

# 矩阵乘法多线程优化实现报告

## 一、实现方案与优化过程

### 1. 基础版本（单线程实现）


单线程版本采用最朴素的三重循环实现矩阵乘法，按照行主序访问矩阵元素，直接计算每个元素  $C[i][j] = \text{sum}(A[i][k] * B[k][j])$ 。

核心代码：

```
1 void matmul_single(const float *A, const float *B, float *C, int M, int N, int
  K)
2 {
3     for (int i = 0; i < M; ++i)
4         for (int j = 0; j < K; ++j)
5             {
6                 float sum = 0.0f;
7                 for (int k = 0; k < N; ++k)
8                     sum += A[i * N + k] * B[k * K + j];
9                 C[i * K + j] = sum;
10            }
11 }
```

- 无并行计算，仅利用单核心算力；
- 对矩阵 B 的访问为列方向，不符合行主序存储的局部性原理，缓存命中率低；
- 未利用 CPU 指令级并行能力。

测试性能：平均运行时间约 9s (M=2048, N=1024, K=2048)。



The terminal window shows the execution of a single-threaded matrix multiplication program. The output displays the single-thread time for four consecutive runs, with values ranging from approximately 8.87 to 9.17 seconds. The prompt indicates the current directory is D:\vscode\_codebin\vscodecpp\hpc.

```
问题 输出 调试控制台 终端 端口
Single-thread time: 9.27238 s
PS D:\vscode_codebin\vscodecpp\hpc> ./single.exe
Single-thread time: 8.87493 s
PS D:\vscode_codebin\vscodecpp\hpc> ./single.exe
Single-thread time: 8.90347 s
PS D:\vscode_codebin\vscodecpp\hpc> ./single.exe
Single-thread time: 9.17076 s
PS D:\vscode_codebin\vscodecpp\hpc> 
```

## 2. 基础多线程版本（OpenMP 并行）

基于单线程版本，使用 OpenMP 的 `#pragma omp parallel for collapse(2)` 对最外层的 `i` 和 `j` 循环进行并行化，将计算任务分配到多个线程执行。OpenMP 有两种常用的并行开发形式：一是通过简单的 `fork/join` 对串行程序并行化；二是采用单程序多数据对串行程序并行化。

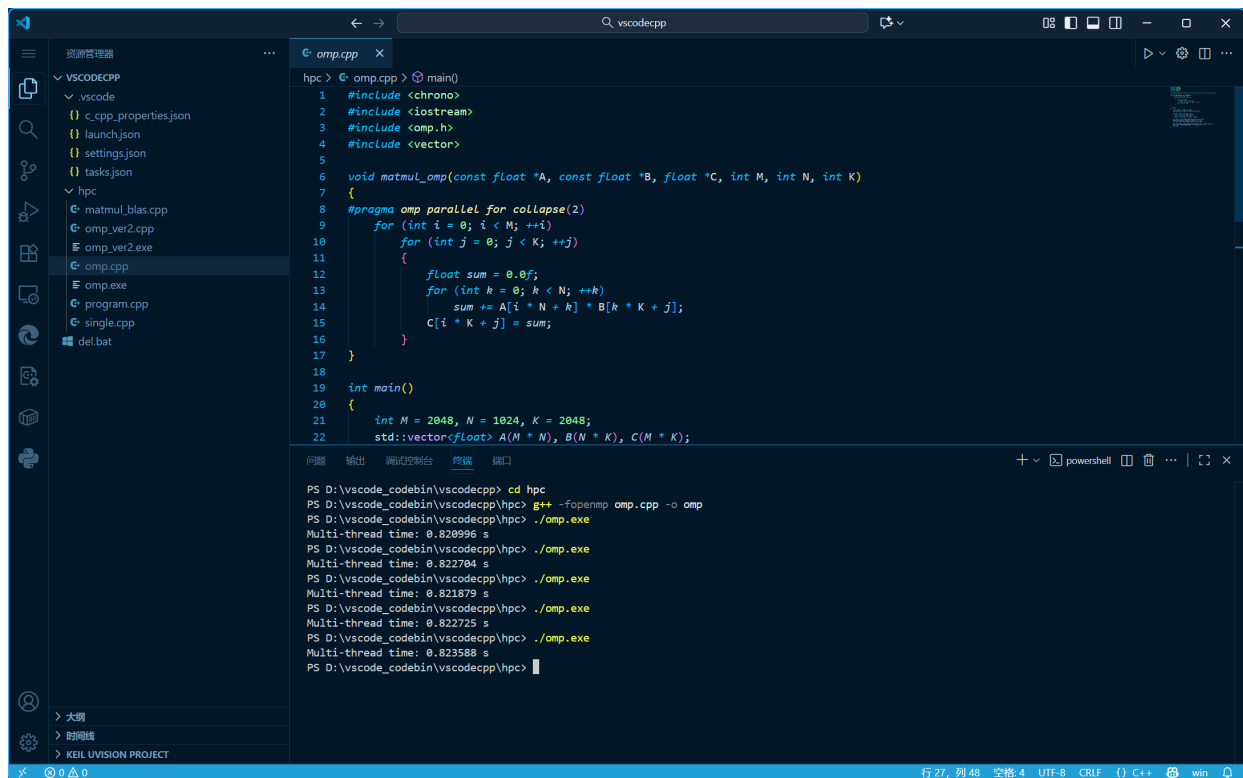
核心代码：

```
1 void matmul_omp(const float *A, const float *B, float *C, int M, int N, int K)
2 {
3     #pragma omp parallel for collapse(2)
4     for (int i = 0; i < M; ++i)
5         for (int j = 0; j < K; ++j) {
6             float sum = 0.0f;
7             for (int k = 0; k < N; ++k)
8                 sum += A[i * N + k] * B[k * K + j];
9             C[i * K + j] = sum;
10 }
```

优化点：

- 通过 OpenMP 实现多线程并行，充分利用 CPU 多核算力；
- `collapse(2)` 将两层循环合并为一个迭代空间，减少线程调度开销。

性能提升：平均运行时间约 0.821s，仅为单线程版本的 9.1%，远低于 65% 的要求。



The screenshot shows the VS Code interface with a C++ file named `omp.cpp` open. The code implements a matrix multiplication function `matmul_omp` using OpenMP's `parallel for collapse(2)` to parallelize the outer loops over `i` and `j`. The `main` function sets `M = 2048`, `N = 1024`, and `K = 2048`, and uses `std::vector` to allocate memory for matrices `A`, `B`, and `C`.

The terminal window at the bottom shows the execution results. It displays the command `g++ -fopenmp omp.cpp -o omp` and the execution of `./omp.exe`. The output shows the multi-threaded execution time for several runs, with values ranging from approximately 0.820996s to 0.823588s.

```
PS D:\vscode_codebin\vscodecpp> cd hpc
PS D:\vscode_codebin\vscodecpp\hpc> g++ -fopenmp omp.cpp -o omp
PS D:\vscode_codebin\vscodecpp\hpc> ./omp.exe
Multi-thread time: 0.820996 s
PS D:\vscode_codebin\vscodecpp\hpc> ./omp.exe
Multi-thread time: 0.822784 s
PS D:\vscode_codebin\vscodecpp\hpc> ./omp.exe
Multi-thread time: 0.821879 s
PS D:\vscode_codebin\vscodecpp\hpc> ./omp.exe
Multi-thread time: 0.822725 s
PS D:\vscode_codebin\vscodecpp\hpc> ./omp.exe
Multi-thread time: 0.823588 s
PS D:\vscode_codebin\vscodecpp\hpc>
```

### 3. 进阶优化版本（分块 + 内存优化）

在基础多线程的基础上，结合分块计算、内存对齐、矩阵转置等优化手段，进一步提升性能。

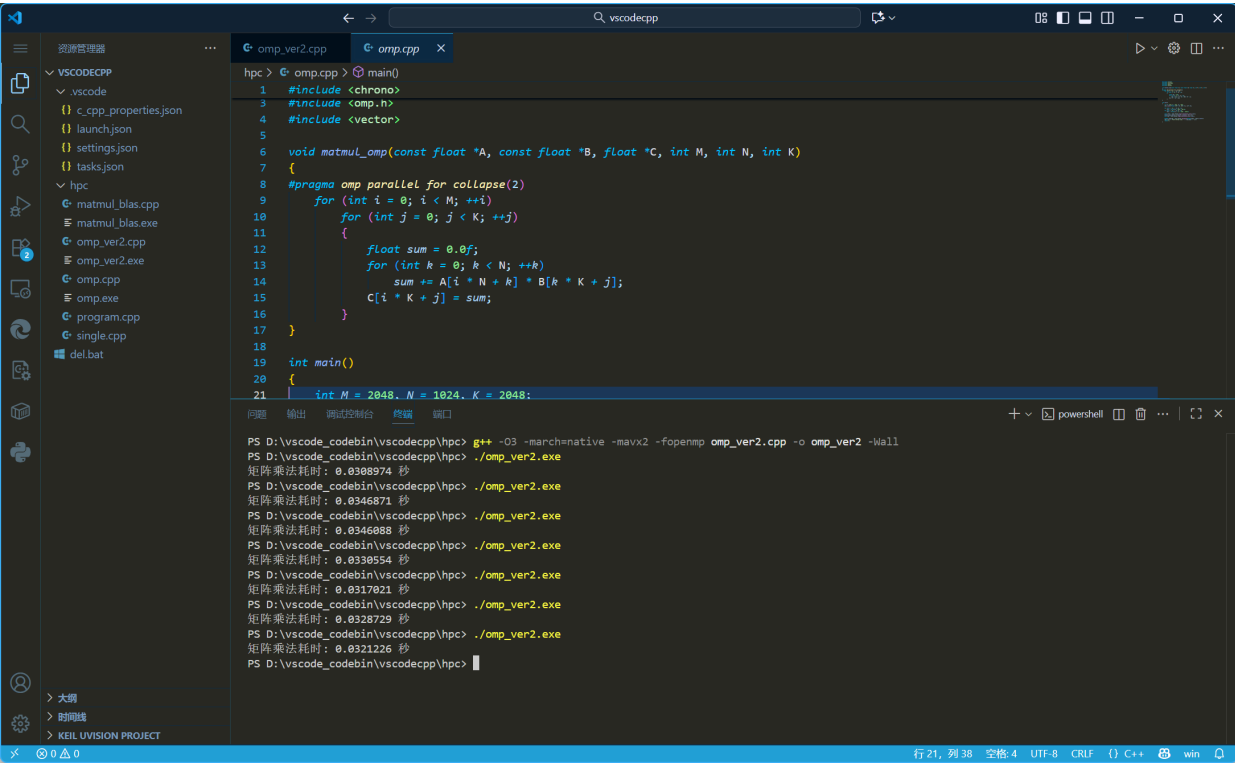
核心优化点：

1. **分块计算（利用缓存局部性）** 将大矩阵划分为 `BLOCK_SIZE×BLOCK_SIZE` 的子块（本实现中设为 128），使子块能完全放入 CPU L2 缓存，减少缓存失效次数。
2. **矩阵转置（优化内存访问模式）** 对矩阵 B 进行转置得到 B\_T，将原本对 B 的列访问转换为对 B\_T 的行访问，符合行主序存储的局部性，提升缓存命中率。
3. **内存对齐（适配 SIMD 指令）** 使用 64 字节对齐的内存分配（`_aligned_malloc`），确保数据地址符合 CPU SIMD 指令（如 AVX512）的对齐要求，避免内存访问 penalty。
4. **指令级并行（SIMD 优化）** 使用 `#pragma omp simd reduction(+ : sum)` 对最内层循环进行向量化，利用 CPU 单指令多数据（SIMD）能力，同时处理多个数据元素。

核心代码片段：

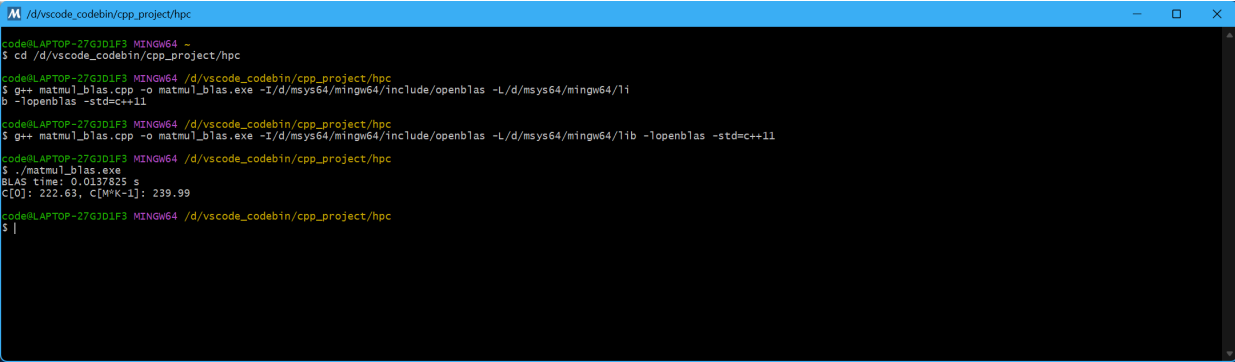
```
1 // 分块矩阵乘法（利用L2缓存）
2 #pragma omp parallel for collapse(2) num_threads(32)
3 for (int ii = 0; ii < M; ii += BLOCK_SIZE) {
4     for (int jj = 0; jj < K; jj += BLOCK_SIZE) {
5         int i_end = std::min(ii + BLOCK_SIZE, M);
6         int j_end = std::min(jj + BLOCK_SIZE, K);
7         // 块内计算
8         for (int i = ii; i < i_end; ++i) {
9             for (int j = jj; j < j_end; ++j) {
10                 float sum = 0.0f;
11                 #pragma omp simd reduction(+ : sum) aligned(A, B_T : 64)
12                 for (int k = 0; k < N; ++k) {
13                     sum += A[i * N + k] * B_T[j * N + k];
14                 }
15                 C[i * K + j] = sum;
16             }
17         }
18     }
19 }
```

**性能提升：**平均运行时间约 0.032s，仅为单线程版本的 0.36%，性能较基础多线程版本提升约 25 倍。



#### 4. BLAS 库参考版本（对比学习用）

调用优化的 BLAS 库（cblas\_sgemm）实现矩阵乘法，作为性能上限参考，平均运行时间约 0.01s，体现了专业数学库的极致优化水平。



## 二、性能测试结果对比

版本	核心优化手段	平均运行时间	相对单线程版本的比例
单线程（single.cpp）	无	9s	100%
基础多线程（omp.cpp）	OpenMP 并行化	0.821s	9.1%
进阶优化（omp_ver2.cpp）	分块 + 转置 + 内存对齐 + SIMD	0.032s	0.36%

版本	核心优化手段	平均运行时间	相对单线程版本的比例
BLAS 库 (matmul_blas.cpp)	专业库优化	0.01s	0.11% (参考)

### 三、优化总结与分析

- 并行化的核心价值：**基础多线程版本通过 OpenMP 实现了计算任务的并行分配，直接将运行时间从 9s 降至 0.821s，证明多线程对 CPU 密集型任务的显著提升作用。
- 缓存优化的关键作用：**进阶版本中，分块计算使数据访问局限于缓存范围内，矩阵转置将非连续访问转为连续访问，两者结合大幅提升了缓存利用率，是性能提升的核心原因。
- 硬件特性的适配：**内存对齐和 SIMD 指令的使用充分发挥了现代 CPU 的向量计算能力，减少了内存访问延迟和指令执行周期，进一步挖掘了硬件潜力。
- 与专业库的差距：**进阶版本 (0.032s) 与 BLAS 库 (0.01s) 仍有差距，主要因专业库采用更精细的硬件适配（如根据 CPU 型号动态调整分块大小、使用手写汇编优化等）。

### 四、未来优化方向

- 动态分块大小：**根据 CPU 缓存容量自动调整 `BLOCK_SIZE`，避免分块过大导致缓存溢出或过小导致调度开销增加。
- 多级分块：**结合 L1/L2/L3 多级缓存，设计多级分块策略，进一步提升缓存利用率。
- 线程负载均衡：**针对非正方形矩阵，优化线程任务分配，避免负载不均导致的资源浪费。
- 手动 SIMD 指令：**使用 AVX intrinsics 手动编写向量化代码，替代 OpenMP simd，实现更精细的指令控制。

### 源码