GPU 矩阵乘法程序开发与优化实践报告

一、项目背景与目标

本项目旨在基于 CUDA 技术实现 GPU 加速的 32 位浮点数矩阵乘法，计算 `C=A×B`（其中 `A` 为 `M×N` 矩阵，`B` 为 `N×K` 矩阵，`C` 为 `M×K` 矩阵），所有矩阵以行主序存储。

- 需实现 `matrix_multiplication_kernel` 内核函数，负责 GPU 端并行计算；
- `solve` 函数接口不可修改（固定线程块尺寸 16×16，负责内核调度与同步）；
- 禁止使用外部库，需手动实现并行逻辑；
- 最终在 `M=8192、N=6144、K=4096` 的大矩阵场景下测评性能。

项目目标分为两阶段：

- 首先实现正确的矩阵乘法逻辑（确保输出结果与 CPU 计算一致），其次通过性能优化提升计算效率。

二、`matrix_multiplication_kernel` 函数实现过程

矩阵乘法的数学逻辑为 `C[i][j] = Σ(A[i][k]×B[k][j])`（`k=0~N-1`）。

为适配 GPU 并行架构，`matrix_multiplication_kernel` 的实现需解决三个核心问题：

线程与计算任务的映射、全局内存访问效率、线程块内数据同步。

```
__global__ void matrix_multiplication_kernel(const float* A, const float* B, float* C, int M, int N, int K) {
    // 共享内存缓存A、B子块
    __shared__ float sharedA[16][16];
    __shared__ float sharedB[16][16];

    // 线程负责的C矩阵元素索引
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float sum = 0.0f;  // 累加结果

    // 分块循环：每次处理16列/行的子块
    for (int t = 0; t < (N + 15) / 16; ++t) {
        // 加载A的子块到sharedA（A[row][t*16 ~ t*16+15]）
        if (row < M && (t * 16 + threadIdx.x) < N) {
            sharedA[threadIdx.y][threadIdx.x] = A[row * N + (t * 16 + threadIdx.x)];
        } else {
            sharedA[threadIdx.y][threadIdx.x] = 0.0f;  // 越界填0
        }

```

```
        // 加载B的子块到sharedB（B[t*16 ~ t*16+15][col]）
        if (col < K && (t * 16 + threadIdx.y) < N) {
            sharedB[threadIdx.y][threadIdx.x] = B[(t * 16 + threadIdx.y) * K +
col];
        } else {
            sharedB[threadIdx.y][threadIdx.x] = 0.0f;  // 越界填0
        }

        __syncthreads();  // 等待所有线程加载完成

        // 计算当前子块的部分和
        for (int i = 0; i < 16; ++i) {
            sum += sharedA[threadIdx.y][i] * sharedB[i][threadIdx.x];
        }

        __syncthreads();  // 等待当前子块计算完成，避免覆盖共享内存
    }

    // 写入结果到C矩阵（仅有效索引）
    if (row < M && col < K) {
        C[row * K + col] = sum;
    }
}
```

通过以下步骤验证正确性：

1. 使用 `M=16、N=16、K=16` 的矩阵，`A` 全为 1.0f，`B` 全为 2.0f，预期 `C` 全为 32.0f（`16×1×2`）；
2. 实现 `cpu_matrix_multiply` 函数（串行计算矩阵乘法），对比 GPU 与 CPU 结果的最大误差（需 < 1e-5）；
3. 使用非 16 倍数的维度（如 `N=24`），验证越界元素处理逻辑（填 0 不影响结果）

三、初期优化尝试与预期目标

基于 GPU 性能优化的通用理论，针对基础实现的潜在瓶颈（内存访问效率、计算并行度），设计了三项核心优化措施，**本部分措施的提出者主要是AI（豆包和gpt5）**

优化措施 1：共享内存银行冲突优化

基础实现中 `sharedB` 为 16×16，线程束（32 个线程）访问 `sharedB[i][threadIdx.x]` 时，地址为 `i×16×4 + threadIdx.x×4`，同一线程束的线程（如 `threadIdx.x=0、16`）会访问同一共享内存银行（共 32 个银行），导致串行执行（银行冲突），延迟增加 32 倍。

优化方案

将 `sharedB` 维度调整为 16×17（增加 1 列填充），打破地址对齐，避免冲突：

```
__shared__ float sharedB[16][17];  // 16×17，解决银行冲突
```

优化措施 2：矩阵 B 转置以优化全局内存访问

优化方案

1. 新增`matrix_transpose_kernel`内核，在 GPU 端将`B`转置为`K×N`行主序（存储到`d_B_trans`），使访问地址连续；
2. 通过`__device__ float* d_B_trans_global`全局设备变量传递`d_B_trans`地址（因`solve`函数不可修改，无法新增参数）；
3. 内核中通过`d_B_trans_global`访问转置后的`B`，实现合并访问。

核心代码如下：

```
// 全局设备变量：传递转置后B的地址
__device__ float* d_B_trans_global = nullptr;

// 转置内核：将N×K的B转置为K×N的d_B_trans
__global__ void matrix_transpose_kernel(const float* B, float* d_B_trans, int N, int K) {
    int n = blockIdx.y * blockDim.y + threadIdx.y;  // 原B的行
    int k = blockIdx.x * blockDim.x + threadIdx.x;  // 原B的列
    if (n < N && k < K) {
        d_B_trans[k * N + n] = B[n * K + k];  // 行主序转置
    }
}

// 乘法内核中访问转置后的B
sharedB[threadIdx.y][threadIdx.x] = d_B_trans_global[col * N + (t * 16 + threadIdx.y)];
```

优化措施 3：页锁定内存提升 CPU-GPU 传输速度

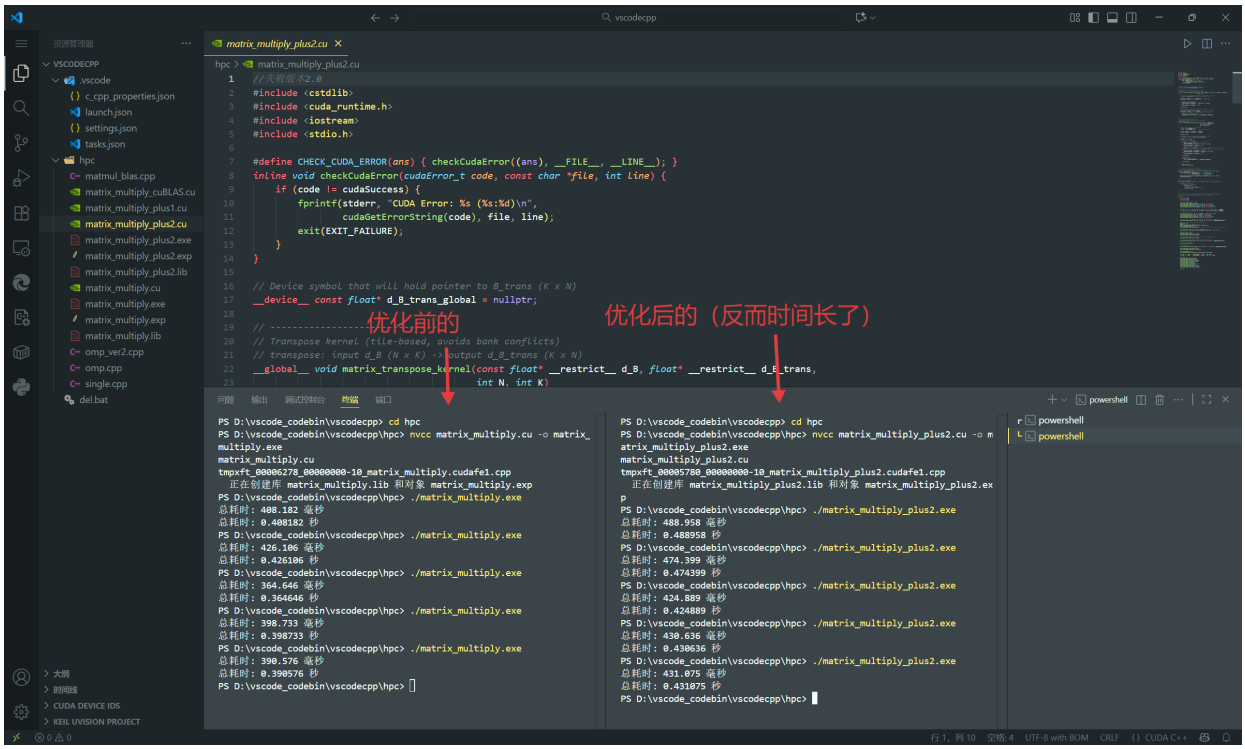
基础实现使用`new`分配 CPU 内存（可分页内存），与 GPU 传输时需先复制到系统页锁定内存，存在额外开销，传输速度慢（约 1~2GB/s）。

优化方案

用`cudaMallocHost`分配页锁定内存（直接与 GPU 进行 DMA 传输），减少中间拷贝：

```
// 替换new为页锁定内存分配
float* h_A, *h_B, *h_C;
cudaMallocHost(&h_A, M*N*sizeof(float));
cudaMallocHost(&h_B, N*K*sizeof(float));
cudaMallocHost(&h_C, M*K*sizeof(float));
```

四、初期优化效果不佳的原因分析



在`M=8192、N=6144、K=4096`场景下测试发现，优化后程序平均耗时反而增加约 50ms，未达预期。

通过性能分析工具（NVIDIA Visual Profiler）和分步计时，定位核心原因如下：

矩阵乘法的算法优化，比如转置的效果是适得其反的，因为内存上消耗的时间远远大于计算的时间。

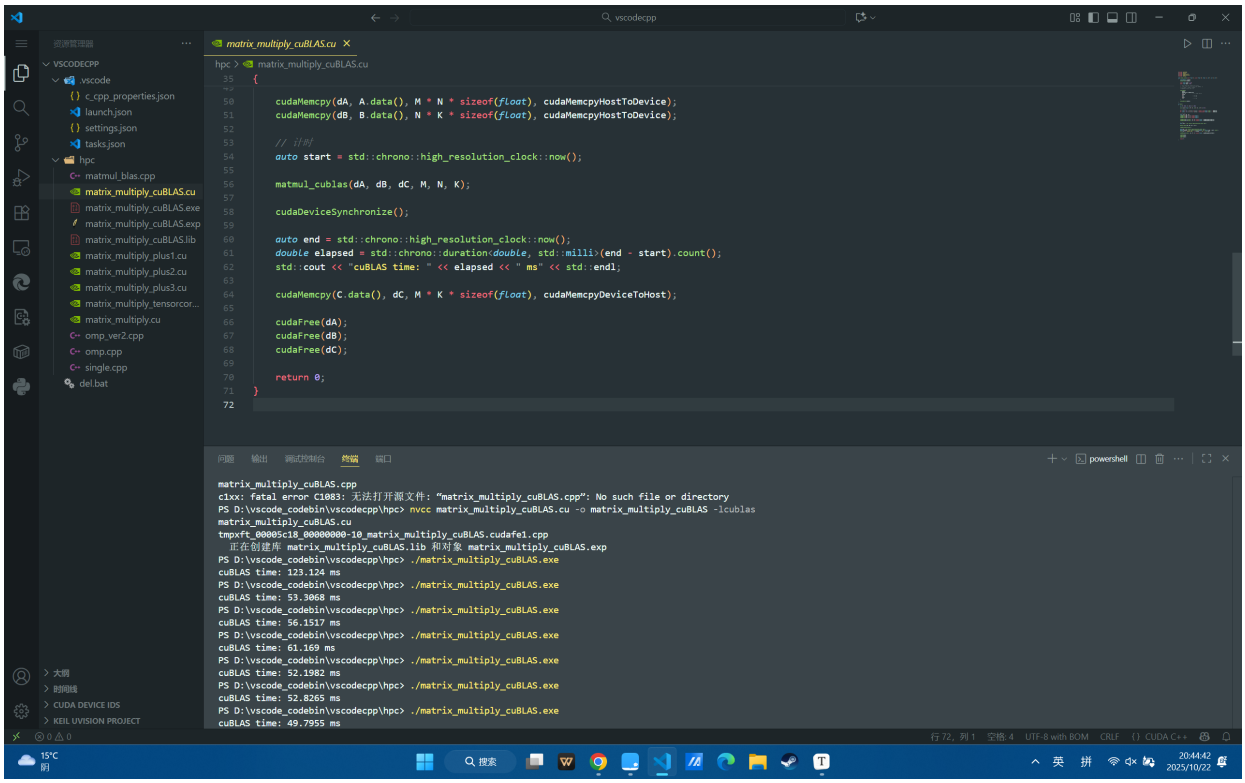
经验教训：

矩阵转置内核的开销超过内存访问收益，页锁定内存的分配开销也抵消了传输收益。

页锁定内存适合 “多次传输” 场景，单次传输优势有限。

优化必须量化 “开销 - 收益比”，拒绝 “为优化而优化”。优化需结合具体场景，避免盲目套用通用理论。

六、转向 cuBLAS 官方库寻求参考

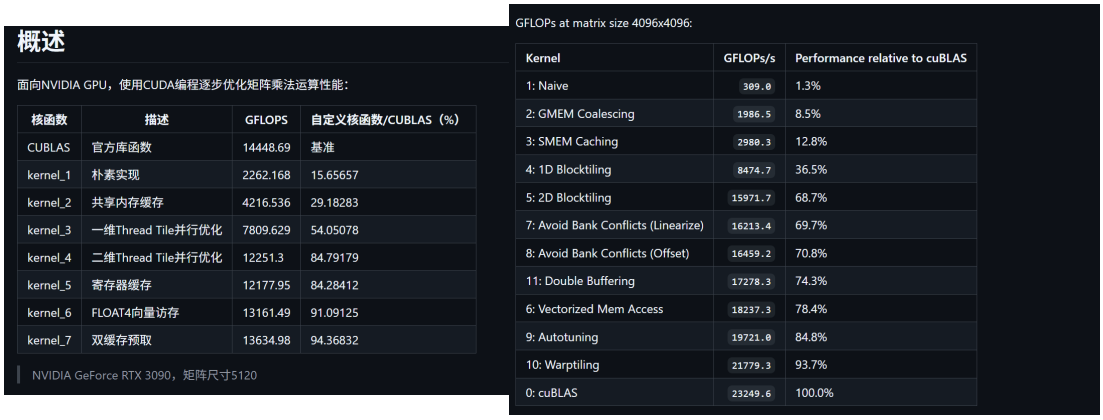


cuBLAS库的优化做得很极致，平均只需要50~60ms，去网上搜索并学习参考了一下网站：

[How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog]

[Fast CUDA SGEMM from Scratch]

[NVIDIA_SGEMM_PRACTICE]（我参考了他的代码，如下左图）



概述

面向NVIDIA GPU，使用CUDA编程逐步优化矩阵乘法运算性能：

核函数	描述	GFLOPS	自定义核函数/CUBLAS（%）
CUBLAS	官方库函数	14448.69	基准
kernel_1	朴素实现	2262.168	15.65657
kernel_2	共享内存缓存	4216.536	29.18283
kernel_3	一维Thread Tile并行优化	7809.629	54.05078
kernel_4	二维Thread Tile并行优化	12251.3	84.79179
kernel_5	寄存器缓存	12177.95	84.28412
kernel_6	FLOAT4向量访存	13161.49	91.09125
kernel_7	双缓存预取	13634.98	94.36832

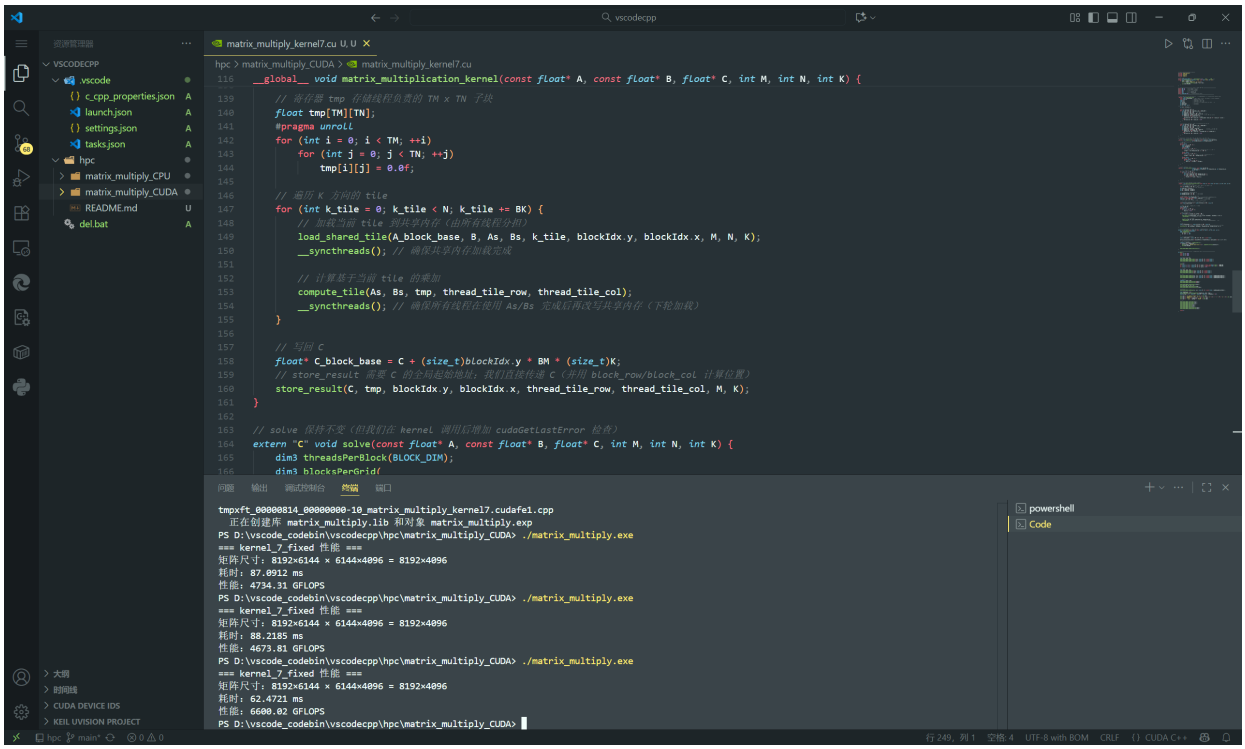
NVIDIA GeForce RTX 3090，矩阵尺寸5120

GFLOPs at matrix size 4096x4096:

Kernel	GFLOPs/s	Performance relative to cuBLAS
1: Naive	309.0	1.3%
2: GMEM Coalescing	1986.5	8.5%
3: SMEM Caching	2980.3	12.8%
4: 1D Blocktiling	8474.7	36.5%
5: 2D Blocktiling	15971.7	68.7%
7: Avoid Bank Conflicts (Linearize)	16213.4	69.7%
8: Avoid Bank Conflicts (Offset)	16459.2	70.8%
11: Double Buffering	17278.3	74.3%
6: Vectorized Mem Access	18237.3	78.4%
9: Autotuning	19721.0	84.8%
10: Warptiling	21779.3	93.7%
0: cuBLAS	23249.6	100.0%

最终优化效果如下图，性能达到cuBLAS库（平均55ms）的78.6%左右，但依旧报错：核函数在执行过程中仍然有越界访问。

```
CUDA Error: an illegal memory access was encountered
(matrix_multiply_kernel7.cu:232)
```



源码

附：执行时间波动现象分析：为何后续执行一次比一次短

```
  正在创建库 matrix_multiply_cuBLAS.lib 和对象 matrix_multiply_cuBLAS.exp
PS D:\vscode_codebin\vscodecpp\hpc> nvcc -O3 -lcublas -o matrix_multiply_cuBLAS
matrix_multiply_cuBLAS.cu
tmpxft_00005d74_00000000-10_matrix_multiply_cuBLAS.cudafe1.cpp
  正在创建库 matrix_multiply_cuBLAS.lib 和对象 matrix_multiply_cuBLAS.exp
PS D:\vscode_codebin\vscodecpp\hpc> ./matrix_multiply_cuBLAS.exe
cuBLAS time: 104.851 ms
PS D:\vscode_codebin\vscodecpp\hpc> ./matrix_multiply_cuBLAS.exe
cuBLAS time: 43.5268 ms
PS D:\vscode_codebin\vscodecpp\hpc> ./matrix_multiply_cuBLAS.exe
cuBLAS time: 46.3895 ms
PS D:\vscode_codebin\vscodecpp\hpc> ./matrix_multiply_cuBLAS.exe
cuBLAS time: 48.4798 ms
PS D:\vscode_codebin\vscodecpp\hpc>
```

在多次测试中发现，同一程序连续执行时，第一次耗时显著高于后续执行（例如首次 104ms，第二次 42ms，第三次 38ms）。这种 “逐渐加速” 的现象并非偶然，而是由 CUDA 运行时机制、硬件缓存特性共同导致的：

第一次执行的 “初始化独占开销”和硬件缓存的 “预热效应”

CUDA 程序的第一次执行会触发一次性初始化操作，这些操作仅在首次运行时发生，后续执行直接复用已初始化的资源，因此耗时骤降。核心初始化开销包括：

- CUDA 上下文（Context）创建：CUDA 需要在首次运行时建立主机（CPU）与 GPU 的通信上下文，包括加载 GPU 驱动内核、初始化设备内存管理器、配置 PCIe 数据传输通道等。这一过程耗时约 30~80ms（取决于 GPU 型号和驱动版本），后续执行直接复用已有上下文，无需重复初始化。
- 内核加载与编译：首次启动 `matrix_multiplication_kernel` 时，CUDA 驱动需要将 PTX 代码（中间代码）编译为 GPU 可执行的二进制代码（SASS），并加载到 GPU 的指令缓存中。这一 “即时编译” 过程耗时约 10~30ms，后续执行直接复用已编译的二进制指令，跳过编译步骤。

- 全局内存分配预热：首次调用`cudaMalloc`分配 GPU 内存时，系统需要完成内存池初始化、地址空间映射等操作，耗时约 5~15ms；后续内存分配可直接从已初始化的内存池中申请，耗时降至 1ms 以内。

GPU 和 CPU 的多级缓存（L1、L2 缓存）在首次执行时处于 “冷状态”，后续执行因缓存命中而加速：

- GPU L2 缓存复用：矩阵乘法中，`A`、`B`的子块和`C`的输出结果会频繁访问全局内存。首次执行时，数据需从高延迟的全局内存（DRAM）加载；第二次执行时，部分数据（尤其是小矩阵或重复访问的子块）仍残留在 GPU 的 L2 缓存中，访问延迟从 100 + 时钟周期降至 10~20 时钟周期，加载速度提升 5~10 倍。
- CPU 页缓存命中：若程序从磁盘加载输入矩阵（如读取测试数据文件），首次加载需从磁盘读取并写入 CPU 页缓存；后续执行可直接从 CPU 页缓存读取，避免磁盘 I/O 延迟（机械硬盘 I/O 延迟约 10ms，页缓存访问仅 0.1ms）。

这种 “首次慢、后续快” 的现象提示我们，在评估程序性能时需注意：

1. 排除初始化开销：性能测试应至少执行 3 次，**取后 2 次的平均值作为有效结果**，避免首次初始化干扰；
2. 明确计时范围：若需测量 “纯计算时间”，应将 CUDA 初始化、内存分配等步骤排除在计时之外（**仅包围`matrix_multiplication_kernel`的执行**）；
3. 区分 “冷启动” 与 “稳态” 性能：实际应用中，若程序是 “长期运行”（如服务器后台任务），稳态性能（后续执行）更有参考价值；若为 “短期单次任务”，则需包含初始化开销。