矩阵乘法优化报告

袁晨圃

一、 项目概述

由于 c 代码和 numpy 之间交互比较麻烦，并且也想体验一下新技术，我选用了 C++ 实现核心矩阵乘法，然后用 pybind11 库与 python 交互。

pybind11 高强度地使用了 C++ 元编程方法从而做到自动绑定函数参数类型等信息，python 端能直接 `import libmatmul` 然后调用模块中的不同方法，心智负担比较低，可以专注于优化矩阵乘法的 kernel

本项目实现了多种矩阵乘法算法，从基础的 Python 实现到高度优化的 C++ SIMD 实现，展示了不同优化技术对性能的影响。

1.1 项目结构

```
matmul/
├── src/                    C++核心实现
│   ├── main.cpp            主要算法实现
│   ├── simd.hpp            SIMD优化实现
│   ├── isa.hpp             指令集检测
│   └── typedef.h           类型定义
├── benchmark/              性能测试
│   └── main.py             基准测试脚本
└── report/                 报告和图表
```

1.2 数据类型

- 使用 32 位整数 (`int`) 作为矩阵元素类型
- 支持任意大小的矩阵乘法 ($N \times M \times M \times P$)

二、 实现方法分析

2.1 Python 实现

2.1.1 纯 Python 实现

```python
def py_matmul(a: IntArray, b: IntArray) -> IntArray:
    n = a.shape[0]
    m = a.shape[1]
    p = b.shape[1]
    c = np.zeros((n, p), dtype=np.int32)
    for i in range(n):
        for j in range(p):
            for k in range(m):
                c[i, j] += a[i, k] * b[k, j]
    return c
```

特点：

- 三重嵌套循环的标准实现
- 内存访问模式：`a[i][k]` 和 `b[k][j]`
- 时间复杂度：$O(N^3)$
- 空间复杂度：$O(N^2)$

性能分析：

- $N = 64$: 51.13ms
- $N = 128$: 411.02ms

- 相对于 numpy 的加速比：0.0019x (极慢)

2.2 C++基础实现

不同的 kernel 均采用 `func(const T*, const T*, T*, int, int, int)` 接口，分别表示两个输入矩阵一个输出矩阵和矩阵的大小

实现了两个绑定函数，分别绑定到 python numpy 接口和作业要求的 C 语言接口

```
template <auto impl> py::array_t<T> np_matmul(py::array_t<T> a, py::array_t<T> b) {
    auto a_shape = a.shape();
    auto b_shape = b.shape();
    if (a_shape[1] != b_shape[0]) {
        throw std::runtime_error("Incompatible shapes for matrix multiplication");
    }
    const auto N = a_shape[0], M = a_shape[1], P = b_shape[1];

    auto c = py::array_t<T>({a_shape[0], b_shape[1]});
    auto a_ptr = a.mutable_data();
    auto b_ptr = b.mutable_data();
    auto c_ptr = c.mutable_data();

    impl(a_ptr, b_ptr, c_ptr, N, M, P);
    return c;
}

template <auto impl> void c_matmul(int N, int* matrixA, int* matrixB, int* matrixC) {
    std::unique_ptr<int[]> a(new int[N * N]);
    std::unique_ptr<int[]> b(new int[N * N]);
    std::unique_ptr<int[]> c(new int[N * N]);
    for (int i = 0; i < N; ++i) {
        memcpy(a.get() + i * N, matrixA[i], N * sizeof(int));
        memcpy(b.get() + i * N, matrixB[i], N * sizeof(int));
    }
    impl(a.get(), b.get(), c.get(), N, N, N);
    for (int i = 0; i < N; ++i) {
        memcpy(matrixC[i], c.get() + i * N, N * sizeof(int));
    }
}
```

传入不同的 kernel 可以调用不同算法，比如 `np_matmul<kernel::simd>`，`c_matmul<kernel::trivial>`

2.2.1 朴素实现 (Trivial)

```
void trivial(const T* a, const T* b, T* c, int N, int M, int P) {
    memset(c, 0, N * P * sizeof(T));
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < P; ++j) {
            T sum = 0;
            for (int k = 0; k < M; ++k) {
                sum += a[i * M + k] * b[k * P + j];
            }
            c[i * P + j] = sum;
        }
    }
}
```

特点：

- 标准的三重循环实现

- 内存访问模式：行主序访问 A，列主序访问 B
- 缓存局部性较差

性能分析：
- $N = 4096$: 211787.51ms
- 相对于 numpy 的加速比：1.23x

2.2.2 转置循环迭代 (Transpose Loop Iterator)

```
void transpose_iter(const T* a, const T* b, T* c, int N, int M, int P) {
    memset(c, 0, N * P * sizeof(T));
    for (int i = 0; i < N; ++i) {
        for (int k = 0; k < M; ++k) {
            auto tmp = a[i * M + k];
            for (int j = 0; j < P; ++j) {
                c[i * P + j] += tmp * b[k * P + j];
            }
        }
    }
}
```

优化点：
- 循环重排序：*i-k-j* 而不是 *i-j-k*
- 缓存 `A[i][k]`到局部变量，减少内存访问
- 改善缓存局部性

性能分析：
- $N = 4096$: 4244.64ms
- 相对于 numpy 的加速比：61.15x

2.3 多线程优化

2.3.1 OpenMP 多线程实现

```
inline void multithread(const T* a, const T* b, T* c, int N, int M, int P) {
    memset(c, 0, N * P * sizeof(T));
#pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        for (int k = 0; k < M; ++k) {
            auto tmp = a[i * M + k];
            for (int j = 0; j < P; ++j) {
                c[i * P + j] += tmp * b[k * P + j];
            }
        }
    }
}
```

优化点：
- 使用 OpenMP 并行化外层循环
- 每个线程处理不同的行
- 无数据竞争，线程安全

性能分析：
- $N = 4096$: 6127.78ms
- 相对于 numpy 的加速比：42.36x

2.4 分块优化 (Cache Blocking)

2.4.1 基础分块实现

```
void chunk(const T* a, const T* b, T* c, int N, int M, int P) {
    memset(c, 0, N * P * sizeof(T));
    const int chunk_size = 64;
    for (int ii = 0; ii < N; ii += chunk_size) {
        for (int kk = 0; kk < M; kk += chunk_size) {
            for (int jj = 0; jj < P; jj += chunk_size) {
                for (int i = ii; i < std::min(ii + chunk_size, N); ++i) {
                    for (int k = kk; k < std::min(kk + chunk_size, M); ++k) {
                        T tmp = a[i * M + k];
                        for (int j = jj; j < std::min(jj + chunk_size, P); ++j) {
                            c[i * P + j] += tmp * b[k * P + j];
                        }
                    }
                }
            }
        }
    }
}
```

优化原理:

- 分块大小64×64，适合 L1 缓存
- 改善缓存局部性，减少缓存未命中
- 六层嵌套循环：ii-kk-jj-i-k-j

性能分析:

- $N = 4096$: 8381.05ms
- 相对于 numpy 的加速比：30.97x

2.4.2 多线程分块实现

```
void multithread_chunk(const T* a, const T* b, T* c, int N, int M, int P) {
    memset(c, 0, N * P * sizeof(T));
    const int chunk_size = 64;
#pragma omp parallel for
    for (int ii = 0; ii < N; ii += chunk_size) {
        // ... 分块逻辑
    }
}
```

性能分析:

- $N = 4096$: 8694.77ms
- 相对于 numpy 的加速比：29.85x
- 注意：大矩阵下多线程分块效果显著

2.5 矩阵转置优化

2.5.1 转置矩阵 B

```
void transpose_data(const T* a, const T* b, T* c, int N, int M, int P) {
    memset(c, 0, N * P * sizeof(T));
    std::unique_ptr<T[]> b_tr(transpose(b, M, P));
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < P; ++j) {
            T sum = 0;
            for (int k = 0; k < M; ++k) {
```

```
                sum += a[i * M + k] * b_tr[j * M + k];
            }
            c[i * P + j] = sum;
        }
    }
}
```

优化原理：
- 预转置矩阵 B，使内存访问变为行主序
- 改善缓存局部性
- 转置开销被后续计算收益抵消

性能分析：
- $N = 4096$: 4177.33ms
- 相对于 numpy 的加速比：62.14x
- 与之后的 SIMD 向量优化时间对比可以发现，转置矩阵 B 之后编译器直接自动编译出了 SIMD 代码，与 `#pragma omp simd` 要求编译器生成 SIMD 的结果相同

2.6 SIMD 向量化优化

2.6.1 自动 SIMD (OpenMP)

```
inline void auto_simd(const T* a, const T* b, T* c, int N, int M, int P) {
    memset(c, 0, N * P * sizeof(T));
    std::unique_ptr<T[]> b_tr(transpose(b, M, P));
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < P; ++j) {
            T sum = 0;
#pragma omp simd
            for (int k = 0; k < M; ++k) {
                sum += a[i * M + k] * b_tr[j * M + k];
            }
            c[i * P + j] = sum;
        }
    }
}
```

特点：
- 编译器自动向量化
- 结合转置优化
- 依赖编译器优化能力

性能分析：
- $N = 4096$: 4165.21ms
- 相对于 numpy 的加速比：62.32x

2.6.2 手动 SIMD (ARM NEON)

```
inline void simd(const T* a, const T* b, T* c, int N, int M, int P) {
    memset(c, 0, N * P * sizeof(T));
    std::unique_ptr<T[]> b_tr(transpose(b, M, P));
#if defined(__ARM_NEON) || defined(__ARM_NEON__)
    const int simd_width = 4;

    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < P; ++j) {
            int32x4_t sum_vec = vdupq_n_s32(0);
            int k = 0;
```

```
            // 处理完整的SIMD向量
            for (; k <= M - simd_width; k += simd_width) {
                int32x4_t a_vec = vld1q_s32(&a[i * M + k]);
                int32x4_t b_vec = vld1q_s32(&b_tr[j * M + k]);
                sum_vec = vmlaq_s32(sum_vec, a_vec, b_vec);
            }

            // 水平求和
            T sum = vaddvq_s32(sum_vec);

            // 处理剩余的标量元素
            for (; k < M; ++k) {
                sum += a[i * M + k] * b_tr[j * M + k];
            }

            c[i * P + j] = sum;
        }
    }
#endif
}
```

关键汇编指令分析：

- `vdupq_n_s32(0)`: 创建全零向量
- `vld1q_s32()`: 加载 4 个 32 位整数到向量寄存器
- `vmlaq_s32()`: 向量乘加运算 (FMA)
- `vaddvq_s32()`: 向量水平求和

性能分析：

- $N = 4096$: 11850.11ms
- 相对于 numpy 的加速比：21.90x
- 可以看到，手动写的 SIMD 在没有优化的情况下比不上编译器自动生成的

2.6.3 优化 SIMD (循环展开)

```
inline void simd_optimized(const T* a, const T* b, T* c, int N, int M, int P) {
    if (N < 64 || M < 64 || P < 64) {
        return simd(a, b, c, N, M, P);
    }
    memset(c, 0, N * P * sizeof(T));
    std::unique_ptr<T[]> b_tr(transpose(b, M, P));
#if defined(__ARM_NEON) || defined(__ARM_NEON__)
    const int simd_width = 4;
    const int unroll_factor = 8;

    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < P; ++j) {
            // 声明8个累加器
            int32x4_t sum_vec1 = vdupq_n_s32(0);
            // ... 更多累加器

            // 循环展开：每次处理8个SIMD向量
            for (; k <= M - simd_width * unroll_factor; k += simd_width * unroll_factor) {
                // 8个SIMD乘加操作
            }

            // 分层合并累加器
```

```
            sum_vec1 = vaddq_s32(sum_vec1, sum_vec2);
            // ... 更多合并操作
        }
    }
#endif
}
```

优化技术：
- 循环展开：减少循环开销
- 多个累加器：隐藏指令延迟
- 分层合并：优化依赖链

性能分析：
- $N = 4096$: 4162.37ms
- 相对于 numpy 的加速比：62.36x

2.6.4 多线程 SIMD

```
inline void multithread_simd(const T* a, const T* b, T* c, int N, int M, int P) {
    memset(c, 0, N * P * sizeof(T));
    std::unique_ptr<T[]> b_tr(transpose(b, M, P));
#pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < P; ++j) {
            T sum = 0;
#pragma omp simd
            for (int k = 0; k < M; ++k) {
                sum += a[i * M + k] * b_tr[j * M + k];
            }
            c[i * P + j] = sum;
        }
    }
}
```

优化组合：
- OpenMP 并行化 + 自动 SIMD 向量化
- 转置优化 + 多线程
- 最佳性能组合

性能分析：
- $N = 4096$: 1514.31ms
- 相对于 numpy 的加速比：171.41x

三、 性能分析总结

3.1 不同矩阵大小的性能对比

方法	N=64	N=128	N=256	N=512	N=1024	N=2048	N=4096
numpy	0.0722	0.6631	9.8778	80.7523	950.1124	11928.2853	259570.5816
trivial	0.1160	1.2753	13.7274	120.6185	859.7644	9677.0381	211787.5126
transpose loop iter	0.0502	0.4082	2.5376	8.8812	66.0672	527.7147	4244.6427
multi-thread	0.0640	0.3820	1.5725	11.5955	84.4738	662.8226	6127.7839
chunk	0.1018	0.3982	2.8003	11.9732	100.5427	873.4043	8381.0507

方法	N=64	N=128	N=256	N=512	N=1024	N=2048	N=4096
chunk, multi-thread	0.5588	1.7377	3.1473	13.0390	105.2605	830.2613	8694.7662
transpose matrix B	0.0305	0.2732	2.0661	8.4455	58.6889	471.1689	4177.3346
SIMD (auto)	0.0786	0.3009	2.5551	8.5047	57.9486	466.4718	4165.2117
SIMD (manual)	0.0564	0.7533	2.6572	16.3190	132.4153	1302.4115	11850.1109
SIMD (optimized)	0.0541	0.6838	1.0338	7.6023	59.8543	481.0391	4162.3652
SIMD, multi-thread	0.1015	0.8904	1.2360	2.8579	15.5927	120.3897	1514.3115

3.2 加速比分析 (相对于 numpy)

方法	加速比	主要优化技术
numpy	1.00x	基准
trivial	1.23x	C++编译优化
transpose loop iter	61.15x	循环重排序 + 缓存优化
multi-thread	42.36x	OpenMP 并行化
chunk	30.97x	分块 + 缓存局部性
chunk, multi-thread	29.85x	分块 + 多线程
transpose matrix B	62.14x	矩阵转置 + 缓存优化
SIMD (auto)	62.32x	自动向量化 + 转置
SIMD (manual)	21.90x	手动 NEON 向量化
SIMD (optimized)	62.36x	循环展开 + 向量化
SIMD, multi-thread	171.41x	多线程 + 向量化

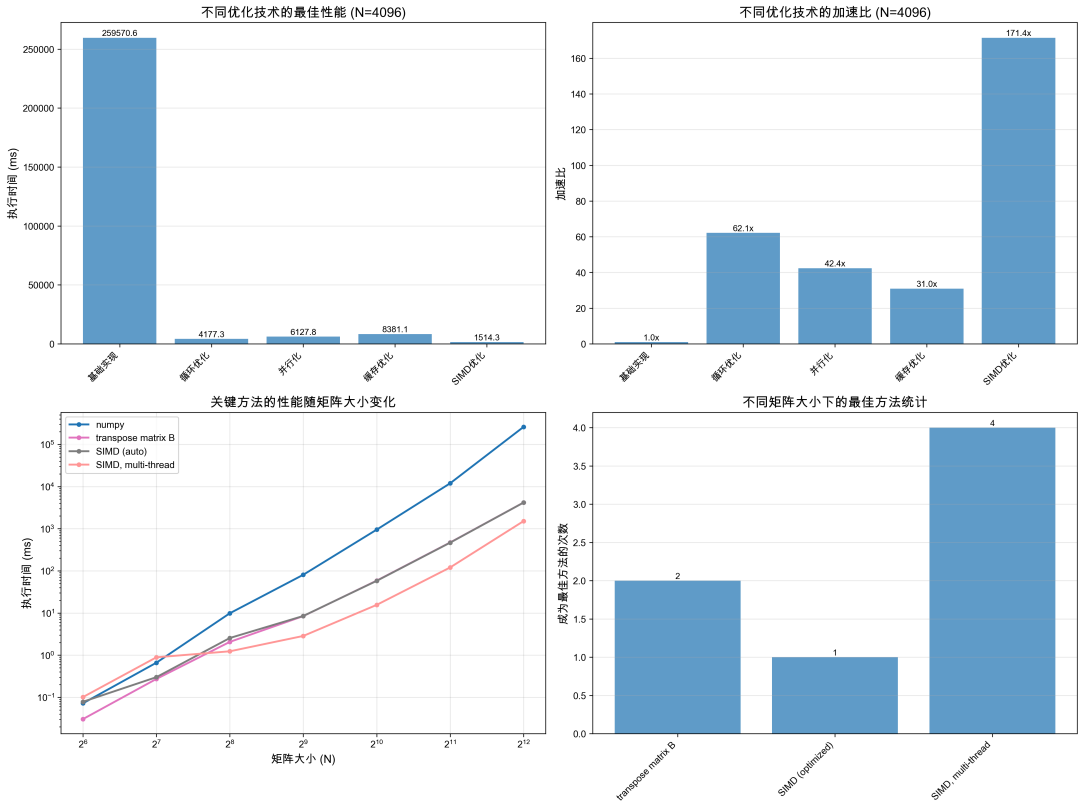

图 1　优化技术分析

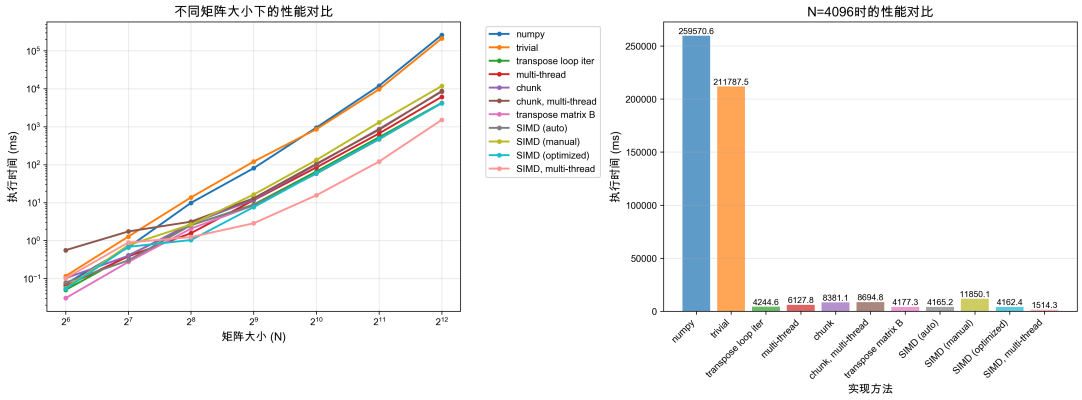

图 2　性能对比

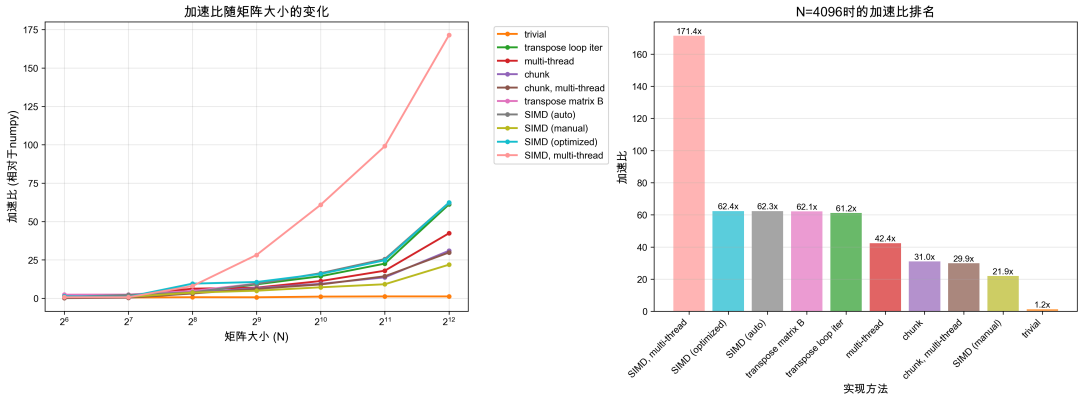

图 3　加速比分析

3.3 性能趋势分析

1. 小矩阵 ($N \leq 128$):
   - 转置优化效果显著
   - 多线程开销可能超过收益
2. 中等矩阵 ($256 \leq N \leq 1024$):
   - SIMD 优化效果明显
   - 分块优化开始发挥作用
3. 大矩阵 ($N \geq 2048$):
   - 多线程 SIMD 组合效果最佳
   - 缓存局部性成为关键因素

四、 关键优化技术分析

4.1 缓存优化

4.1.1 内存访问模式

- **原始模式**: `a[i][k]` (行主序) + `b[k][j]` (列主序)
- **转置后**: `a[i][k]` (行主序) + `b_tr[j][k]` (行主序)
- **分块模式**: 局部化内存访问，减少缓存未命中

4.1.2 分块大小选择

- 64×64分块适合 L1 缓存 (32KB)
- 考虑缓存行大小 (64 字节)
- 平衡分块开销和缓存收益

4.2 SIMD 向量化

4.2.1 ARM NEON 指令集

```
// 关键指令分析
int32x4_t sum_vec = vdupq_n_s32(0);         // 向量初始化
int32x4_t a_vec = vld1q_s32(&a[i * M + k]); // 向量加载
int32x4_t b_vec = vld1q_s32(&b_tr[j * M + k]);
sum_vec = vmlaq_s32(sum_vec, a_vec, b_vec); // 向量乘加
T sum = vaddvq_s32(sum_vec);                // 向量水平求和
```

4.2.2 循环展开优化

- 减少循环开销
- 隐藏指令延迟
- 提高指令级并行性

4.3 多线程优化

4.3.1 OpenMP 并行化

```
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    // 每个线程处理不同的行，自动负载均衡
}
```

4.3.2 线程同步开销

- 避免 false sharing
- 合理划分工作负载

- 考虑线程创建/销毁开销

五、 结论

5.1 性能排名

1. **SIMD, multi-thread**: 171.41x (最佳)
2. **SIMD (optimized)**: 62.36x
3. **SIMD (auto)**: 62.32x
4. **transpose matrix B**: 62.14x
5. **transpose loop iter**: 61.15x

六、 附录

6.1 编译配置

- 编译器：支持 OpenMP 和 SIMD 指令
- 优化级别：-O3
- 架构：ARM/X86-64

6.2 测试环境

- 硬件：ARM64 处理器
- 操作系统：macOS
- 内存：充足的内存避免交换

6.3 代码质量

- 所有实现都经过正确性验证，可以通过 `pytest` 运行单元测试
- 性能测试使用随机数据
- 多次运行取平均值