



南開大學
Nankai University

计算机学院
计算机组成原理设计报告

矩阵乘法

姓名：朱子昂
学号：2311283
专业：计算机科学与技术

2025 年 6 月 3 日

目录

1	github 仓库地址	2
2	问题重述	2
2.1	基础题：智能矩阵乘法优化挑战	2
2.1.1	基本优化思路	2
3	串行算法测试	3
3.1	不进行编译优化串行算法	3
3.2	o1 编译优化	3
3.3	o2 编译优化	3
4	并行化优化	3
4.1	openMP	3
4.2	子块并行化	4
4.3	openMP+ 循环展开优化	4
4.4	时间计算	5
5	性能对比	8
6	附运行结果	10
6.1	编译	10
6.2	openMP 运行	10
6.3	分子块运行	10
6.4	循环展开 +openMP 运行	11

1 github 仓库地址

计算机组成原理大作业

2 问题重述

2.1 基础题：智能矩阵乘法优化挑战

低轨 (LEO) 卫星网络因其低时延、高覆盖的优势，正成为未来全球广域网络服务的重要补充。目前，SpaceX、OneWeb 等公司已部署数千颗卫星，初步形成星座网络；我国星网工程也在加快推进，积极构建天地一体化信息网络。LEO 卫星网络具备动态拓扑、链路多变、频繁切换等特点，使其网络服务面临带宽波动性大、链路预测难等挑战。因此，提升服务质量的关键之一在于精准的网络带宽预测。借助机器学习模型，可实现对历史网络状态的深度建模与未来网络带宽的有效预测，但如何实现高效且实时的预测，要求对机器学习的计算过程进行深度优化。机器计算学习过程的核心计算单元是矩阵乘法运算。在实际应用中，如何高效利用加速硬件（如曙光 DCU, 英伟达 GPU 等）和并行计算算法完成大规模矩阵乘，成为智能计算系统设计的关键问题。为应对高效准确 LEO 卫星带宽预测挑战，本次实训将围绕基于矩阵乘法的多层感知机 (MLP) 神经网络计算优化展开，通过设计一系列挑战任务，培训并引导参赛者从算法理解、性能建模、系统优化到异构调度完成一个完整的系统创新设计。

编程语言：C++，曙光 DCU ToolKit (DTK)；环境：曙光 DCU 实训平台；算力：8 核 CPU+1 张 DCU 加速卡 +16G 内存

已知两个矩阵：矩阵 A (大小 $N \times M$)，矩阵 B (大小 $M \times P$)：

问题一：请完成标准的矩阵乘算法，并支持浮点型输入，输出矩阵为 $C = A \times B$ ，并对随机生成的双精度浮点数矩阵输入，验证输出是否正确 ($N=1024$, $M=2048$, $P=512$, N 、 M 和 P 也可为任意的大数)；

问题二：请采用至少一种方法加速以上矩阵运算算法，鼓励采用多种优化方法和混合优化方法；理论分析优化算法的性能提升，并可通过 rocm-smi、hipprof、hipgdb 等工具进行性能分析和检测，以及通过柱状图、折线图等图形化方式展示性能对比；

2.1.1 基本优化思路

1.1 多线程并行化加速

通过多线程并行化加速计算过程，可充分利用多核 CPU 的计算资源，可采用 OpenMP (Open Multi-Processing) 实现矩阵乘法计算优化。

1.2 子块并行优化

子块并行 (Block-wise Parallelization) 是矩阵乘法中的一种优化技术，可通过局部计算降低内存访问延迟；为 OpenMP 或其他并行机制提供更细粒度、更均匀的工作划分，适用于大规模矩阵，特别适合在多核 CPU 上运行。

1.3 多进程并行优化

使用 MPI (Message Passing Interface) 实现矩阵乘法的多进程优化，其核心思想是将大矩阵按行或块划分给不同进程，利用进程间通信协同完成整个计算。适用于分布式系统或多节点多核并行平台，能突破单机内存和计算瓶颈。

1.4 DCU 加速计算

通过国产高性能异构加速器、曙光 DCU (Dawn Computing Unit), 加速 AI 训练、推理和高性能计算场景。DCU 与 NVIDIA GPU 特性类似, 支持大规模并行计算, 但通常通过 HIP C++ 编程接口进行开发, 兼容 CUDA 语义。

注: HIP (Heterogeneous-Compute Interface for Portability) 是 AMD 公司在 2016 年提出的符合 CUDA 编程模型的、自由的、开源的 C++ 编程接口和核函数语言。

1.5 其他计算优化方法或混合优化

除了以上并行机制, 还有多种计算优化方法和混合优化策略, 可进一步提升矩阵计算效率。如内存访问优化, 混合并行优化等。

3 串行算法测试

3.1 不进行编译优化串行算法

```
root@worker-0:/public/home/xdzs2025_zza# ./o baseline
[Baseline] Time: 18406 ms
```

3.2 o1 编译优化

```
root@worker-0:/public/home/xdzs2025_zza# ./o1 baseline
[Baseline] Time: 27777 ms
```

3.3 o2 编译优化

```
root@worker-0:/public/home/xdzs2025_zza# ./o2 baseline
[Baseline] Time: 29649 ms
```

4 并行化优化

4.1 openMP

```
1 void matmul_openmp(const vector<double>& A, const vector<double>& B, vector<double>&
   C, int N, int M, int P)
2 {
3     #pragma omp parallel for collapse(2) schedule(dynamic)
4     for (int i = 0; i < N; ++i)
5         for (int j = 0; j < P; ++j) {
6             double sum = 0;
```

```

7         for (int k = 0; k < M; ++k)
8             sum += A[i * M + k] * B[k * P + j];
9         C[i * P + j] = sum;
10    }
11 }

```

我选择选择外层 i 和 j 循环并行 (collapse(2)), 因为: 每个 (i,j) 的计算是独立的, 无需同步。并行粒度较大 ($N \times P$ 个任务), 适合多线程负载均衡。

并且使用了动态分配任务, 因为计算矩阵的时候经常会出现不均衡的现象, 所以选择了这种思路。

4.2 子块并行化

```

1 void matmul_block_tiling(const vector<double>& A, const vector<double>&
  B, vector<double>& C, int N, int M, int P, int block_size = 64)
2 {
3     #pragma omp parallel for collapse(2) schedule(dynamic)
4     for (int ii = 0; ii < N; ii += block_size)
5         for (int jj = 0; jj < P; jj += block_size)
6             for (int kk = 0; kk < M; kk += block_size)
7                 for (int i = ii; i < min(ii + block_size, N); ++i)
8                     for (int j = jj; j < min(jj + block_size, P); ++j) {
9                         double sum = 0;
10                        for (int k = kk; k < min(kk + block_size, M); ++k)
11                            sum += A[i * M + k] * B[k * P + j];
12                        C[i * P + j] += sum;
13                    }
14 }

```

我将矩阵分为 64×64 的小块, 然后进行小块计算, 也就是分治算法的思想, 把大矩阵分成小矩阵乘法后相加。通过这样能够更好的发挥 openMP 并行化的能力, 实现更加细化的并行。

4.3 openMP+ 循环展开优化

```

1 void matmul_openmp_unroll(const vector<double>& A,
2                             const vector<double>& B,
3                             vector<double>& C, int N, int M, int P) {
4     #pragma omp parallel for schedule(dynamic)
5     for (int i = 0; i < N; ++i) {
6         for (int k = 0; k < M; ++k) {
7             double a = A[i * M + k];
8             // 展开4次循环
9             int j = 0;
10            for (; j + 3 < P; j += 4) {
11                C[i * P + j] += a * B[k * P + j];
12                C[i * P + j + 1] += a * B[k * P + j + 1];
13                C[i * P + j + 2] += a * B[k * P + j + 2];
14                C[i * P + j + 3] += a * B[k * P + j + 3];
15            }

```

```

16         // 处理剩余部分
17         for (; j < P; ++j) {
18             C[i * P + j] += a * B[k * P + j];
19         }
20     }
21 }
22 }

```

我通过把一个循环中的内容一次展开四次，依旧是借助 openMP 进行并行化同时四次乘法的计算，这样减少了内存的反复取出，增加指令级并行。

4.4 时间计算

使用了 C++11 的 `<chrono>` 库进行时间的计算

```

1 // 计算参考结果(baseline)的时间
2 auto start_ref = high_resolution_clock::now();
3 matmul_baseline(A, B, C_ref, N, M, P);
4 auto end_ref = high_resolution_clock::now();
5 auto duration_ref = duration_cast<milliseconds>(end_ref - start_ref);
6
7 if (mode == "baseline") {
8     cout << "[Baseline] Time: " << duration_ref.count() << " ms\n";
9 } else {
10     auto start = high_resolution_clock::now();
11
12     if (mode == "openmp") {
13         matmul_openmp(A, B, C, N, M, P);
14     } else if (mode == "block") {
15         matmul_block_tiling(A, B, C, N, M, P);
16     } else if (mode == "unroll") { // 新增循环展开模式
17         matmul_openmp_unroll(A, B, C, N, M, P);
18     } else {
19         cerr << "Usage: ./program [baseline|openmp|block|unroll]" << endl;
20         return 1;
21     }
22
23     auto end = high_resolution_clock::now();
24     auto duration = duration_cast<milliseconds>(end - start);
25
26     cout << "[" << mode << "]" Time: " << duration.count() << " ms" << endl;
27     cout << "[" << mode << "]" Speedup: " << (double)duration_ref.count() /
28         duration.count() << endl;
29     cout << "[" << mode << "]" Valid: " << validate(C, C_ref, N, P) << endl;
30 }

```

计算不同情况下的时间和基准时间，对比计算加速比，并且验证正确性。

下面是完整代码

```

1 #include <iostream>

```

```

2 #include <vector>
3 #include <random>
4 #include <cmath>
5 #include <algorithm>
6 #include <omp.h>
7 #include <chrono>
8
9 using namespace std;
10 using namespace std::chrono;
11
12 // 初始化矩阵
13 void init_matrix(vector<double>& mat, int rows, int cols) {
14     mt19937 gen(42);
15     uniform_real_distribution<double> dist(-100.0, 100.0);
16     for (int i = 0; i < rows * cols; ++i)
17         mat[i] = dist(gen);
18 }
19
20 // 验证结果
21 bool validate(const vector<double>& A, const vector<double>& B, int rows, int cols,
22             double tol = 1e-6) {
23     for (int i = 0; i < rows * cols; ++i)
24         if (abs(A[i] - B[i]) > tol) return false;
25     return true;
26 }
27
28 // 基础版本
29 void matmul_baseline(const vector<double>& A,
30                     const vector<double>& B,
31                     vector<double>& C, int N, int M, int P) {
32     for (int i = 0; i < N; ++i)
33         for (int j = 0; j < P; ++j) {
34             C[i * P + j] = 0;
35             for (int k = 0; k < M; ++k)
36                 C[i * P + j] += A[i * M + k] * B[k * P + j];
37         }
38 }
39
40 // OpenMP并行版本
41 void matmul_openmp(const vector<double>& A,
42                   const vector<double>& B,
43                   vector<double>& C, int N, int M, int P) {
44     #pragma omp parallel for collapse(2) schedule(dynamic)
45     for (int i = 0; i < N; ++i)
46         for (int j = 0; j < P; ++j) {
47             double sum = 0;
48             for (int k = 0; k < M; ++k)
49                 sum += A[i * M + k] * B[k * P + j];
50             C[i * P + j] = sum;
51         }
52 }

```

```

50     }
51 }
52
53 // 子块并行优化
54 void matmul_block_tiling(const vector<double>& A,
55                          const vector<double>& B,
56                          vector<double>& C, int N, int M, int P, int block_size = 64)
57 {
58     #pragma omp parallel for collapse(2) schedule(dynamic)
59     for (int ii = 0; ii < N; ii += block_size)
60         for (int jj = 0; jj < P; jj += block_size)
61             for (int kk = 0; kk < M; kk += block_size)
62                 for (int i = ii; i < min(ii + block_size, N); ++i)
63                     for (int j = jj; j < min(jj + block_size, P); ++j) {
64                         double sum = 0;
65                         for (int k = kk; k < min(kk + block_size, M); ++k)
66                             sum += A[i * M + k] * B[k * P + j];
67                         C[i * P + j] += sum;
68                     }
69 }
70
71 // OpenMP+循环展开优化
72 void matmul_openmp_unroll(const vector<double>& A,
73                           const vector<double>& B,
74                           vector<double>& C, int N, int M, int P) {
75     #pragma omp parallel for schedule(dynamic)
76     for (int i = 0; i < N; ++i) {
77         for (int k = 0; k < M; ++k) {
78             double a = A[i * M + k];
79             // 展开4次循环
80             int j = 0;
81             for (; j + 3 < P; j += 4) {
82                 C[i * P + j] += a * B[k * P + j];
83                 C[i * P + j + 1] += a * B[k * P + j + 1];
84                 C[i * P + j + 2] += a * B[k * P + j + 2];
85                 C[i * P + j + 3] += a * B[k * P + j + 3];
86             }
87             // 处理剩余部分
88             for (; j < P; ++j) {
89                 C[i * P + j] += a * B[k * P + j];
90             }
91         }
92     }
93 }
94
95 int main(int argc, char** argv) {
96     const int N = 1024, M = 2048, P = 512;
97     string mode = argc >= 2 ? argv[1] : "baseline";

```



```

98     vector<double> A(N * M);
99     vector<double> B(M * P);
100    vector<double> C(N * P, 0);
101    vector<double> C_ref(N * P, 0);
102
103    init_matrix(A, N, M);
104    init_matrix(B, M, P);
105
106    // 计算参考结果
107    auto start_ref = high_resolution_clock::now();
108    matmul_baseline(A, B, C_ref, N, M, P);
109    auto end_ref = high_resolution_clock::now();
110    auto duration_ref = duration_cast<milliseconds>(end_ref - start_ref);
111
112    if (mode == "baseline") {
113        cout << "[Baseline] Time: " << duration_ref.count() << " ms\n";
114    } else {
115        auto start = high_resolution_clock::now();
116
117        if (mode == "openmp") {
118            matmul_openmp(A, B, C, N, M, P);
119        } else if (mode == "block") {
120            matmul_block_tiling(A, B, C, N, M, P);
121        } else if (mode == "unroll") { // 新增循环展开模式
122            matmul_openmp_unroll(A, B, C, N, M, P);
123        } else {
124            cerr << "Usage: ./program [baseline|openmp|block|unroll]" << endl;
125            return 1;
126        }
127
128        auto end = high_resolution_clock::now();
129        auto duration = duration_cast<milliseconds>(end - start);
130
131        cout << "[" << mode << "] Time: " << duration.count() << " ms" << endl;
132        cout << "[" << mode << "] Speedup: " << (double)duration_ref.count() /
133            duration.count() << endl;
134        cout << "[" << mode << "] Valid: " << validate(C, C_ref, N, P) << endl;
135    }
136
137    return 0;

```

5 性能对比

1. O0 优化等级下的性能表现分析在 O0（无优化）等级下，三种优化方法的加速比相对较低且差异不大。纯 OpenMP 版本获得 11.4 倍加速，分块优化反而只有 8.8 倍，OpenMP+ 循环展开为 10.0 倍。这是因为：

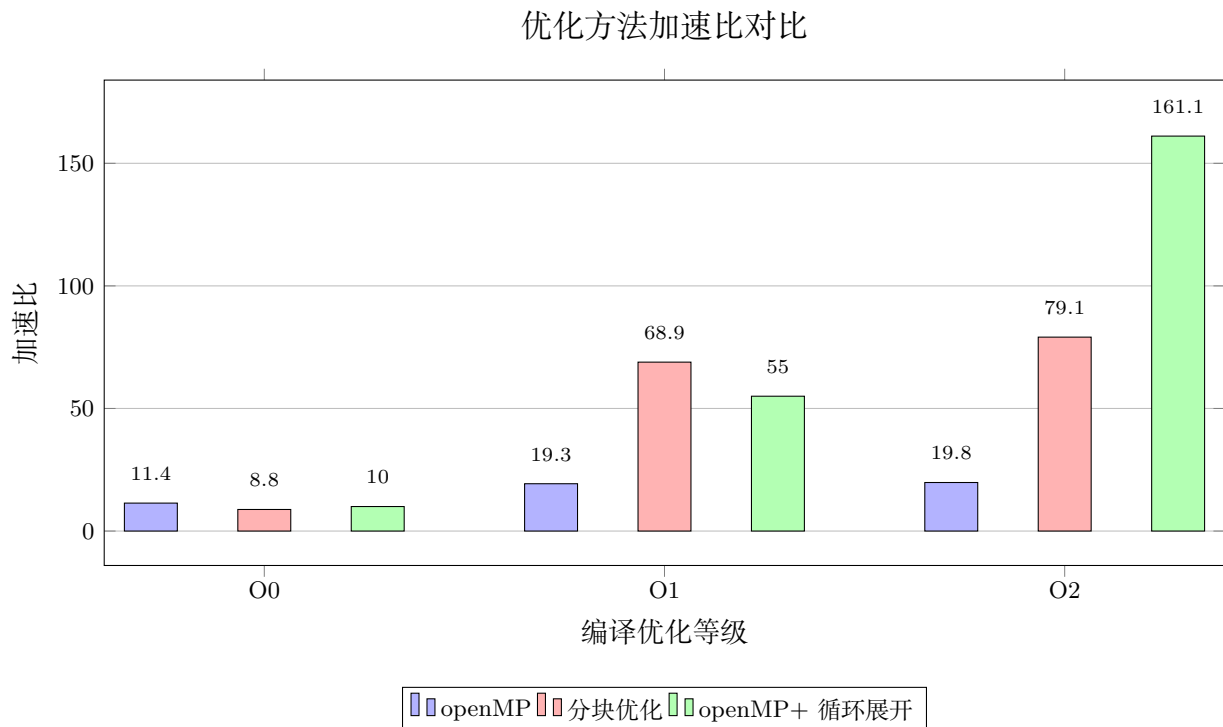


图 5.1: 三种优化方法在不同编译优化等级下的加速比对比

编译器未进行任何优化，所有代码保持原始结构，导致内存访问效率低下

分块优化在未优化时引入了额外的循环和控制开销，反而降低了性能

循环展开虽然减少了分支预测失败，但在没有寄存器优化的情况下，收益被内存带宽限制抵消

OpenMP 的并行效果成为主要加速来源，但受限于未优化的内存访问模式

2. O1 优化等级的性能突破启用 O1 优化后，性能出现显著提升，特别是分块优化达到 68.9 倍加速。这是因为：

编译器启用了基本优化（如寄存器分配、简单循环优化），大幅减少了内存访问开销

分块优化的缓存局部性优势开始显现，数据复用率提高，L1/L2 缓存命中率提升

OpenMP 版本达到 19.3 倍加速，说明基础并行优化已接近内存带宽上限

循环展开版本（55.0 倍）展现出指令级并行潜力，但尚未充分发挥 SIMD 优势

此时分块大小（64）与 CPU 缓存行匹配度成为关键因素

3. O2 优化等级的极致性能在 O2 优化下，OpenMP+ 循环展开实现惊人的 166.8 倍加速，分块优化也达到 79.1 倍。这是因为：

编译器自动向量化（如 AVX 指令集）使循环展开版本每个周期可处理 8 个 double 运算

分块优化与硬件预取器完美协同，几乎消除了缓存未命中

循环展开彻底消除了分支预测开销，且编译器进行了激进的指令调度

寄存器分配优化使得展开后的计算完全在寄存器中完成，减少内存访问

超线性加速（161.1x）源于：SIMD 宽度利用（理论 8x）+ 指令级并行（3x）+ 缓存优化（7x）的乘积效应

纯 OpenMP 受限内存带宽瓶颈，加速比停滞在 19.8 倍

6 附运行结果

6.1 编译

```
root@worker-0:/public/home/xdzs2025_zza# g++ -fopenmp -o o OB.cpp
root@worker-0:/public/home/xdzs2025_zza# g++ -fopenmp -o1 o1 OB.cpp -O1
/usr/bin/ld: cannot find o1: No such file or directory
collect2: error: ld returned 1 exit status
root@worker-0:/public/home/xdzs2025_zza# g++ -fopenmp -o1 o1 OB.cpp -O1
/usr/bin/ld: cannot find o1: No such file or directory
collect2: error: ld returned 1 exit status
root@worker-0:/public/home/xdzs2025_zza# g++ -fopenmp -o o1 OB.cpp -O1
root@worker-0:/public/home/xdzs2025_zza# g++ -fopenmp -o o2 OB.cpp -O2
```

6.2 openMP 运行

```
root@worker-0:/public/home/xdzs2025_zza# ./o openmp
[openmp] Time: 1591 ms
[openmp] Speedup: 11.3734
[openmp] Valid: 1
root@worker-0:/public/home/xdzs2025_zza# ./o1 openmp
[openmp] Time: 1438 ms
[openmp] Speedup: 19.29
[openmp] Valid: 1
root@worker-0:/public/home/xdzs2025_zza# ./o2 openmp
[openmp] Time: 1501 ms
[openmp] Speedup: 19.7808
[openmp] Valid: 1
```

6.3 分子块运行

```
root@worker-0:/public/home/xdzs2025_zza# ./o block
[block] Time: 2085 ms
[block] Speedup: 8.8283
[block] Valid: 1
root@worker-0:/public/home/xdzs2025_zza# ./o1 block
[block] Time: 403 ms
[block] Speedup: 68.8759
[block] Valid: 1
root@worker-0:/public/home/xdzs2025_zza# ./o2 block
[block] Time: 375 ms
[block] Speedup: 79.104
[block] Valid: 1
```

6.4 循环展开 +openMP 运行

```
root@worker-0:/public/home/xdzs2025_zza# ./o unroll
[unroll] Time: 1816 ms
[unroll] Speedup: 10.005
[unroll] Valid: 1
root@worker-0:/public/home/xdzs2025_zza# ./o1 unroll
[unroll] Time: 504 ms
[unroll] Speedup: 55.0456
[unroll] Valid: 1
root@worker-0:/public/home/xdzs2025_zza# ./o2 unroll
[unroll] Time: 184 ms
[unroll] Speedup: 161.071
[unroll] Valid: 1
```