size_t i = 0; i < A.num_rows(); ++i) {
        double A_ik = A(i, k);
        float64x2_t A_ik_vec = vdupq_n_f64(A_ik);
        for (size_t j = 0; j < B.num_cols(); j += 2) {
            float64x2_t B_kj_vec = vld1q_f64(&B(k, j));
            float64x2_t C_ij_vec = vld1q_f64(&C(i, j));
            C_ij_vec = vfmaq_f64(C_ij_vec, A_ik_vec, B_kj_vec);
            vst1q_f64(&C(i, j), C_ij_vec);
        }
    }
}
```