
中山大学计算机院本科生报告

(2024 学年春季学期)

课程名称: 并行程序设计

批改人:

实验	期末大作业	专业 (方向)	计算机科学与技术计科一班
学号	21307099	姓名	***
Email	***@mail2.sysu.edu.cn	完成日期	2024 年 7 月 16 日

目录

1 报告内容	3
2 理论分析报告	3
2.1 串行通用矩阵乘法及其优化	3
2.1.1 分析	3
2.1.2 实现方法	6
2.1.3 优化建议	10
2.2 基于分布式内存的并行编程框架的矩阵乘法 (MPI)	11
2.2.1 分析	11
2.2.2 实现方法	14
2.2.3 优化建议	21
2.3 基于共享内存的 CPU 多线程编程 (OpenMP)	23
2.3.1 分析	23
2.3.2 实现方法	24
2.3.3 优化建议	24
2.4 GPU 多线程编程 CUDA	28
2.4.1 分析	28

2.4.2 实现方法	34
2.4.3 优化建议	40
3 对比分析	41

1 报告内容

1. 理论分析报告 (section 2): 包括以下各部分的分析、实现方法和优化建议.

- 串行通用矩阵乘法及其优化 (section 2.1).
- 基于分布式内存的并行编程框架的矩阵乘法 (MPI) (section 2.2).
- 基于共享内存的 CPU 多线程编程 (OpenMP) (section 2.3).
- GPU 多线程编程 CUDA (section 2.4).

在分析中给出算法的理论分析, 在实现方法中给出 C/C++/Cuda 或伪代码实现, 在优化建议中给出优化思想.

部分性能分析基于在我的电脑环境 (Ubuntu 20.04, Intel i7-14700K, 96GB RAM, NVIDIA RTX 4070) 中的实验.

2. 对比分析 (section 3): 以矩阵乘法为例, 总结不同编程框架在实现中需要注意的问题.
3. 所有代码均在本人 GitHub 仓库 (LiYJ GEMM) 中, 若 pdf 有任何格式问题可见 tex 源码 (Final Overleaf).

2 理论分析报告

2.1 串行通用矩阵乘法及其优化

该部分代码可见于本人 GitHub 仓库 (Serial GEMM).

2.1.1 分析

串行矩阵乘法即在单处理器上进行矩阵乘法运算.

对于输入矩阵 $\mathbf{A}_{M \times N}$ 和 $\mathbf{B}_{N \times K}$ 计算其乘积 $\mathbf{C}_{M \times K}$ 对于 \mathbf{C} 中的每个元素, 需要计算

$$\mathbf{C}_{ij} = \sum_{k=1}^N \mathbf{A}_{ik} \mathbf{B}_{kj} \quad (1)$$

算法实现见 section 2.1.2

算法优化

1. **空间局部性优化 (section 2.1.2)** 在 algorithm 1 的实现中, 第 6-7 行会对 **B** 进行列访问, 这会导致大量的 **cache miss**(因为矩阵是行主序的), 因此我们需要调整循环顺序为 $i - k - j$, 这样矩阵 **B** 的访问也会变成按行访问.
2. **循环展开 (section 2.1.2)** 我们还可以进行循环展开

这主要有以下优点:

- 减少了循环控制语句 (如增加循环计数器、检查循环条件) 的开销
 - **进一步提高数据局部性** 通过将四次独立的乘法和加法操作组合在一起, 可以更有效地利用 CPU 缓存. $A[i][k]$ 到 $A[i][k+3]$ 和 $B[k][j]$ 到 $B[k+3][j]$ 这些连续的数据访问模式, 有利于提高缓存的命中率. 第一个元素被加载到缓存时, 相邻的元素很可能也被预取到缓存中, 这样随后的访问就可以直接从缓存中获取数据, 而不是较慢的主存.
 - **提高指令级并行性** 现代 CPU 具有指令级并行 (ILP) 能力. 通过在一个迭代中执行多个独立的加法和乘法操作, 这种循环展开策略有助于编译器生成可以更好地利用这种 CPU 能力的指令序列. 这意味着 CPU 可以同时进行多个操作, 而不是顺序执行它们, 从而提高执行效率.
 - **减少数据依赖** 将 $\text{sum}0$ 到 $\text{sum}3$ 的结果累加到 $C[i][j]$, 而不是每次计算后立即写回, 可以减少对 $C[i][j]$ 的写操作次数, 同时消除对这一块寄存器的依赖.
3. **编译优化 (section 2.1.2)** 我们对代码中的矩阵指针添加 `__restrict__` 关键字, 以便于编译器进行优化: `__restrict__` 关键字告诉编译器, 对于指定的指针, 在其生命周期内, 它是访问所指向数据的唯一且未被别名的方式. 即 **A**, **B** 指向的内存区域不会重叠, 这允许编译器更积极地优化.
 4. **Divide and Conquer(section 2.1.2)** Divide and Conquer 方法通过将大矩阵分解成更小的子矩阵, 然后进行分块矩阵乘法来实现, 这样做的好处是可能**提高缓存利用率**: 将大矩阵分块后, 子矩阵的大小更适合缓存, 从而减少 cache miss 的次数, 提高计算效率. 同时它可能减少了 IO 的次数, 从而提高计算效率.

5. **Strassen's Method(section 2.1.2)** Strassen's 方法是一种用于矩阵乘法的快速算法, 它通过将矩阵分解成更小的子矩阵, 并使用递归的方式减少乘法次数, 从而提高计算效率. 但它基本不在实际应用中使用, 这主要有以下原因 (参考stack overflow):

- **高常数因子:** Strassen 方法中的常数因子较高, 使得对于典型应用, Naive 方法更为有效.
- **子矩阵的额外空间:** Strassen 方法中的递归需要额外的空间来存储子矩阵.
- **有限精度和误差积累:** 由于计算机在非整数值上的有限精度, Strassen 方法中累积的误差更大.
- **不适合稀疏矩阵:** 对于实际应用中常见的稀疏矩阵, 有专门设计的更好的方法.
- **实用性** 事实上, 由下图:

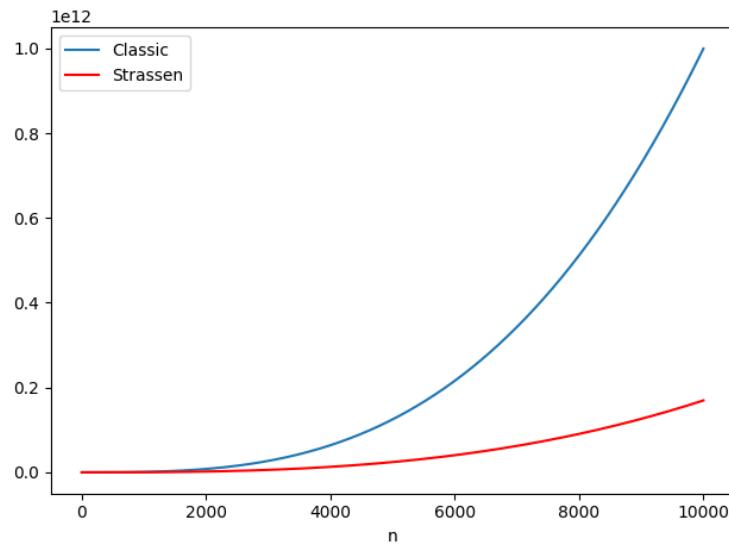


图 1: Why not Strassen

可以看出在很大的矩阵规模下, Strassen 方法才会明显优于传统算法, 但在实际应用中, 大型稠密矩阵乘法的需求并不频繁, 更多的是其中一个或两个是稀疏、对称、三角矩阵.

2.1.2 实现方法

Naive Method 我们假设 \mathbf{C} 矩阵已经初始化为 0.

Algorithm 1 串行矩阵乘法

```
1: Input: 矩阵  $\mathbf{A}_{M \times N}$ ,  $\mathbf{B}_{N \times K}$ 
2: Output: 矩阵  $\mathbf{C}_{M \times K}$ 
3: for  $i = 0$  to  $M - 1$  do
4:     for  $j = 0$  to  $K - 1$  do
5:         for  $k = 0$  to  $N - 1$  do
6:              $\mathbf{C}[i][j] \leftarrow \mathbf{C}[i][j] + \mathbf{A}[i][k] * \mathbf{B}[k][j]$ 
7:         end for
8:     end for
9: end for
```

空间局部性优化 (更改循环顺序)

Algorithm 2 空间局部性优化

```
1: Input: 矩阵  $\mathbf{A}_{M \times N}$ ,  $\mathbf{B}_{N \times K}$ 
2: Output: 矩阵  $\mathbf{C}_{M \times K}$ 
3: for  $i = 0$  to  $M - 1$  do
4:     for  $k = 0$  to  $N - 1$  do
5:         for  $j = 0$  to  $K - 1$  do
6:              $\mathbf{C}[i][j] \leftarrow \mathbf{C}[i][j] + \mathbf{A}[i][k] * \mathbf{B}[k][j]$ 
7:         end for
8:     end for
9: end for
```

循环展开

Algorithm 3 循环展开的矩阵乘法

```
1: Input: 矩阵  $\mathbf{A}_{M \times N}$ ,  $\mathbf{B}_{N \times K}$ 
2: Output: 矩阵  $\mathbf{C}_{M \times K}$ 
3: for  $i = 0$  to  $M - 1$  do
4:   for  $j = 0$  to  $K - 1$  do
5:     sum0  $\leftarrow 0$ 
6:     sum1  $\leftarrow 0$ 
7:     sum2  $\leftarrow 0$ 
8:     sum3  $\leftarrow 0$ 
9:     for  $k = 0$  to  $N - 4$  step 4 do
10:    sum0  $\leftarrow \mathbf{sum0} + \mathbf{A}[i][k] * \mathbf{B}[k][j]$ 
11:    sum1  $\leftarrow \mathbf{sum1} + \mathbf{A}[i][k + 1] * \mathbf{B}[k + 1][j]$ 
12:    sum2  $\leftarrow \mathbf{sum2} + \mathbf{A}[i][k + 2] * \mathbf{B}[k + 2][j]$ 
13:    sum3  $\leftarrow \mathbf{sum3} + \mathbf{A}[i][k + 3] * \mathbf{B}[k + 3][j]$ 
14:   end for
15:   for  $k$  remaining  $N - 1$  do
16:     sum0  $\leftarrow \mathbf{sum0} + \mathbf{A}[i][k] * \mathbf{B}[k][j]$ 
17:   end for
18:    $\mathbf{C}[i][j] \leftarrow \mathbf{sum0} + \mathbf{sum1} + \mathbf{sum2} + \mathbf{sum3}$ 
19: end for
20: end for
```

编译优化

```
1  void matrix_multiplication(int m, int n, int p, double**  
2    __restrict__ A, double** __restrict__ B, double** __restrict__ C  
3  ) {  
4    for(int i = 0; i < m; ++i) {  
5      for (int j = 0; j < p; ++j){  
6        for (int k = 0; k < n; ++k) {  
7          C[i][j] += A[i][k] * B[k][j];  
8        }  
9      }  
10    }  
11  }
```

Compile: gcc -O3 -march=native -funroll-loops <FILENAME.c> -o <FILENAME>

Divide and Conquer

Algorithm 4 Divide and Conquer

```
1: Input: 矩阵  $\mathbf{A}_{M \times N}$ ,  $\mathbf{B}_{N \times K}$ 
2: Output: 矩阵  $\mathbf{C}_{M \times K}$ 
3: if 矩阵  $\mathbf{A}$  或  $\mathbf{B}$  边长为 1 then
4:     直接进行矩阵乘法
5:     for  $i = 0$  to  $M - 1$  do
6:         for  $j = 0$  to  $K - 1$  do
7:              $\mathbf{C}[i][j] \leftarrow 0$ 
8:             for  $k = 0$  to  $N - 1$  do
9:                  $\mathbf{C}[i][j] \leftarrow \mathbf{C}[i][j] + \mathbf{A}[i][k] * \mathbf{B}[k][j]$ 
10:            end for
11:        end for
12:    end for
13: else
14:     将矩阵  $\mathbf{A}$  和  $\mathbf{B}$  分解成子矩阵
15:     计算子矩阵乘法并合并结果
16:      $\mathbf{C}_{11} = \mathbf{A}_{11} \times \mathbf{B}_{11} + \mathbf{A}_{12} \times \mathbf{B}_{21}$ 
17:      $\mathbf{C}_{12} = \mathbf{A}_{11} \times \mathbf{B}_{12} + \mathbf{A}_{12} \times \mathbf{B}_{22}$ 
18:      $\mathbf{C}_{21} = \mathbf{A}_{21} \times \mathbf{B}_{11} + \mathbf{A}_{22} \times \mathbf{B}_{21}$ 
19:      $\mathbf{C}_{22} = \mathbf{A}_{21} \times \mathbf{B}_{12} + \mathbf{A}_{22} \times \mathbf{B}_{22}$ 
20:     将子矩阵结果合并到矩阵  $\mathbf{C}$ 
21: end if
```

Strassen's Method

Algorithm 5 Strassen's Method

```
1: Input: 矩阵  $\mathbf{A}_{M \times N}$ ,  $\mathbf{B}_{N \times K}$ 
2: Output: 矩阵  $\mathbf{C}_{M \times K}$ 
3: if 矩阵  $\mathbf{A}$  或  $\mathbf{B}$  边长为 1 then
4:     直接进行矩阵乘法
5:     for  $i = 0$  to  $M - 1$  do
6:         for  $j = 0$  to  $K - 1$  do
7:              $\mathbf{C}[i][j] \leftarrow 0$ 
8:             for  $k = 0$  to  $N - 1$  do
9:                  $\mathbf{C}[i][j] \leftarrow \mathbf{C}[i][j] + \mathbf{A}[i][k] * \mathbf{B}[k][j]$ 
10:            end for
11:        end for
12:    end for
13: else
14:     将矩阵  $\mathbf{A}$  和  $\mathbf{B}$  分解成子矩阵
15:     计算 Strassen's 方法的七个乘积
16:      $M1 \leftarrow (\mathbf{A}_{11} + \mathbf{A}_{22}) \times (\mathbf{B}_{11} + \mathbf{B}_{22})$ 
17:      $M2 \leftarrow (\mathbf{A}_{21} + \mathbf{A}_{22}) \times \mathbf{B}_{11}$ 
18:      $M3 \leftarrow \mathbf{A}_{11} \times (\mathbf{B}_{12} - \mathbf{B}_{22})$ 
19:      $M4 \leftarrow \mathbf{A}_{22} \times (\mathbf{B}_{21} - \mathbf{B}_{11})$ 
20:      $M5 \leftarrow (\mathbf{A}_{11} + \mathbf{A}_{12}) \times \mathbf{B}_{22}$ 
21:      $M6 \leftarrow (\mathbf{A}_{21} - \mathbf{A}_{11}) \times (\mathbf{B}_{11} + \mathbf{B}_{12})$ 
22:      $M7 \leftarrow (\mathbf{A}_{12} - \mathbf{A}_{22}) \times (\mathbf{B}_{21} + \mathbf{B}_{22})$ 
23:     计算最终子矩阵
24:      $\mathbf{C}_{11} \leftarrow M1 + M4 - M5 + M7$ 
25:      $\mathbf{C}_{12} \leftarrow M3 + M5$ 
26:      $\mathbf{C}_{21} \leftarrow M2 + M4$ 
27:      $\mathbf{C}_{22} \leftarrow M1 - M2 + M3 + M6$ 
28:     将子矩阵结果合并到矩阵  $\mathbf{C}$ 
29: end if
```

2.1.3 优化建议

对于串行的矩阵乘法，性能瓶颈主要在于

- **cache miss:** 矩阵元素在内存中的行主序存储，若访问模式不匹配可能导致大量的 cache miss，尤其是当处理大矩阵时
- **计算与内存访问的比例:** 在传统的三重循环矩阵乘法中，每次计算操作相对于内存访问较少，这导致了计算资源的利用率不高.
- **指令流水线的利用:** 如果循环的迭代次数不足以填满 CPU 的指令流水线，或者循环中存在大量的依赖，将导致流水线效率降低.

针对上述瓶颈，可以采取以下优化措施：

- **优化内存访问模式:** 调整数据的访问顺序，使之更符合缓存的预取逻辑，例如通过循环置换等技术改善缓存利用率.
- **编译优化** 配合编译器编码并使用相应的编译选项，如-O3, -march=native, -funroll-loops 等，以提高代码的执行效率
- **增加计算强度:** 使用 Block Matrix Multiplication 等方法减少内存访问频率，增加每次内存访问后的计算量.
- **利用指令级并行:** 对循环进行调整，减少迭代间的数据依赖，以增加指令级并行性，并利用编译器优化或手动向量化提高性能.
- **向量化的使用:** 现代处理器具有向量指令集，能够在单个操作中处理多个数据点. 传统的串行矩阵乘法可能没有充分利用这一特性，从而错过了加速的机会.

实验后的性能如下图，基本符合上述的分析

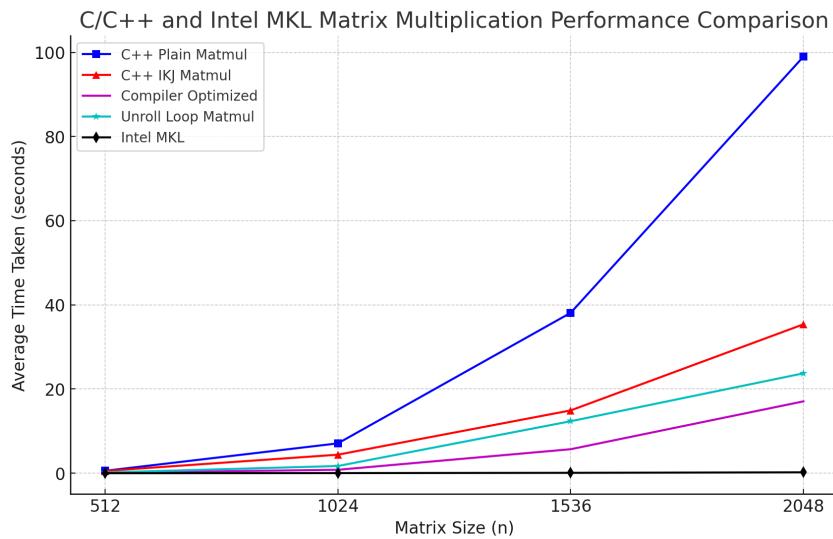


图 2: Serial GEMM Performances

2.2 基于分布式内存的并行编程框架的矩阵乘法 (MPI)

该部分代码可见于本人 GitHub 仓库 (MPI GEMM 1 和 MPI GEMM 2).

2.2.1 分析

从通信方式上区分, 有两种实现方式, 即使用点对点通信 (MPI GEMM 1) 和使用集合通信 (MPI GEMM 2).

从数据划分方式上, 至少有两种实现方式, 比如

- 每个进程计算 \mathbf{A} 的部分行和完整的 \mathbf{B} 的乘积, 从而得到 \mathbf{C} 的一些行
- 每个进程计算 \mathbf{A} 的部分行和 \mathbf{B} 的部分列的乘积, 从而得到 \mathbf{C} 的一块

这两种划分方式的总计算量显然是一样的. 假设对输入 $\mathbf{A}_{M \times N}$ 和 $\mathbf{B}_{N \times K}$ 计算其乘积, 有 P 个进程, 则 Slave process 有 $P-1$ 个.

第一种划分方式中, 每个进程被分到的数据量为

$$\frac{MN}{P-1} + NK \quad (2)$$

计算量为

$$\frac{2MNK}{P-1} \quad (3)$$

第二种划分方式中, 每个进程被分到的数据量为

$$\frac{MN}{p} + \frac{NK}{q}, \quad pq = P-1 \quad (4)$$

计算量为

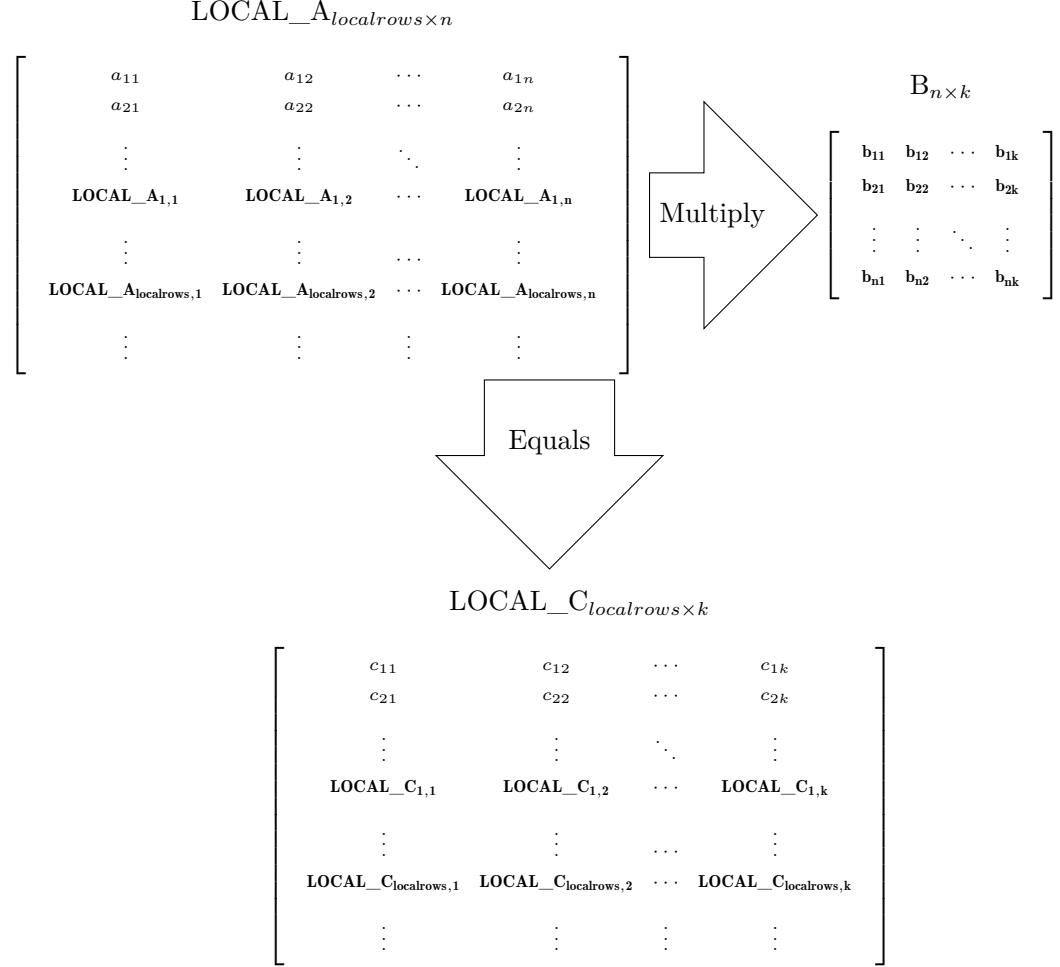
$$\frac{2MNK}{P-1} \quad (5)$$

显然, 在第二种情形中, 除非

$$p = P-1, q = 1 \quad (6)$$

否则每个进程分得的数据量比第一种更少, 理论上第二种分发方式可以更快. 同时由于数据量更少, 空间局部性也要优于第一种分发

1. 点对点通信 点对点通信使用 MPI_Send 和 MPI_Recv(族) 函数，并采用 **Master-Slave 模式**, 即有一个负责数据分发的主进程 (fig. 3) 以及 $n - 1$ 个负责运算的从进程 (fig. 4). 以主进程负责分发 **A** 的部分行和完整的 **B**, 同时接收 **C** 的部分行进行拼凑为例:



2. 集合通信 一个显而易见的优化是使用 `mpi_type_create_struct` 聚合 MPI 进程内变量后通信, 这样可以减少通信的次数, 而进程间通信的开销是非常大的. 我们分别讨论两种数据划分方式:

- (a) 每个进程计算 **A** 的部分行和完整的 **B** 的乘积, 从而得到 **C** 的一些行只需对 **A** 的行做填充 (可能不能整除), 然后将填充后的 **temp_A** 按行均匀拆分成 `comm_size` 部分, 后面跟上完整的 **B**, 塞入缓冲区 (fig. 6 左半部分和 fig. 7) 即可.
- (b) 每个进程计算 **A** 的部分行和 **B** 的部分列的乘积, 从而得到 **C** 的一块将 **A** 分成 `A_blocks` 块, **B** 分成 `B_blocks` 块, 每块的乘积为 **C** 的一部分. 相应地, 按分块矩阵相乘的顺序填入缓冲区 (fig. 6 右半部分和 fig. 8)

矩阵乘法采用朴素的矩阵乘法. 数据分发的实现过程见 section 2.2.2, 矩阵乘法的实现过程见 section 2.2.2

相比于点对点通信:

- (a) 当进程数量较少时, 通信并没有占满总线带宽, 而集合通信可以更好地利用通信资源, 因此集合通信的性能要优于点对点通信.
- (b) 而当进程数量较多时, 工作负载分布更加均匀, 点对点通信的频繁通信开销被更多的进程分摊, 单个进程的通信压力减小. 同时随着进程数量增加, 通信资源的使用接近饱和状态, 此时点对点通信和集合通信的性能差异不太大, 因为都受到总线带宽瓶颈的限制.
- (c) 另外在进程数量较多的情况下, 通信操作更为频繁且并行度更高, 总体的同步开销相对减少 (因为数据量减少了), 两种通信方式的性能差异因此减小

2.2.2 实现方法

点对点通信

Master 进程中的数据分发和 Slave 中的接收:

```

1 //include "MPI_P2P_Matrix.hpp"
2
3 /* MPI点对点矩阵乘法
4 * @param A A[n*k]
5 * @param B B[n*k]
6 * @param C C[n*k]=AB
7 * @param elapsed 计算时间(s)
8 * @param comm_size 进程总数
9 */
10 void mpi_matmul_main_process(const Matrix &A, const Matrix &B, Matrix &C, double &elapsed, const int comm_size)
11 {
12     double start_time = MPI_Wtime();
13     if (comm_size == 1) {
14         sequential_matmul(A, B, C);
15         double end_time = MPI_Wtime();
16         elapsed = end_time - start_time;
17         return;
18     }
19     // 开始计时
20     int m = A.rows;
21     int n = A.cols;
22     int k = B.cols;
23     /* 找出除了主进程外的所有进程，将整个B发送给每个进程
24      * 主进程在本程序的两个进程中计算 LOCAL_A_[localrows * n] * B_[n * k] = LOCAL_C_[localrows * k]
25      * 主进程负责分发矩阵LOCAL_C_，组织局部的
26      * 主进程不参与分配，因此-1
27      */
28     /* - rows_per_process = 每个进程的行数
29     * - size_per_process = 每个进程的大小
30     */
31     /* remaining_rows = 剩余平行时的剩余行数(当发送最后一个进程的行数)
32     * (remaining_size - 1) * n = 剩余平行时的剩余行数(发送最后一个进程的大小)
33     */
34     int rows_per_process = m / (comm_size - 1);
35     int size_per_process = rows_per_process * n;
36     int remaining_rows = m % (comm_size - 1) + rows_per_process;
37     int remaining_size = remaining_rows * n;
38
39     /* 如果发送的是主进程：
40      * 第一次除了最后一个进程发送size_per_process的矩阵A(TAG=1)和完整矩阵B(TAG=2)
41      * 然后最后一个进程发送remaining_size的矩阵A和完整矩阵B
42      * 然后等待接收LOCAL_C_[localrows * k]即可
43      */
44
45     // 发送阶段
46     for (int process_id = 1; process_id < comm_size - 1; process_id++) {
47         MPI_Send(&A.MAT[size_per_process * (process_id - 1)], size_per_process, MPI_DOUBLE, process_id, 1, MPI_COMM_WORLD);
48         MPI_Send(B.MAT, n * k, MPI_DOUBLE, process_id, 2, MPI_COMM_WORLD);
49     }
50     MPI_Send(&A.MAT[size_per_process * (comm_size - 2)], remaining_size, MPI_DOUBLE, comm_size - 1, 1, MPI_COMM_WORLD);
51     MPI_Send(&B.MAT, n * k, MPI_DOUBLE, comm_size - 1, 2, MPI_COMM_WORLD);
52
53     // 接收阶段
54     for (int process_id = 1; process_id < comm_size - 1; process_id++) {
55         MPI_Recv(&C.MAT[rows_per_process * k * (process_id - 1)], rows_per_process * k, MPI_DOUBLE, process_id, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
56     }
57     MPI_Recv(&C.MAT[rows_per_process * k * (comm_size - 2)], remaining_rows * k, MPI_DOUBLE, comm_size - 1, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
58     double end_time = MPI_Wtime();
59     elapsed = end_time - start_time;
60     return;
61 }

```

图 3: Master

Slave 进程中的数据计算:

```

62 /* 每个进程中计算 LOCAL_A_[localrows * n] * B_[n * k] = LOCAL_C_[localrows * k]
63 * 需注意最后一个进程计算 LOCAL_A_[remainingrows * n] * B_[n * k] = LOCAL_C_[remainingrows * k]
64 */
65 void mpi_matmul_worker_process(const int m, const int n, const int k, const int my_rank, const int comm_size)
66 {
67     int rows_per_process = m / (comm_size - 1);
68     int size_per_process = rows_per_process * n;
69     int remaining_rows = m % (comm_size - 1) + rows_per_process;
70     int remaining_size = remaining_rows * n;
71
72     if (my_rank != comm_size - 1) {
73         Matrix LOCAL_A(rows_per_process, n, false);
74         Matrix LOCAL_B(n, k, false);
75         Matrix LOCAL_C(rows_per_process, k, false);
76         MPI_Recv(LOCAL_A.MAT, size_per_process, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
77         MPI_Recv(LOCAL_B.MAT, n * k, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
78         sequential_matmul(LOCAL_A, LOCAL_B, LOCAL_C);
79         //std::cout << "RANK " << my_rank << " LOCAL_C" << std::endl;
80         //LOCAL_C.print_matrix();
81
82         MPI_Send(LOCAL_C.MAT, rows_per_process * k, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD);
83     } else {
84         Matrix LOCAL_A(remaining_rows, n, false);
85         Matrix LOCAL_B(n, k, false);
86         Matrix LOCAL_C(remaining_rows, k, false);
87         MPI_Recv(LOCAL_A.MAT, remaining_size, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
88         MPI_Recv(LOCAL_B.MAT, n * k, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
89         sequential_matmul(LOCAL_A, LOCAL_B, LOCAL_C);
90         //std::cout << "RANK " << my_rank << " LOCAL_C" << std::endl;
91         //LOCAL_C.print_matrix();
92
93         MPI_Send(LOCAL_C.MAT, remaining_rows * k, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD);
94     }
95     return;
96 }

```

图 4: Slave

集合通信

自定义的数据结构:

```
2 // 自定义MPI派生类型
3 void build_mpi_mat_combined_type(const BUFFER *const buffer, MPI_Datatype *COMBINED_MATRIX)
4 {
5     int size_A = buffer->Get_A_size_per_process();
6     int size_B = buffer->Get_B_size_per_process();
7     int array_of_blocklengths[2] = {size_A, size_B};
8     MPI_Aint array_of_displacements[2] = {0, static_cast<MPI_Aint>(size_A * sizeof(double))};
9     MPI_Datatype array_of_types[2] = {MPI_DOUBLE, MPI_DOUBLE};
10
11     MPI_Type_create_struct(2, array_of_blocklengths, array_of_displacements, array_of_types, COMBINED_MATRIX);
12     MPI_Type_commit(COMBINED_MATRIX);
13 }
14
15 }
```

图 5: mpi_type_create_struct

缓冲区：

```
A_Perf.cpp // Buffer.h > Shuffer.h > BUFFER.h; int, int, int)
1 class Matrix {
2     class BUFFER {
3         public:
4             Shuffer *shuffer;
5             int A_rows_per_process;
6             int A_cols_per_process;
7             int B_rows_per_process;
8             int B_cols_per_process;
9             int com_size;
10        };
11    public:
12        BUFFER() : Buffer(multip), A_rows_per_process(0), A_cols_per_process(0), B_cols_per_process(0) {};
13        BUFFER(int ar, int ac, int br, int bc) : Buffer(multip), A_rows_per_process(ar), A_cols_per_process(ac), B_rows_per_process(bc), com_size(ar*ac);
14        ~BUFFER()
15        {
16            if (buffer != nullptr) {
17                delete[] buffer;
18            }
19            buffer = nullptr;
20        }
21    };
22    int a_m_alignment();
23    void buffer_initMatrix_AA(const Matrix AB, const int pad);
24    inline int Get_A_size_per_process() const;
25    inline int Get_B_size_per_process() const;
26    inline int Get_Buffer_size() const;
27 };
28
29 inline int BUFFER::Get_A_size_per_process() const
30 {
31     return A_rows_per_process * A_cols_per_process;
32 }
33
34 inline int BUFFER::Get_B_size_per_process() const
35 {
36     return A_cols_per_process * B_cols_per_process;
37 }
38
39 inline int BUFFER::Get_Buffer_size() const
40 {
41     return Get_A_size_per_process() + Get_B_size_per_process();
42 }
43
44
A_Perf.cpp // Matrix.h > MatrixUtil.h > MatrixUtilPerf.h; int, int, int)
45 class Matrix {
46     class BUFFER {
47         public:
48             Shuffer *shuffer;
49             int A_rows_per_process;
50             int A_cols_per_process;
51             int B_rows_per_process;
52             int B_cols_per_process;
53             int com_size;
54             int A_blocks;
55             int B_blocks;
56        };
57    public:
58        BUFFER() : Buffer(multip), A_rows_per_process(0), A_cols_per_process(0), B_cols_per_process(0), A_blocks(0), B_blocks(0) {};
59        BUFFER(int ar, int ac, int br, int bc) : Buffer(multip), A_rows_per_process(ar), A_cols_per_process(ac), B_rows_per_process(br), B_cols_per_process(bc), com_size(ar*ac), A_blocks(ar), B_blocks(br) {};
60        ~BUFFER()
61        {
62            if (buffer != nullptr) {
63                delete[] buffer;
64            }
65            buffer = nullptr;
66        }
67    };
68    int a_m_alignment();
69    void buffer_initMatrix_AB(const Matrix AB, const int A_pad_rows, const int B_pad_cols);
70
71    inline int Get_A_size_per_process() const;
72    inline int Get_B_size_per_process() const;
73    inline int Get_Buffer_size() const;
74 };
75
76 inline int BUFFER::Get_A_size_per_process() const
77 {
78     return A_rows_per_process * A_cols_per_process;
79 }
80
81 inline int BUFFER::Get_B_size_per_process() const
82 {
83     return A_cols_per_process * B_cols_per_process;
84 }
85
86 inline int BUFFER::Get_Buffer_size() const
87 {
88     return Get_A_size_per_process() + Get_B_size_per_process();
89 }
```

图 6: Buffer

缓冲区的填充:

1. 分发方式 1

```
A_Partitioning > buffer.cpp > buffer_init(Matrix &A, const Matrix &B, const int pad)
1  #include "Matrix.hpp"
2  #include "buffer.hpp"
3
4  int BUFFER::size_alignment()
5  {
6      int pad_rows = (this->comm_size - this->A_rows_per_process % this->comm_size) %
7          this->comm_size;
8      int new_m = this->A_rows_per_process + pad_rows;
9      this->A_rows_per_process = new_m / comm_size;
10     return pad_rows;
11 }
12
13 void BUFFER::buffer_init(Matrix &A, const Matrix &B, const int pad)
14 {
15     Matrix temp_A(A);
16     temp_A.row_padding(pad);
17     int A_size_per_process = Get_A_size_per_process();
18     int B_size_per_process = Get_B_size_per_process();
19     int size_per_process = A_size_per_process + B_size_per_process;
20     buffer = new double[size_per_process * comm_size]();
21     for (int i = 0; i < comm_size; i++) {
22         for (int j = 0; j < A_size_per_process; j++) {
23             buffer[j + size_per_process * i] = temp_A.MAT[j + A_size_per_process * i];
24         }
25         for (int j = 0; j < B_size_per_process; j++) {
26             buffer[(j + A_size_per_process) + size_per_process * i] = B.MAT[j];
27         }
28     }
29 }
```

图 7: Partition A

2. 分发方式 2

```

1. #include "Matrix.hpp"
2. #include "buffer.hpp"
3. #include <cmath>
4.
5. int BUFFER::m_alignment()
6 {
7     //std::cout << "A_BLOCKS" << A_blocks << std::endl;
8     int A_pad_rows = (this->A_blocks - this->A_rows_per_process % this->A_blocks) %
9         this->A_blocks;
10    int new_m = this->A_rows_per_process + A_pad_rows;
11    this->A_rows_per_process = new_m / A_blocks;
12    return A_pad_rows;
13 }
14
15 int BUFFER::k_alignment()
16 {
17     //std::cout << "B_BLOCKS" << B_blocks << std::endl;
18     int B_pad_cols = (this->B_blocks - this->B_cols_per_process % this->B_blocks) %
19         this->B_blocks;
20     int new_k = this->B_cols_per_process + B_pad_cols;
21     this->B_cols_per_process = new_k / B_blocks;
22     return B_pad_cols;
23 }
24
25 void BUFFER::partition()
26 {
27     int sqrt_m = static_cast<int>(sqrt(comm_size + 0.1)); // 计算平方根并向下取整
28     for (int factor = sqrt_m; factor >= 1; --factor) {
29         if (comm_size % factor == 0) {
30             this->A_blocks = comm_size / factor;
31             this->B_blocks = factor;
32             break;
33         }
34     }
35     return;
36 }
37
38 void BUFFER::buffer_init(Matrix &A, const Matrix &B, const int A_pad_rows, const int B_pad_cols)
39 {
40     Matrix temp_A(A);
41     Matrix temp_B(B);
42     temp_A.row_padding(A.pad_rows);
43     temp_B.col_padding(B.pad_cols);
44     int A_size_per_process = Get_A_size_per_process();
45     int B_size_per_process = Get_B_size_per_process();
46     int size_per_process = A_size_per_process * B_size_per_process;
47     //std::cout << "INIT CHECK" << B.size_per_process << endl;
48     buffer = new double[size_per_process * comm_size];
49     for (int i = 0; i < A_blocks; ++i) {
50         for (int j = 0; j < B_blocks; ++j) {
51             for (int k = 0; k < A_size_per_process; ++k) {
52                 buffer[(i * B_blocks + j) * size_per_process + k] = temp_A.MAT[i * A_size_per_process + k];
53             }
54         for (int k = A_size_per_process; k < size_per_process; ++k) {
55             //std::cout << "k check" << i << ' ' << j << ' ' << (k - A_size_per_process) << ' ' << (k - A_size_per_process) / B_size_per_process * temp_B;
56             buffer[(i * B_blocks + j) * size_per_process + k] = temp_B.MAT[(k - A_size_per_process) / B_size_per_process * temp_B.size_per_process + (k - A_size_per_process) % B_size_per_process];
57         }
58     }
59 }
60 //std::cout << "b" << A_blocks << ' ' << B_blocks << ' ' << A_size_per_process << ' ' << size_per_process << endl;
61 //print_mat(buffer, comm_size, size_per_process);
62

```

图 8: Partition A and B

此处给出 49-59 行代码的说明: 我们在 56 行的代码处计算每一块的 LOCAL_B[k] 在原始 B 中的坐标, 算法如下:

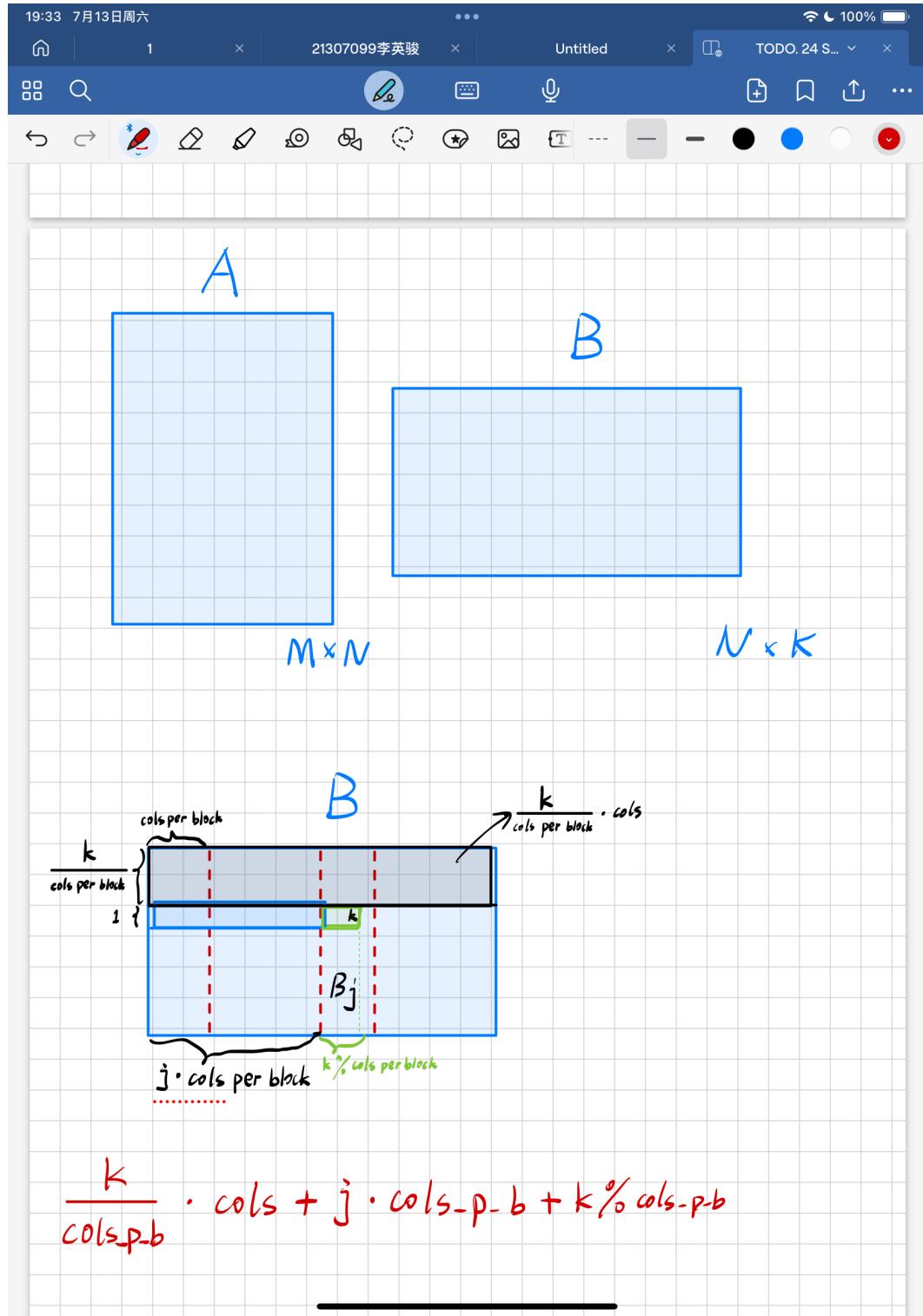


图 9: Partition A and B

矩阵乘法主过程 (详见注释):

1. 分发方式 1

```
16  /** MPI混合通信矩阵乘法
17  * @param Buffer 缓冲区
18  * @param C_C_【m=k】=AB
19  * @param COMBINED_MATRIX 变量类型指针
20  */
21 void mpi_matmul_main_process(const BUFFER *const Buffer, Matrix &C, MPI_Datatype *COMBINED_MATRIX)
22 {
23     // 计算每个进程分配的数据大小, 用于生成local_buffer
24     int size_local_A = Buffer->Get_A_size_per_process();
25     int size_local_B = Buffer->Get_B_size_per_process();
26     // 计算每个进程算出的C的大小, 用于生成local_C
27     int size_local_C = Buffer->A_rows_per_process * Buffer->B_cols_per_process;
28     // 计算全局总并后的comm_C的大小(在行数不相同时, 可能有填充)
29     int size_comm_C = Buffer->comm_size * size_local_C;
30     // 计算最终结果C的大小
31     int size_C = C.rows * C.cols;
32
33     double *local_buffer = new double[size_local_A + size_local_B]();
34     double *local_C = new double[size_local_C]();
35     double *comm_C = new double[size_comm_C]();
36
37     // 单次散射
38     MPI_Scatter(Buffer->buffer, 1, *COMBINED_MATRIX, local_buffer, 1, *COMBINED_MATRIX, 0, MPI_COMM_WORLD);
39
40     /*std::cout<<"A"<<std::endl;
41     print_mat(local_buffer,Buffer->A_rows_per_process,Buffer->A_cols_per_process);
42     std::cout << "B" << std::endl;
43     print_mat(local_buffer+size_local_A,Buffer->A_cols_per_process, Buffer->B_cols_per_process);*/
44     sequential_matmul(local_buffer, local_buffer + size_local_A, local_C, Buffer->A_rows_per_process, Buffer->A_cols_per_process, Buffer->B_cols_per_process);
45     /*std::cout << "C" << std::endl;
46     print_mat(local_C,Buffer->A_rows_per_process,Buffer->B_cols_per_process);*/
47
48     // 单次聚集
49     MPI_Gather(local_C, size_local_C, MPI_DOUBLE, comm_C, size_local_C, MPI_DOUBLE, 0, MPI_COMM_WORLD);
50
51     for (int i = 0; i < size_C; ++i) {
52         C.MAT[i] = comm_C[i];
53     }
54
55     delete[] local_buffer;
56     delete[] local_C;
57     delete[] comm_C;
58     local_buffer = nullptr;
59     local_C = nullptr;
60     comm_C = nullptr;
61 }
62
63 /** 每个进程中计算 LOCAL_A_{localrows * n} * B_{n * k} = LOCAL_C_{localrows * k}
64 */
65 void mpi_matmul_worker_process(const BUFFER *const Buffer, MPI_Datatype *COMBINED_MATRIX)
66 {
67     int size_A = Buffer->Get_A_size_per_process();
68     int size_B = Buffer->Get_B_size_per_process();
69     int size_C = Buffer->A_rows_per_process * Buffer->B_cols_per_process;
70     double *local_buffer = new double[size_A + size_B];
71     double *local_C = new double[size_C];
72     MPI_Scatter(nullptr, 0, MPI_DATATYPE_NULL, local_buffer, 1, *COMBINED_MATRIX, 0, MPI_COMM_WORLD);
73     /*std::cout << "A" << std::endl;
74     print_mat(local_buffer, Buffer->A_rows_per_process, Buffer->A_cols_per_process);
75     std::cout << "B" << std::endl;
76     print_mat(local_buffer + size_A, Buffer->A_cols_per_process, Buffer->B_cols_per_process);*/
77
78     sequential_matmul(local_buffer, local_buffer + size_A, local_C, Buffer->A_rows_per_process, Buffer->A_cols_per_process, Buffer->B_cols_per_process);
79     /*std::cout << "C" << std::endl;
80     print_mat(local_C, Buffer->A_rows_per_process, Buffer->B_cols_per_process);*/
81     MPI_Gather(local_C, size_C, MPI_DOUBLE, nullptr, 0, MPI_DATATYPE_NULL, 0, MPI_COMM_WORLD);
82     delete[] local_buffer;
83     delete[] local_C;
84     local_buffer = nullptr;
85     local_C = nullptr;
86 }
```

图 10: GEMM 1

2. 分发方式 2

```

16  /* MPI集台通信矩阵乘法
17  * @param Buffer 缓冲区
18  * @param C C_{i,j}=(A_i*B_j)
19  * @param COMBINED_MATRIX 变量类型指针
20  */
21 void mpi_mmml_main_process(const BUFFER *const Buffer, Matrix &C, MPI_Datatype *COMBINED_MATRIX)
22 {
23     // 计算每个进程分配的数据大小，用于生成local_buffer
24     int size_local_A = Buffer->Get_A_size_per_process();
25     int size_local_B = Buffer->Get_B_size_per_process();
26     // 计算每个进程算出的的大小，用于生成local_C
27     int size_local_C = Buffer->A_rows_per_process * Buffer->B_cols_per_process;
28     // 计算全局合并后的comm_C的大小(行数或列数不能整除时，可能有填充)
29     int size_comm_C = Buffer->comm_size * size_local_C;
30     // 计算最终结果C的大小
31     int size_C = C.rows * C.cols;
32     // 计算全局合并后comm_C的列数，用于将comm_C移到真实的結果C中
33     int comm_C_cols = Buffer->B_cols_per_process * Buffer->B_blocks;
34
35
36     double *local_buffer = new double[size_local_A + size_local_B()];
37     double *local_C = new double[size_local_C()];
38     double *comm_C = new double[size_comm_C()];
39
40
41     // 单次散射
42     MPI_Scatter(Buffer->buffer, 1, *COMBINED_MATRIX, local_buffer, 1, *COMBINED_MATRIX, 0, MPI_COMM_WORLD);
43
44     sequential_mmml(local_buffer, local_buffer + size_local_A, local_C, Buffer->A_rows_per_process, Buffer->A_cols_per_process, Buffer->B_cols_per_process);
45
46     // 单次聚集
47     MPI_Gather(local_C, size_local_C, MPI_DOUBLE, comm_C, size_local_C, MPI_DOUBLE, 0, MPI_COMM_WORLD);
48
49     // comm_C迁移到C中
50     for (int i = 0; i < C.rows; ++i) {
51         for (int j = 0; j < C.cols; ++j) {
52             int block_row_id = i / Buffer->A_rows_per_process; //0
53             int block_col_id = j / Buffer->B_cols_per_process; //1
54             int comm_C_row_id = block_row_id * Buffer->B_blocks + block_col_id;
55             int comm_C_col_id = (i % Buffer->A_rows_per_process) * Buffer->B_cols_per_process +
56                                 j % Buffer->B_cols_per_process;
57             C.MAT[i * C.cols + j] =
58                 comm_C[comm_C_row_id * comm_C_cols + comm_C_col_id];
59         }
60     }
61     delete[] local_buffer;
62     delete[] local_C;
63     delete[] comm_C;
64     local_buffer = nullptr;
65     local_C = nullptr;
66     comm_C = nullptr;
67 }

```

图 11: GEMM 2

在此给出 50-59 行部分代码的说明:

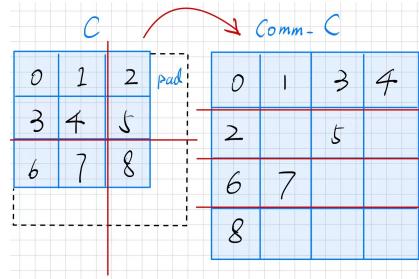


图 12: GEMM 2

```

49     // comm_C迁移到C中
50     /* C_{i,j} 被分到编号为 (A block_row_id, B block_col_id) 的块中, 如C中第2行第2列(从0计数)的8, 属于块(A1,B1)
51     * comm_C_row_id 根据 块的编号 计算 C_{i,j} 在 Comm_C 中的行编号, 如(A1,B1)在第 1+2+1=3行
52     * comm_C_col_id 计算 C_{i,j} 在 Comm_C 中的列编号, 即计算 C_{i,j} 是其所在块的第几个元素. 如C_{2,2}是第 (2%2*2+2%2)=0个元素
53     * 因此C_{2,2} = Comm_C_{3,0}
54     */
55     for (int i = 0; i < C.rows; ++i) {
56         for (int j = 0; j < C.cols; ++j) {
57             int block_row_id = i / Buffer->A_rows_per_process;
58             int block_col_id = j / Buffer->B_cols_per_process;
59             int comm_C_row_id = block_row_id * Buffer->B_blocks + block_col_id;
60             int comm_C_col_id = (i % Buffer->A_rows_per_process) * Buffer->B_cols_per_process +
61                                 j % Buffer->B_cols_per_process;
62             C.MAT[i * C.cols + j] =
63                 comm_C[comm_C_row_id * comm_C_cols + comm_C_col_id];
64         }
65     }

```

图 13

2.2.3 优化建议

1. 使用集合通信和 MPI_Type_create_struct 聚合 MPI 进程内变量后通信, 减少通信次数 (item 2).
2. 使用不同的数据分发方式, 减少通信的分发量和每个进程的数据量, 获得更好的空间局部性 (section 2.2.1).
3. 使用非阻塞的通信方式, 在通信的同时进行计算, 这样可以掩盖部分通信时长. 可以参考我的 MPI FFT 实现:

```
235 void cfft2(int n, double x[], double y[], int comm_sz, int my_rank, double sgn, double *local_x, double *local_y, double *local_temp)
236 {
237     MPI_Datatype complex_type;
238     create_complex_mpi_type(&complex_type);
239     int per_size = n / comm_sz;
240     MPI_Request send_req[per_size];
241     MPI_Request recv_req[per_size];
242     MPI_Status states[per_size];
243     int total_depth = log(n) / log(2);
244     for (int depth = total_depth - 1; depth != -1; --depth)
245     {
246         int step = pow(2, depth);
247         int cells = n / step;
248
249         // 我们先建立非阻塞通信,这样在此过程中仍可以进行计算
250         for (int work_index = my_rank * per_size; work_index < (my_rank + 1) * per_size; ++work_index)
251         {
252             // 若当前任务在上一层属于前一半
253             if (work_index % step == work_index % (2 * step))
254             {
255                 // 向后一半发送G 接收H
256                 MPI_Issend(&local_x[2 * (work_index - my_rank * per_size)], 1, complex_type,
257                             (work_index + step) / per_size, work_index,
258                             MPI_COMM_WORLD, &send_req[work_index - my_rank * per_size]);
259                 MPI_Irecv(&local_temp[2 * (work_index - my_rank * per_size)], 1, complex_type,
260                           (work_index + step) / per_size, work_index + step,
261                           MPI_COMM_WORLD, &recv_req[work_index - my_rank * per_size]);
262             }
263             else
264             {
265                 // 若在后半段,向前一半, 接受G,发送H
266                 MPI_Issend(&local_x[2 * (work_index - my_rank * per_size)], 1, complex_type,
267                             (work_index - step) / per_size, work_index,
268                             MPI_COMM_WORLD, &send_req[work_index - my_rank * per_size]);
269                 MPI_Irecv(&local_temp[2 * (work_index - my_rank * per_size)], 1, complex_type,
270                           (work_index - step) / per_size, work_index - step,
271                           MPI_COMM_WORLD, &recv_req[work_index - my_rank * per_size]);
272             }
273         }
274         // 保证通信建立完成
275         MPI_Barrier(MPI_COMM_WORLD);
276     }
277 }
```

图 14: MPI FFT

4. 精细化通信粒度: 在保持通信开销可接受的前提下, 细化通信粒度, 使数据包更小更频繁地传输. 这有助于提高网络带宽的利用率, 尤其是在网络延迟较高的情况下.
5. 利用硬件特性: 根据具体硬件环境优化 MPI 配置和通信策略. 例如, 在具有高速网络连接的集群上, 优化数据传输路径和网络拓扑利用.
6. 参数调优和性能分析: 使用工具如 MPI Profiler 或者 TAU 对应用程序进行性能分析, 找出瓶颈和优化点

下图的实验结果和上述讨论 (item 2) 基本符合.

1. 分发方式 2 要略微优于分发方式 1(数据量太小区别不大, 但是可以看出来)
2. 在大型矩阵上, 进程少时. 集合通信要远优于点对点通信, 而进程数量较多时两者性能差距不大.

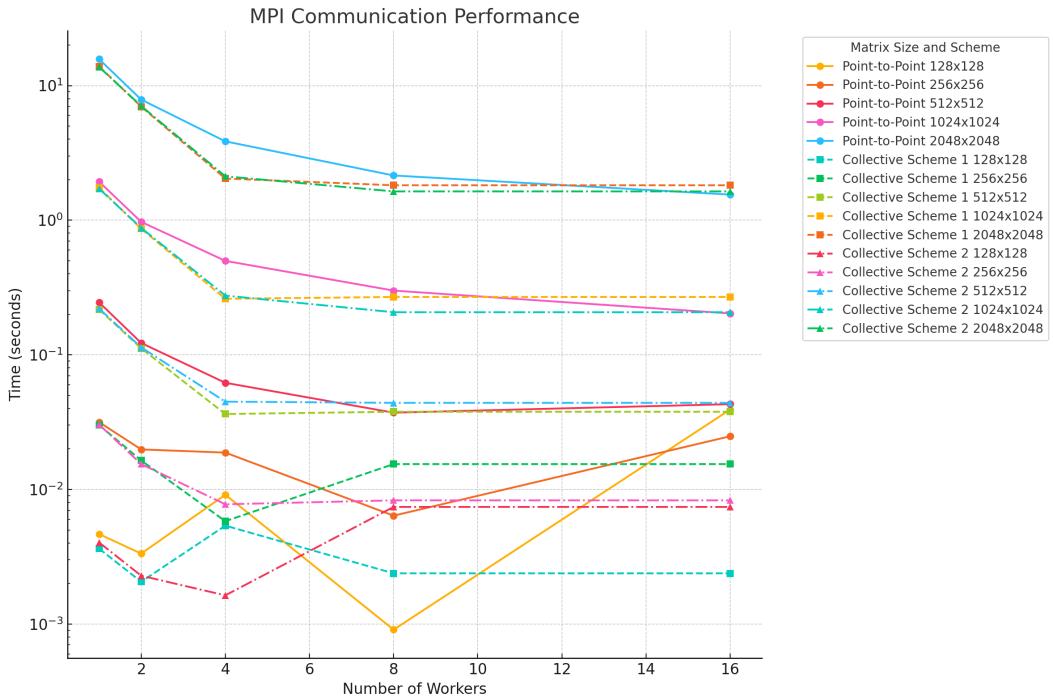


图 15: MPI GEMM

2.3 基于共享内存的 CPU 多线程编程 (OpenMP)

该部分代码可见于本人 GitHub 仓库 OpenMP GEMM.

2.3.1 分析

实现方法详见 fig. 17.

首先, 我们选择在 $i - k - j$ 的矩阵乘法基础上使用 OpenMP 进行优化, 这样可以**展开循环的前两层而不构成写竞争**. 但是这样一来, 不同线程就会写入到同一个 $C_{i,j}$ 的位置上, 造成竞态条件.

为了解决这个问题, 我们使用一个 private 的 **C_local** 来储存每个线程的计算结果, 最后再将其 atomic 地加到 **C** 上, 如果在运算过程中使用 atomic 则过于频繁, 而造成性能损失. (实测大矩阵乘法快 2 到 3 倍不等) 经实测, 这样的效果要优于展开单层循环的和使用 reduction 的.

调度模式可以选择 default/static/dynamic/guided/auto 等. 显然该任务中每个线程的任务量基本均分, 不太会存在线程空等待的问题, 因此 chunksize 基本不应有影响. 同时预先分配任务的 static 可能会略微优于动态分配任务的其他调度模式, 因为它在 runtime 中不会有额外的开销.

auto 似乎在最新版的 gcc 中仍然没有被实现 (gcc/libgomp/loop.c line196), 而是被映射到 icv->run_sched_chunk_size = 0 的 static:

```
176  bool
177  GOMP_loop_runtime_start (long start, long end, long incr,
178  ||| long *istart, long *iend)
179  {
180  struct gomp_task_icv *icv = gomp_icv (false);
181  switch (icv->run_sched_var & ~GFS_MONOTONIC)
182  {
183  case GFS_STATIC:
184  return gomp_loop_static_start (start, end, incr,
185  ||| icv->run_sched_chunk_size,
186  ||| istart, iend);
187  case GFS_DYNAMIC:
188  return gomp_loop_dynamic_start (start, end, incr,
189  ||| icv->run_sched_chunk_size,
190  ||| istart, iend);
191  case GFS_GUIDED:
192  return gomp_loop_guided_start (start, end, incr,
193  ||| icv->run_sched_chunk_size,
194  ||| istart, iend);
195  case GFS_AUTO:
196  /* For now map to schedule(static), later on we could play with feedback
driven choice. */
197  ||| return gomp_loop_static_start (start, end, incr, 0, istart, iend);
198  default:
199  ||| abort ();
200  }
201  }
202 }
```

图 16: GCC OMP auto

但是在 intel 的编译器中的 omp 已经实现了, 可以自动选择最优的调度模式.

2.3.2 实现方法

GEMM 主循环部分

调度方式 使用 46 和 47 行的 `omp_set_num_threads` 和 `omp_set_schedule` 来调整线程数量, 调度模式和 `chunk_size`. 同时使用 `schedule(runtime)` 指令使配置生效.

```
27  /**
28   * 使用 OpenMP 并行计算矩阵乘法 C_{m, p} += A_{m, n} B_{n, p}, 并允许设置调度类型和块大小
29   * @param A 指向A
30   * @param B 指向B
31   * @param C 指向C
32   * @param m
33   * @param n
34   * @param p
35   * @param num_threads 线程数
36   * @param sched_type 调度类型(omp_sched_static/omp_sched_dynamic/omp_sched_guided/omp_sched_auto)
37   * @param chunk_size chunk_size
38   *
39   * 使用每个线程局部数组 C_local 存储中间结果, 以减少对共享内存的竞争使用
40   * 最后使用原子操作+=更新全局结果矩阵 C。
41   */
42 void omp_matmul(const double *A, const double *B, double *C,
43                 int m, int n, int p,
44                 int num_threads, omp_sched_t sched_type, int chunk_size)
45 {
46     omp_set_num_threads(num_threads);
47     omp_set_schedule(sched_type, chunk_size);
48
49     #pragma omp parallel
50     {
51         std::vector<double> C_local(m * p, 0.0); // 初始化局部数组
52
53         #pragma omp for collapse(2) schedule(runtime)
54         for (int i = 0; i < m; ++i) {
55             for (int k = 0; k < n; ++k) {
56                 for (int j = 0; j < p; ++j) {
57                     C_local[i * p + j] += A[i * n + k] * B[k * p + j];
58                 }
59             }
60         }
61
62         // 使用 atomic 更新全局矩阵 C
63         for (int i = 0; i < m; ++i) {
64             for (int j = 0; j < p; ++j) {
65                 #pragma omp atomic
66                 C[i * p + j] += C_local[i * p + j];
67             }
68         }
69     }
70 }
```

图 17: OMP GEMM

2.3.3 优化建议

1. **内存访问模式:** 考虑到内存访问对性能的影响, 调整循环顺序或对输入矩阵 **A** 和 **B** 进行转置等, 使得内存访问更为连续, 从而显著提高缓存命中率, 降低延迟.
2. **使用不同的调度策略:** 在不同的任务下, 不同的调度模式可能会提供不同的性能, 在此 GEMM 任务下 `static` 调度提供了不错的性能, 但针对不同的问题规模和硬件特性, `dynamic` 或 `guided` 调度策略可能会更好, 尤其是在处理不均匀的工作负载时. 在这种情形下, `static` 可能造成线程空等待, 如下图

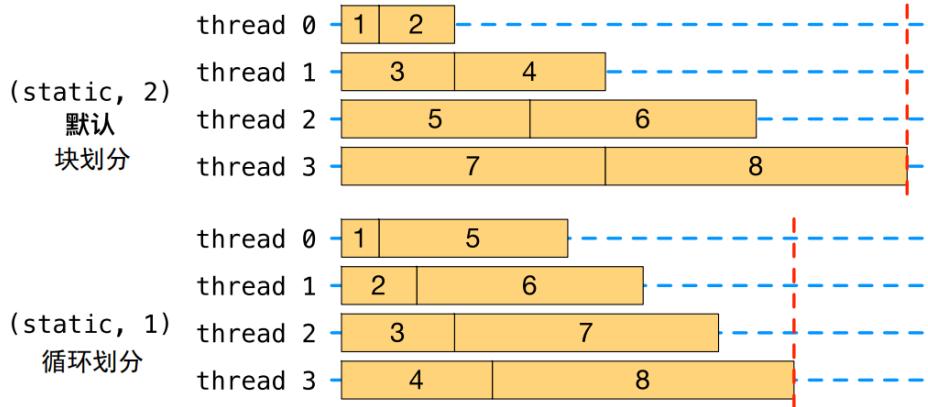


图 18: OMP static

即不同线程完成任务的时间不同,从而造成浪费. 而 dynamic 在运行中动态分配任务,guided 则可以根据任务量自动调整 chunk size(在部分实现中是一直二分).

以下部分参考了Intel® Cilk Plus White Paper

3. **细粒度控制线程亲和性:** 通过设置线程亲和性, 控制线程在特定核上运行, 以减少线程迁移和缓存失效. 在多 socket 系统上, 正确的线程亲和性设置可以避免跨 socket 的数据传输, 降低内存访问延迟.
4. **细粒度调整 chunk size:** 用算法或实验寻找最优的 chunk_size. 在 dynamic 或 guided 调度中, 适当的 chunk_size 可以平衡负载并减少调度开销. 也可以使用基于问题大小的启发式方法来选择最佳的 chunk_size.

5. 细粒度的并行策略: 考虑在更细的层面上进行并行化, 例如在内部最内层循环中引入并行.

另外, 对于 gcc 的 default 调度, 查看 gcc 源码 (GCC), 可以看到在最新的 gcc 中使用的是 dynamic 和 chunk_size=1

```

66  /* Default values of ICVs according to the OpenMP standard,
67  | except for default_device-var. */
68 const struct gomp_default_icv gomp_default_icv_values = {
69   .nthreads_var = 1,
70   .thread_limit_var = UINT_MAX,
71   .run_sched_var = GFS_DYNAMIC,
72   .run_sched_chunk_size = 1,
73   .default_device_var = INT_MIN,
74   .max_active_levels_var = 1,
75   .bind_var = omp_proc_bind_false,
76   .nteams_var = 0,
77   .teams_thread_limit_var = 0,
78   .dyn_var = false
79 };

```

图 19: GCC source code-Default

(这与 ppt 中所述不符:)

- 当迭代数多于线程数时, 需要调度线程
 - 某些线程将执行多个迭代
 - `#pragma omp parallel for schedule(type,[chunk size])`
 - type 包括 static, dynamic, guided, auto, runtime
 - 默认为 static

图 20: Silde

另外, 也可以修改 gcc/libgomp 中的调度策略, 编译出适合特定任务的 omp 调度策略, 比如修改 `omp_sched_monotonic` 的使用.

实验结果与上述讨论基本相符, 即 chunksizes 基本没有影响

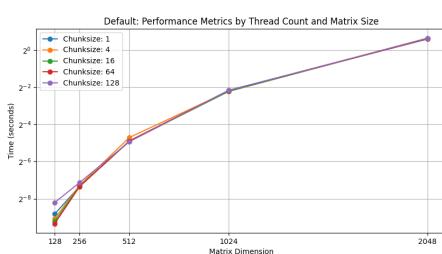


图 21: Static Chunksize-Performance

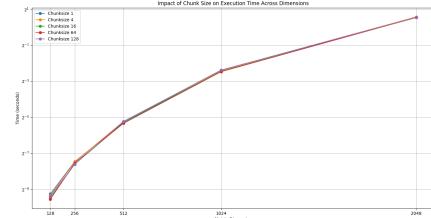


图 22: Dynamic Chunksize-Performance

而性能基本和线程数量成反比 (下面对数图中, 柱顶呈一条直线), 另外三种调度方式也基本没有影响.

与分析不同的是,static 在线程数较多时性能反而略比 dynamic 差一些, 这可能是由于实际运行环境中, 即使任务量相同, 处理器也不一定同时完成运算, 造成了空等待.

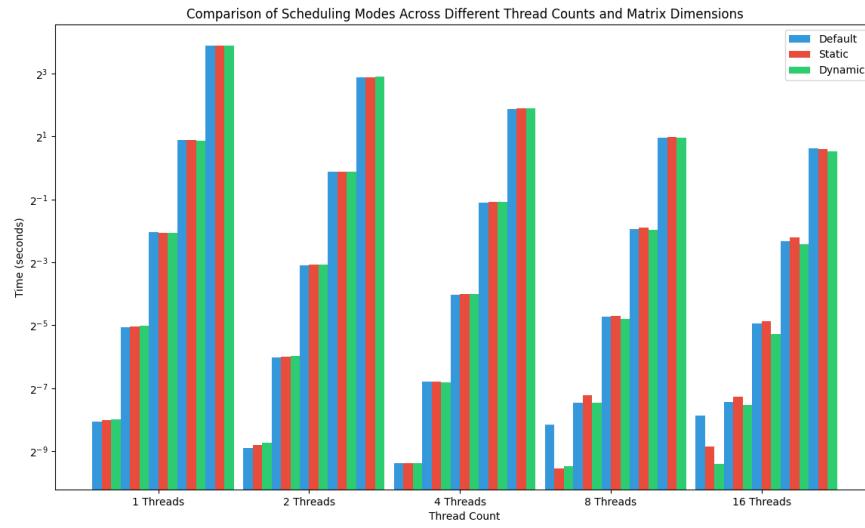


图 23: OMP GEMM Performances

(每组从左到右分别是 128/256/512/1024/2048 维度矩阵的数据)

2.4 GPU 多线程编程 CUDA

该部分代码可见于本人 GitHub 仓库 CUDA GEMM.

2.4.1 分析

Naive

$$A_{M \times N} B_{N \times K} = C_{M \times K}$$

在每个线程中计算结果矩阵 C 的一个元素 $C_{m,k} = \sum_{n=0}^{N-1} A_{m,n} B_{n,k}$

若每个 BLOCK SIZE 为 $B_X \times B_Y$, 则 GRID SIZE 为 $\lceil \frac{M}{B_X} \rceil \times \lceil \frac{K}{B_Y} \rceil$

访存分析

每个元素的计算需要读取 A 的整行和 B 的整列, 即共 $2MKN$ 次 GMEM(全局内存)读, MK 次 GMEM 写.

分块乘法

1. 先考虑 GMEM 到 SMEM(共享内存) 的分块:

将小矩阵块先存储到 SMEM 中, 而后计算时可以直接从 SMEM 中取数, 从而减少访存时延. 同时, 该算法可以显著地减少对 GMEM 的访存次数.

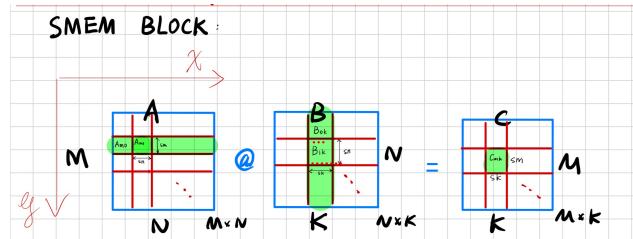


图 24: GMEM \rightarrow SMEM BLOCKING

如上图, 我们将 A 分成 $sm \times sn$ 的 tiles, B 分成 $sn \times sk$ 的 tiles, 则最后的 C 矩阵分成 $sm \times sk$ 的 tiles. 即

$$A:M \times N = (M \cdot sm) \times (N \cdot sn) \quad (7)$$

$$B:N \times K = (N \cdot sn) \times (K \cdot sk) \quad (8)$$

$$C:M \times K = (M \cdot sm) \times (K \cdot sk) \quad (9)$$

每个 BLOCK 计算 C 的一个子矩阵 $C_{m,k}$, 则共 $M \cdot K$ 个 BLOCK. 每个 BLOCK 执行以下计算:

$$C_{m,k} = \sum_{n=0}^{N-1} A_{mn} B_{nk} \quad (10)$$

其中 A_{mn} 和 B_{nk} 都是子矩阵. 显然, GRID SIZE 为 $M_- \times K_- = \frac{M}{sm} \times \frac{K}{sk}$

访存分析

每个 BLOCK 需要读取 $N_- \cdot (sm \cdot sn + sn \cdot sk)$ 次 GMEM, 并写入 GMEM.
即读取 GMEM 的次数为:

$$M_- \cdot K_- \cdot N_- \cdot (sm \cdot sn + sn \cdot sk) \quad (11)$$

$$= MNK \left(\frac{1}{sm} + \frac{1}{sk} \right) \quad (12)$$

即 sm 和 sk 越大, 对 GMEM 的访存次数越少.

2. 然后考虑 SMEM 到 Register 的分块:

同样地, 由于对 Register 的访问速度远快于 SMEM, 我们可以将 SMEM 中的数据分块到 Register 中, 从而减少访存次数和访存时延.

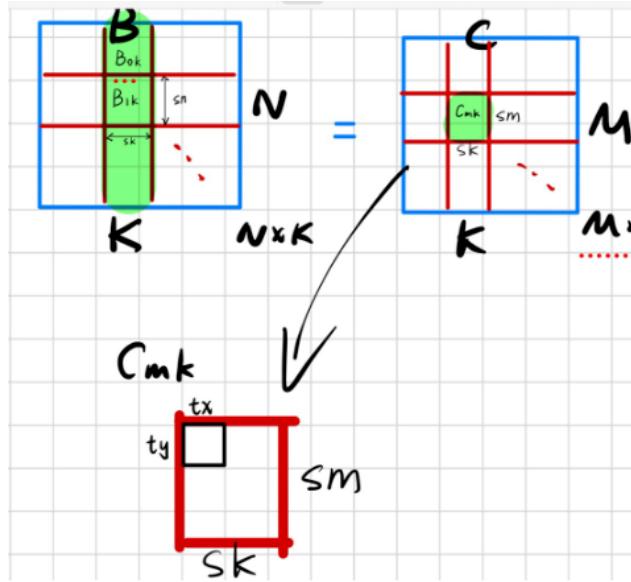


图 25: SMEM → REG BLOCKING

用以下算法每个线程中计算一个 tile 中的 $tx \times ty$ 个元素:

$$C = \sum_{i=0}^{sn-1} C_i = \sum_{i=0}^{sn-1} A_i (B_i^T)^T = \sum_{i=0}^{sn-1} \begin{bmatrix} a_{1i}b_{i1} & a_{1i}b_{i2} & \cdots & a_{1i}b_{in} \\ a_{2i}b_{i1} & a_{2i}b_{i2} & \cdots & a_{2i}b_{in} \\ \vdots & \vdots & \ddots & \vdots \\ a_{mi}b_{i1} & a_{mi}b_{i2} & \cdots & a_{mi}b_{in} \end{bmatrix} \quad (13)$$

其中 C 由 sn 个 C_i 的和组成, A_i 是 A 的第 i 列, $(B_i^T)^T$ 是 B 的第 i 行, 它们做外积.

访存分析

若不对 SMEM 分块, 则每个 BLOCK 需要负责计算一个 tile, BLOCK 中的每个线程负责计算 tile 中的单个元素. 需要读取 $2sm \cdot sk \cdot sn$ 次 SMEM.

我们对 SMEM 分块后, 每一块大小为 $tx \times ty$, 则 BLOCK 中的每个 thread 负责计算一个 tile 中的 $tx \times ty$ 个元素.

此时 BLOCK 中的线程数量 (BLOCK SIZE) 为 $\frac{sm}{ty} \times \frac{sk}{tx}$, 每个 BLOCK 需要读取

$$\frac{sm}{ty} \times \frac{sk}{tx} \cdot (tx + ty)sn \quad (14)$$

$$= sm \cdot sk \cdot sn \left(\frac{1}{tx} + \frac{1}{ty} \right) \quad (15)$$

次 SMEM.

3. 考虑上述操作中的约束条件:

显然上述的多次分块会占用存储资源, 因此必然会有一些约束条件. 以下讨论基于 Ada 架构 GPGPU.

我们安排每个 thread 计算一个 tile 中 $tx \times ty$ 个元素. 由于每个线程 BLOCK 计算 C 的一个大小为 $sm \times sk$ 的子矩阵 $C_{m,k}$, 一个 BLOCK 中的线程数量 (BLOCK SIZE) 为 $\frac{sm}{ty} \times \frac{sk}{tx}$:

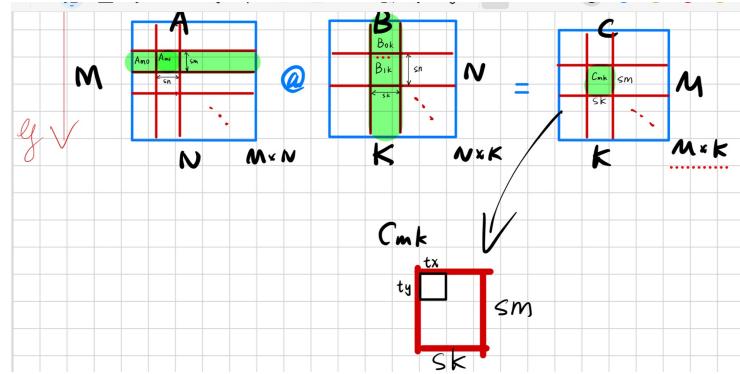


图 26: Workload per thread

然后需要考虑到几个限制 (如下图):

1. 每个 BLOCK 的线程数量有限:

$$\frac{sm}{ty} \times \frac{sk}{tx} \leq 1024 \quad (16)$$

2. 每个 BLOCK 的 SMEM 容量有限.

考虑到后面的 double buffer:

$$4 \text{ Bytes} \times (sm \cdot sn + sn \cdot sk) \leq 49152/2 \text{ Bytes} = 24576 \text{ Bytes} \quad (17)$$

3. 每个 BLOCK 的寄存器数量有限.

考慮到 Gmem to Smem 和 Smem to Reg 需要一些寄存器, 留一半的寄存器给其他用途, 则

$$sm \times sk \leq 65536/2 = 128 \times 256 \quad (18)$$

```
> ./test
Device 0: NVIDIA GeForce RTX 4070
Total number of SMs: 46
Maximum number of threads per SM: 1536
Maximum number of threads per block: 1024
Maximum size of each dimension of a block: 1024 x 1024 x 64
Maximum size of each dimension of a grid: 2147483647 x 65535 x 65535
Shared memory per block: 49152 bytes
Total global memory: 11.9937 GB
Number of registers per SM: 65536
Number of registers per block: 65536
Maximum registers per thread: 64
```

图 27: Architecture: Ada Lovelace (RTX 4070)

LDS.128 指令每次读 4 个 float32, 并且为了尽可能减少迭代次数 sn 不应太小, 故 sn 可以取 8 或 16. 再考虑到上面 sm 和 sk 越大, 对 GMEM 的访存次数越少. 并且测试矩阵中有 128×128 , 即 sm 和 sk 不能大于 128.

我们使用 Cuda_Occupancy_Calculator 计算 Occupancy 最佳的 BLOCK SIZE 和 GRID SIZE:

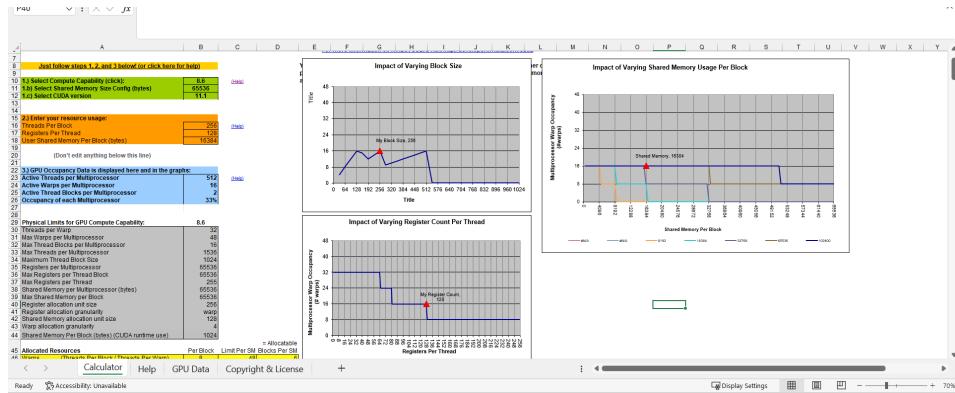


图 28: CUDA Occupancy Calculator

可以看出两组理论上可能最佳的参数 (在计算能力 8.6 上 Occupancy 为 33%, 7.5 上为 50%):

- $sm = 128, sn = 16, sk = 128, tx = 8, ty = 8$

此时 BLOCK SIZE 为 $\frac{sm}{ty} \times \frac{sk}{tx} = 16 \times 16$, GRID SIZE 为 $\frac{M}{128} \times \frac{K}{128}$, 每个线程存计算结果占用的寄存器数量为 $8 \times 8 = 64$, 每个 BLOCK 占用 16284 Bytes.

- $sm = 128, sn = 8, sk = 128, tx = 8, ty = 8$

此时 BLOCK SIZE 为 $\frac{sm}{ty} \times \frac{sk}{tx} = 16 \times 16$, GRID SIZE 为 $\frac{M}{128} \times \frac{K}{128}$, 每个线程存计算结果占用的寄存器数量为 $8 \times 8 = 64$, 每个 BLOCK 占用 8192 Bytes.

其他可取的 BLOCK SIZE(对应不同的任务划分) 在上述不等式16,17,18的限制下. 为方便我们选取正方形的 BLOCK 和 THREAD, 则其他可以取的参数为:

1. $sm = 64, sn = 8, sk = 64, tx = 8, ty = 8$

此时 BLOCK SIZE 为 $\frac{sm}{ty} \times \frac{sk}{tx} = 8 \times 8$, GRID SIZE 为 $\frac{M}{64} \times \frac{K}{64}$, 每个线程存计算结果占用的寄存器数量为 $8 \times 8 = 64$, 每个 BLOCK 占用 4096Bytes.

2. $sm = 64, sn = 16, sk = 64, tx = 8, ty = 8$

此时 BLOCK SIZE 为 $\frac{sm}{ty} \times \frac{sk}{tx} = 8 \times 8$, GRID SIZE 为 $\frac{M}{64} \times \frac{K}{64}$, 每个线程存计算结果占用的寄存器数量为 $8 \times 8 = 64$, 每个 BLOCK 占用 8192 Bytes.

向量化访存

把上述的访存改用 float4, 告诉编译器用向量化的 LDG.128 和 STG.128 指令一次读 4 个 float32. 实现可见 section 2.4.2

Bank Conflict

我们在取 A 来计算时取的是它的一列, 这样显然会有 bank conflict(A 的列数是 128 的倍数). 因此我们可以把 A 转置地存入 SMEM, 并且切块处理 BLOCK 负责的矩阵(如下图), 这样可以避免取到同一个 bank 中的元素, 从而避免 SMEM 的 bank conflict.

register 的 bank conflict 似乎要手动调寄存器映射. Nvidia 似乎没有提供 Ada 架构的 register bank 规范, 故不做处理.

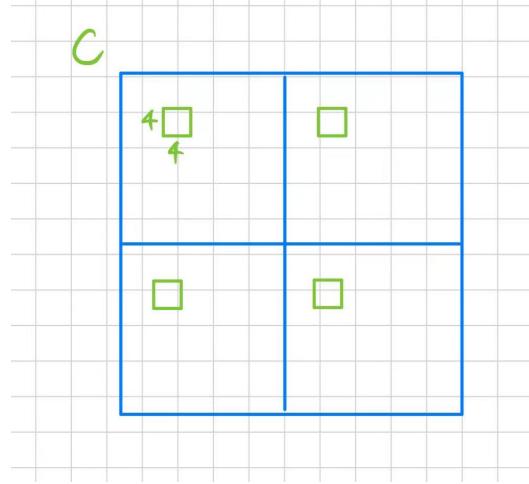


图 29: Reduce SMEM Bank Conflict

具体来说就是在 GMEM2SMEM 时, 将 A 转置后存入 SMEM, 然后在 SMEM2REG 时, 按照上图中的方式取出对应的元素, 实现见 section 2.4.2

Double Buffer

独立出第一次 GMEM2SMEM 和最后一次计算, 中间的计算和访存两两一组, 用一个 pointer 判断计算和写入的位置, 从而异步地并行起来, 相当于用 2 倍的 SMEM 来换取掩盖访问 GMEM 延迟时间的效果. 具体实习见 section 2.4.2

为什么 BLOCKSIZE 最好取 8 和 16 一个 BLOCK 负责加载 A 的 $sm * sn$ 部分到 A_SMEM, 一个 BLOCK 中的线程数量为 $BLOCKSIZE * BLOCKSIZE$. 则每个线程负责加载:

$$\frac{sm * sn}{BLOCKSIZE * BLOCKSIZE} \text{ 个 float} \quad (19)$$

我们限定每个线程处理 8*8 的 tile, 则:

$$sm = 8 \times BLOCKSIZE \quad (20)$$

而为了进行后面的矩阵乘法不产生过多 bank conflict(同时若增加算法从多行中取数, 会使得寄存器超过 128 个, 降低 occupancy), 每次取到 A_SMEM 的元素不能超过 A_SMEM 的一行, 即:

$$\frac{sm * sn}{BLOCKSIZE * BLOCKSIZE} = \frac{8sn}{BLOCKSIZE} \leq sn \quad (21)$$

从而:

$$BLOCKSIZE \geq 8 \quad (22)$$

另一方面, 打开`-ptxas-options=-v`, 可以看到每个 kernel 函数使用的寄存器数量大致为 127-128:

```
ptxas info : Function properties for _Z13blockGEMM_sn8PfS_S_iiii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 128 registers, 8192 bytes smem, 388 bytes cmem[0]
ptxas info : Compiling entry function '_Z13blockGEMM_sn8PfS_S_iiii' for 'sm_89'
ptxas info : Function properties for _Z16conflictFreeGEMMPfS_S_iiii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 128 registers, 8192 bytes smem, 388 bytes cmem[0]
ptxas info : Overriding maximum register limit 256 for '_Z16conflictFreeGEMMPfS_S_iiii' with 200 of maxrregcount option
ptxas info : 0 bytes gmem
ptxas info : Compiling entry function '_Z16conflictFreeGEMMPfS_S_iiii' for 'sm_89'
ptxas info : Function properties for _Z16conflictFreeGEMMPfS_S_iiii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 127 registers, 16384 bytes smem, 388 bytes cmem[0]
ptxas info : Overriding maximum register limit 256 for '_Z9naiveGEMMPfS_S_iiii' with 200 of maxrregcount option
ptxas info : 0 bytes gmem
ptxas info : Compiling entry function '_Z9naiveGEMMPfS_S_iiii' for 'sm_89'
```

图 30: reg

当 BLOCKSIZE 为 32 时, 每个 BLOCK 的寄存器数量为 $32 * 32 * 128 = 131072 > 65536$, 超过了寄存器数量的限制. 再考虑整除, BLOCKSIZE 最好取 8 或 16(当然其他的值也是可以的).

2.4.2 实现方法

具体的 GRID SIZE 和 BLOCK SIZE 已经在 section 2.4.1 中说明, 这里不再赘述.

Naive

```
49 // comm_C迁移到C中
50 /** C_{i,j} 被分到编号为 (A block_row_id, B block_col_id) 的块中, 如C中第2行第2列(从0计数)的8, 属于块(A1,B1)
51 * comm_C_row_id 根据 块的编号 计算 C_{i,j} 在 Comm_C 中的行编号, 如(A1,B1)在第 1*2+1=3行
52 * comm_C_col_id 计算 C_{i,j} 在 Comm_C 中的列编号, 即计算 C_{i,j} 是其所在块的第几个元素, 如C_{2,2}是第(2%2*2+2%2)=0个元素
53 * 因此C_{2,2} = Comm_C_{3,0}
54 */
55 for (int i = 0; i < C.rows; ++i) {
56     for (int j = 0; j < C.cols; ++j) {
57         int block_row_id = i / Buffer->A_rows_per_process;
58         int block_col_id = j / Buffer->B_cols_per_process;
59         int comm_C_row_id = block_row_id * Buffer->B_blocks + block_col_id;
60         int comm_C_col_id = (i % Buffer->A_rows_per_process) * Buffer->B_cols_per_process +
61                             (j % Buffer->B_cols_per_process);
62         C.MAT[i * C.cols + j] =
63             comm_C[comm_C_row_id * comm_C_cols + comm_C_col_id];
64     }
65 }
```

图 31: Naive

Block

```

2 // @include <cuda.h>
3 global__ void blockGEMM_sn8(
4     float * __restrict__ a, float * __restrict__ b, float * __restrict__ c,
5     const int M, const int N, const int K)
6 {
7     // BLOCK = sm/ty * sk/tx
8     constexpr int sm_ty = BLOCK_SIZE;
9     constexpr int sk_tx = BLOCK_SIZE;
10    // 可调的sn
11    constexpr int sn = 8;
12    // 每个线程负责计算的子矩阵大小tx*ty
13    constexpr int tx = 8;
14    constexpr int ty = 8;
15    // sm = BLOCK_SIZE * ty, sk = BLOCK_SIZE * tx
16    constexpr int sm = sm_ty * ty;
17    constexpr int sk = sk_tx * tx;
18    // 计算BLOCK的索引
19    // const int M_ = M / sm;
20    const int N_ = N / sn;
21    // const int K_ = K / sk;
22    const int B_x = blockIdx.x;
23    const int B_y = blockIdx.y;
24    const int T_x = threadIdx.x;
25    const int T_y = threadIdx.y;
26    // 计算线程的索引
27    const int thread_index = T_y * blockDim.x + T_x;
28    // SMEM
29    __shared__ float A_smem[sm][sn];
30    __shared__ float B_smem[sn][sk];
31    // Register per thread
32    float C_reg[ty][tx] = {0.0f};
33
34    // 计算从GMEM中加载到SMEM的索引
35    // 一个BLOCK加载的sm*sn, 共BLOCK_SIZE*BLOCK_SIZE个线程
36    // 每个线程负责加载的sm*sn/BLOCK_SIZE^2 = sn*ty/BLOCK_SIZE个元素
37    constexpr int read_per_thread = sn * ty / BLOCK_SIZE;
38    const int a_smem_y = thread_index * ty / BLOCK_SIZE;
39    const int a_smem_x = (thread_index * read_per_thread) % sn;
40    const int b_smem_y = thread_index * read_per_thread / sk;
41    const int b_smem_x = (thread_index * read_per_thread) % sk;
42    // 计算从GMEM加载的固定index
43    const int a_gmem_y = B_y * sm + a_smem_y;
44    const int b_gmem_x = B_x * sk + b_smem_x;
45
46    for (int n = 0; n < N_; ++n)
47    {
48        // 把数据从GMEM加载到SMEM
49        // 计算与n相关的index
50        int a_gmem_x = n * sn + a_smem_x;
51        int b_gmem_y = n * sn + b_smem_y;
52        int a_gmem_addr = OFFSET(a_gmem_y, a_gmem_x, N);
53        int b_gmem_addr = OFFSET(b_gmem_y, b_gmem_x, K);
54 #pragma unroll
55        for (int i = 0; i < read_per_thread; ++i)
56        {
57            A_smem[a_smem_y][a_smem_x + i] = a[a_gmem_addr + i];
58            B_smem[b_smem_y][b_smem_x + i] = b[b_gmem_addr + i];
59        }
60        __syncthreads();
61    }
62    // 计算C_reg
63 #pragma unroll
64    for (int tn = 0; tn < sn; ++tn)
65    {
66 #pragma unroll
67        for (int tm = 0; tm < ty; ++tm)
68        {
69 #pragma unroll
70            for (int tk = 0; tk < tx; ++tk)
71            {
72                // 每次加载同一列的第一个元素
73                int a_smem_addr = T_y * ty + tm;
74                int b_smem_addr = T_x * tx + tk;
75                C_reg[tn][tk] += A_smem[a_smem_addr][tn] * B_smem[tn][b_smem_addr];
76            }
77        }
78    }
79    __syncthreads();
80 }
81
82 // 把C_reg写回GMEM
83 for (int ri = 0; ri < ty; ++ri)
84 {
85     int C_gmem_y = B_y * sm + T_y * ty + ri;
86 #pragma unroll
87     for (int rj = 0; rj < tx; rj += 4)
88     {
89         int C_gmem_X = B_x * sk + T_x * tx + rj;
90         int C_gmem_addr = OFFSET(C_gmem_y, C_gmem_X, K);
91         C_gmem_addr += C_reg[ri][rj];
92         C_gmem_addr += C_reg[ri][rj+1];
93         C_gmem_addr += C_reg[ri][rj+2];
94         C_gmem_addr += C_reg[ri][rj+3];
95     }
96 }

```

图 32: Block Matmul

Vectorized Mem Access

```
14 #define FLOAT4(pointer) (reinterpret_cast<float4 *>(&(pointer))[0])
```

图 33: FLOAT 4 DEF

下面两张图左边为向量化访存, 右边为普通访存, 对比展示:

```
52     for (int index = 0; index < read_per_thread; index += 4)
53     {
54         FLOAT4(a_smem[a_smem_y][a_smem_x + index]) = FLOAT4(a
55         [a_gmem_addr + index]);
56         FLOAT4(b_smem[b_smem_y][b_smem_x + index]) = FLOAT4(b
57         [b_gmem_addr + index]);
58     }
59     __syncthreads();
60
61     int a_gmem_addr = offset(a_gmem_y, a_gmem_x, K);
62     #pragma unroll
63     for (int i = 0; i < read_per_thread; ++i)
64     {
65         A_smem[a_smem_y][a_smem_x + i] = a[a_gmem_addr + i];
66         B_smem[b_smem_y][b_smem_x + i] = b[b_gmem_addr + i];
67     }
68     __syncthreads();
```

图 34

```
78 // 把C_reg写回GMEM
79 for (int ri = 0; ri < ty; ++ri)
80 {
81     int C_gmem_y = B_y * sm + T_y * ty + ri;
82     #pragma unroll
83     for (int rj = 0; rj < tx; rj += 4)
84     {
85         int C_gmem_x = B_x * sk + T_x * tx + rj;
86         int C_gmem_addr = OFFSET(C_gmem_y, C_gmem_x, K);
87         FLOAT4(c[C_gmem_addr]) = FLOAT4(C.reg[ri][rj]);
88     }
89 }
90
91 // 把C_reg写回GMEM
92 for (int ri = 0; ri < ty; ++ri)
93 {
94     int C_gmem_y = B_y * sm + T_y * ty + ri;
95     #pragma unroll
96     for (int rj = 0; rj < tx; rj += 4)
97     {
98         int C_gmem_x = B_x * sk + T_x * tx + rj;
99         int C_gmem_addr = OFFSET(C_gmem_y, C_gmem_x, K);
100        c[C_gmem_addr] = C.reg[ri][rj];
101        C_gmem_addr += 1;
102        C_gmem_addr += 2;
103        C_gmem_addr += 3;
104    }
105 }
```

图 35

Bank Conflict

```
182
183     // 使用register中转掩盖GMEM latency
184     #pragma unroll
185     for (int index = 0; index < read_per_thread; index += 4)
186     {
187         FLOAT4(A_reg_load[index]) = FLOAT4(a[a_gmem_addr + index]);
188         FLOAT4(B_reg_load[index]) = FLOAT4(b[b_gmem_addr + index]);
189     }
190     // 从register中转到SMEM
191     #pragma unroll
192     for (int index = 0; index < read_per_thread; index += 4)
193     {
194         A_smem[a_smem_x + index][a_smem_y] = A_reg_load[index];
195         A_smem[a_smem_x + index + 1][a_smem_y] = A_reg_load[index + 1];
196         A_smem[a_smem_x + index + 2][a_smem_y] = A_reg_load[index + 2];
197         A_smem[a_smem_x + index + 3][a_smem_y] = A_reg_load[index + 3];
198         FLOAT4(B_smem[b_smem_y][b_smem_x + index]) = FLOAT4(B_reg_load[index]);
199     }
200     __syncthreads();
```

图 36: GMEM → SMEM

```

126 __global__ void conflictFreeGEMM_sn16(
127     for (int tn = 0; tn < sn; tn++)
128     {
129         // 从SMEM中加载到register
130         // 取出A的对应列和B的对应行
131         // 由于A在存入时转置了,所以在SMEM中都是按行取出
132         FLOAT4(A_reg_compute[0]) = FLOAT4(A_smem[tn][T_y * ty / 2]);
133         FLOAT4(A_reg_compute[4]) = FLOAT4(A_smem[tn][T_y * ty / 2 + sm / 2]);
134         FLOAT4(B_reg_compute[0]) = FLOAT4(B_smem[tn][T_x * tx / 2]);
135         FLOAT4(B_reg_compute[4]) = FLOAT4(B_smem[tn][T_x * tx / 2 + sm / 2]);
136
137 #pragma unroll
138         for (int tm = 0; tm < ty; tm++)
139         {
140 #pragma unroll
141             for (int tk = 0; tk < tx; tk++)
142             {
143                 C_reg[tm][tk] += A_reg_compute[tm] * B_reg_compute[tk];
144             }
145         }
146
147         __syncthreads();
148     }
149
150     // 写回上面两块tiles的计算结果
151 #pragma unroll
152     for (int ri = 0; ri < ty / 2; ri++)
153     {
154         int store_c_gmem_m = B_y * sm + T_y * ty / 2 + ri;
155         int store_c_gmem_n = B_x * sk + T_x * tx / 2;
156         int store_c_gmem_addr = OFFSET(store_c_gmem_m, store_c_gmem_n, N);
157         FLOAT4(c[store_c_gmem_addr]) = FLOAT4(C_reg[ri][0]);
158         FLOAT4(c[store_c_gmem_addr + sk / 2]) = FLOAT4(C_reg[ri][4]);
159     }
160
161     // 写回下面两块tiles的计算结果
162 #pragma unroll
163     for (int ri = 0; ri < ty / 2; ri++)
164     {
165         int store_c_gmem_m = B_y * sm + T_y * ty / 2 + ri + sm / 2;
166         int store_c_gmem_n = B_x * sk + T_x * tx / 2;
167         int store_c_gmem_addr = OFFSET(store_c_gmem_m, store_c_gmem_n, N);
168         FLOAT4(c[store_c_gmem_addr]) = FLOAT4(C_reg[ri + ty / 2][0]);
169         FLOAT4(c[store_c_gmem_addr + sk / 2]) = FLOAT4(C_reg[ri + ty / 2][4]);
170     }

```

图 37: Compute and SMEM → GMEM

Double Buffer

注意 GPU 不能乱序执行, 主循环中需要先将下一次循环计算需要的 Global Memory 中的数据 load 到寄存器, 然后进行本次计算, 之后再将 load 到寄存器中的数据写到 Shared Memory, 这样在 LDG 指令向 Global Memory 做 load 时, 不会影响后续 FFMA 及其它运算指令的 launch 执行.

```

29 // SMMW
30 // 注意为了减少bank conflict, 此处A按列存储, B按行存储,
31 __shared__ float A_smem[2][sn][sm];
32 __shared__ float B_smem[2][sn][sk];
33

```

图 38: Double Mem usage

```

52 {
53     int a_gmem_x = a_smem_x;
54     int b_gmem_y = b_smem_y;
55     int a_gmem_addr = OFFSET(a_gmem_y, a_gmem_x, N);
56     int b_gmem_addr = OFFSET(b_gmem_y, b_gmem_x, K);
57 #pragma unroll
58     for (int index = 0; index < read_per_thread; index += 4)
59     {
60         FLOAT4(A_reg_load[index]) = FLOAT4(a[a_gmem_addr + index]);
61         FLOAT4(B_reg_load[index]) = FLOAT4(b[b_gmem_addr + index]);
62     }
63 #pragma unroll
64     for (int index = 0; index < read_per_thread; index += 4)
65     {
66         A_smem[0][a_smem_x + index][a_smem_y] = A_reg_load[index];
67         A_smem[0][a_smem_x + index + 1][a_smem_y] = A_reg_load[index + 1];
68         A_smem[0][a_smem_x + index + 2][a_smem_y] = A_reg_load[index + 2];
69         A_smem[0][a_smem_x + index + 3][a_smem_y] = A_reg_load[index + 3];
70         FLOAT4(B_smem[0][b_smem_y][b_smem_x + index]) = FLOAT4(B_reg_load[index]);
71     }
72     __syncthreads();
73 }

```

图 39: First MEM Access

```

125
126 #pragma unroll
127     for (int tn = 0; tn < sn; tn++)
128     {
129         // 从SMEM中加载到register
130         // 取出A的对应列和B的对应行
131         // 由于A在存入时转置了,所以在SMEM中都是按行取出
132         FLOAT4(A_reg_compute[0]) = FLOAT4(A_smem[1][tn][T_y * ty / 2]);
133         FLOAT4(A_reg_compute[4]) = FLOAT4(A_smem[1][tn][T_y * ty / 2 + sm / 2]);
134         FLOAT4(B_reg_compute[0]) = FLOAT4(B_smem[1][tn][T_x * tx / 2]);
135         FLOAT4(B_reg_compute[4]) = FLOAT4(B_smem[1][tn][T_x * tx / 2 + sm / 2]);
136
137 #pragma unroll
138     for (int tm = 0; tm < ty; tm++)
139     {
140 #pragma unroll
141         for (int tk = 0; tk < tx; tk++)
142         {
143             C_reg[tm][tk] += A_reg_compute[tm] * B_reg_compute[tk];
144         }
145     }
146 }
147 // 写回上面两块tiles的计算结果

```

图 40: Last Compute

在下图中, 可以看到用 pointer 和 pointer next, 并行地完成本轮的计算和下一轮的数据准备.

即完整的 pipeline 为:([取数据₀] → [计算₀, 取数据₁] → ⋯ → [计算_{N-1}, 取数据_N] → [计算_N])

```

78     int pointer = (n - 1) % 2;
79     int pointer_next = n % 2;
80     int a_gmem_x = n * sn + a_smem_x;
81     int b_gmem_y = n * sn + b_smem_y;
82     int a_gmem_addr = OFFSET(a_gmem_y, a_gmem_x, N);
83     int b_gmem_addr = OFFSET(b_gmem_y, b_gmem_x, K);
84
85     // 使用register中转掩盖GMEM latency
86 #pragma unroll
87     for (int index = 0; index < read_per_thread; index += 4)
88     {
89         FLOAT4(A_reg_load[index]) = FLOAT4(a[a_gmem_addr + index]);
90         FLOAT4(B_reg_load[index]) = FLOAT4(b[b_gmem_addr + index]);
91     }
92     // 从register中转到SMEM
93     // 计算C_reg
94 #pragma unroll
95     for (int tn = 0; tn < sn; tn++)
96     {
97         // 从SMEM中加载到register
98         // 取出A的对应列和B的对应行
99         // 由于A在存入时转置了,所以在SMEM中都是按行取出
100        FLOAT4(A_reg_compute[0]) = FLOAT4(A_smem[pointer][tn][T_y * ty / 2]);
101        FLOAT4(A_reg_compute[4]) = FLOAT4(A_smem[pointer][tn][T_y * ty / 2 + sm / 2]);
102        FLOAT4(B_reg_compute[0]) = FLOAT4(B_smem[pointer][tn][T_x * tx / 2]);
103        FLOAT4(B_reg_compute[4]) = FLOAT4(B_smem[pointer][tn][T_x * tx / 2 + sm / 2]);
104
105 #pragma unroll
106     for (int tm = 0; tm < ty; tm++)
107     {
108 #pragma unroll
109         for (int tk = 0; tk < tx; tk++)
110         {
111             C_reg[tm][tk] += A_reg_compute[tm] * B_reg_compute[tk];
112         }
113     }
114 }
115 #pragma unroll
116     for (int index = 0; index < read_per_thread; index += 4)
117     {
118     ✦ A_smem[pointer_next][a_smem_x + index][a_smem_y] = A_reg_load[index];
119     A_smem[pointer_next][a_smem_x + index + 1][a_smem_y] = A_reg_load[index + 1];
120     A_smem[pointer_next][a_smem_x + index + 2][a_smem_y] = A_reg_load[index + 2];
121     A_smem[pointer_next][a_smem_x + index + 3][a_smem_y] = A_reg_load[index + 3];
122     FLOAT4(B_smem[pointer_next][b_smem_y][b_smem_x + index]) = FLOAT4(B_reg_load[index]);
123 }
124 __syncthreads();
125 }
```

图 41: pipeline

2.4.3 优化建议

1. GMEM 的访问延迟远远大于 SMEM, 使用共享内存减少对全局内存的访问次数, 从而提高性能.
2. 使用分块等任务划分方式, 使得每个线程负责计算一个 tile 中的多个元素, 从而提高并行度, 减少 GMEM to SMEM 和 SMEM to REGISTER 的访存次数.
3. 使用 padding/手动 map 到其他次序执行的方式规避 bank conflict.
4. 使用空间换时间, 即可以用 double buffer 等方法, 在访存的同时计算, 从而掩盖访存延迟提高性能.
5. 编译时可以开启-ptxas-options=-v 和-maxrregcount=* 等选项, 控制编译出文件的寄存器数量, 从而提高性能.
6. 分析算法是 IO 密集型还是计算密集型, 并使用 CUDA Occupancy Calculator/n-sys profilling 等工具, 找到性能瓶颈. 选择合适的 BLOCK SIZE, GRID SIZE 和任务划分方式, 从而提高性能.
7. 使用 Tensor Core 编程等硬件加速方式, 从而提高性能.

下表中的实验结果和上述理论分析 (section 2.4.1) 符合.

Matrix dims MxNxK: 2048x2048x2048			
Algorithm BLOCKSIZE=16	Time (s)	TFLOPS	绝对加速比 (%)
CUBLAS	0.001023	16.792684	100.00
naiveGEMM	0.009086	1.890818	11.26
blockGEMM_sn8	0.001568	10.956499	65.25
blockGEMM_sn16	0.001625	10.573978	62.97
vec_GEMM_sn8	0.001234	13.921037	82.90
vec_GEMM_sn16	0.001249	13.750360	81.88
conflictFreeGEMM_sn8	0.001097	15.658899	93.25
conflictFreeGEMM_sn16	0.001043	16.466607	98.06
doubleBufferGEMM_sn8	0.000980	17.538562	104.44
doubleBufferGEMM_sn16	0.000943	18.214506	108.47

表 1: Performance for matrix dimensions and block dims 16x16

3 对比分析

每种架构优化时要注意的问题均已在 section 2的优化建议中详细讨论:

串行见 (section 2.1.3), MPI 见 (section 2.2.3), OpenMP 见 (section 2.3.3),cuda 见 (section 2.4.3). 下面进一步总结矩阵乘法中不同编程框架在实现中需要注意的问题 (容易出错的地方等等):

1. 串行

- **算法选择:** 选择合适的矩阵乘法算法, 如交换循环顺序的三重循环算法就比较高效.
- **内存管理:** 确保矩阵内存分配和释放正确, 避免内存泄漏.
- **缓存优化:** 尽量提高缓存命中率, 如使用局部性较好的算法.
- **数组越界:** 确保矩阵索引不越界, 避免访问非法内存
- **初始化变量:** 在使用矩阵之前, 确保所有元素都已正确初始化.

2. MPI

- **数据分块:** 将矩阵分块, 分配到不同的进程中进行计算, 注意下标的算法.
- **通信开销:** 尽量减少通信次数, 优化通信方式, 如使用非阻塞通信.
- **负载均衡:** 确保各进程负载均衡, 避免某些进程空闲或过载.
- **集合通信顺序:** 确保所有进程在相同的集合通信调用顺序
- **Send/Recv 对齐:** 保证发送和接收的匹配, 避免死锁比如

Listing 1: MPI S/R

```
1 if (my_rank % 2 == 0){  
2     MPI_Send(local_A, local_n, MPI_INT, partner, 0, comm);  
3     MPI_Recv(temp_B, local_n, MPI_INT, partner, 0, comm, &  
4             status);  
5 } else {  
6     MPI_Recv(temp_B, local_n, MPI_INT, partner, 0, comm, &  
7             status);  
8     MPI_Send(local_A, local_n, MPI_INT, partner, 0, comm);  
9 }
```

- **内存泄漏:** 所有分配的内存都应在进程结束前释放.

3. OpenMP

- **并行化粒度:** 选择合适的并行化粒度.
- **数据竞争:** 多线程编程需要注意避免数据竞争问题, 确保共享变量的正确同步. 要使用合适的同步机制, 如临界区/原子操作/lock/barrier 等.

- **内存一致性:** 注意内存一致性问题, 确保不同线程之间的数据正确性.
- **错误的循环并行化:** 注意避免对不能独立运行的循环进行并行化, 比如有循环依赖时, 也要注意循环条件不能是 $!=$ 等判断符号.
- **注意变量作用域:** 不要超作用域使用变量
- **注意编译器是否支持 omp** 否则会直接忽略 omp 指令
- **调度策略:** 选择合适的调度策略, 避免线程空等待或任务不均衡
- **线程数:** 合理设置线程数, 避免过多线程导致的线程切换开销
- **注意并行指令开始的位置:** 在合适的位置使用 `#pragma omp parallel` 和 `#pragma omp for` 等指令

4. CUDA

- **内存管理:** 显存的分配与释放, 使用 `cudaMalloc` 和 `cudaFree` 等函数.
- **线程分配:** 合理分配线程和线程块, 考虑硬件限制和线程调度效率.
- **共享内存:** 利用共享内存提高访问速度, 避免全局内存访问的延迟.
- **内存传输:** 确保你知道数据在 device 还是 host 上, 减少主机与设备之间的数据传输次数, 优化数据传输策略.
- **避免未对齐的内存访问:** 确保内存访问对齐, 避免性能下降.
- **显存泄漏:** 所有分配的显存都应在使用完毕后释放.
- **线程同步:** 使用合适的同步机制, 如 `__syncthreads()`, 确保所有线程执行结束.
- **硬件限制:** 考虑硬件限制, 如线程块大小, 共享内存大小, 寄存器数量等