南開大學
Nankai University

计算机学院

计算机组成原理实验报告

矩阵乘法

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 5 月 29 日

目录

1 基础要求

1.1 问题重述

已知两个矩阵：矩阵 A（大小 N × M），矩阵 B（大小 M × 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 × B，并对随机生成的的双进度浮点数矩阵输入，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；

1.2 算法设计

1.2.1 标准的矩阵乘算法

```
// 基础的矩阵乘法baseline实现（使用一维数组）
void matmul_baseline(const std::vector<double>& A,
                     const std::vector<double>& B,
                     std::vector<double>& C, int N, int M, int P) {
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < P; ++j) {
            C[i * P + j] = 0;
            for (int k = 0; k < M; ++k)
                C[i * P + j] += A[i * M + k] * B[k * P + j];
        }
}
```

以上是标准的矩阵乘算法的实现，通过三重循环实现了平凡的矩阵乘法，并使用一维容器数组来进行矩阵的存储，时间复杂度为 O(NMP)。但是对于较大规模的矩阵来说，其时间复杂度会明显升高，并且由于访问顺序是 A 的行与 B 的列，在矩阵较大时可能会出现缓存不命中的问题，性能较差。

1.2.2 利用 OpenMP 进行多线程并发的编程

```
void matmul_openmp(const std::vector<double>& A,
                   const std::vector<double>& B,
                   std::vector<double>& C, int N, int M, int P) {
    std::cout << "matmul_openmp methods..." << std::endl;
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < P; ++j) {
            double sum = 0;
            for (int k = 0; k < M; ++k)
                sum += A[i * M + k] * B[k * P + j];
            C[i * P + j] = sum;
        }
}
```

```
root@worker-0:~/matrix/asignment# ./test openmp
[Baseline] Time: 21.1835 seconds
matmul_openmp methods...
[OpenMP] Valid: 1, Time: 2.19074 seconds
```

图 1.1: openmp 算法

接着我们实现了 OpenMP 算法，利用 CPU 多核架构显著提升执行效率。但是我们发现，原始版本中矩阵 B 是按列访问，由于 C++ 中矩阵是按行主序存储，这种访问方式会导致频繁的缓存不命中，影响性能。为此，我们对矩阵 B 进行了转置，使得其在乘法过程中的访问模式由列优先变为行优先（连续访问），从而显著提升缓存命中率。优化代码如下所示：

```
void matmul_openmp_optimized(const std::vector<double>& A,
                             const std::vector<double>& B,
                             std::vector<double>& C,
                             int N, int M, int P) {
    std::cout << "matmul_openmp_optimized methods..." << std::endl;

    // Step 1: Transpose B => B_T[P][M]
    std::vector<double> B_T(P * M);
    for (int i = 0; i < M; ++i)
        for (int j = 0; j < P; ++j)
            B_T[j * M + i] = B[i * P + j];

    // Step 2: Matrix multiplication using transposed B
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < P; ++j) {
            double sum = 0;
            for (int k = 0; k < M; ++k) {
                sum += A[i * M + k] * B_T[j * M + k];  // Both accesses are row-major
                    (cache-friendly)
            }
            C[i * P + j] = sum;
        }
    }
}
```

可以看到明显实现了加速，加速比为 21.39。

```
root@worker-0:~/matrix/asignment# ./test openmp
[Baseline] Time: 27.7374 seconds
matmul_openmp_optimized methods...
[OpenMP] Valid: 1, Time: 1.30548 seconds
```

图 1.2: openmp 优化

1.2.3 利用子块并行思想，进行缓存友好型的并行优化方法

```
void matmul_block_tiling(const std::vector<double>& A,
```

```
                        const std::vector<double>& B,
                        std::vector<double>& C, int N, int M, int P, int
                            block_size=64) {
    std::cout << "matmul_block_tiling methods..." << std::endl;

    #pragma omp parallel for collapse(2)
    for (int ii = 0; ii < N; ii += block_size)
        for (int jj = 0; jj < P; jj += block_size)
            for (int kk = 0; kk < M; kk += block_size)
                for (int i = ii; i < std::min(ii + block_size, N); ++i)
                    for (int j = jj; j < std::min(jj + block_size, P); ++j) {
                        double sum = 0;
                        for (int k = kk; k < std::min(kk + block_size, M); ++k)
                            sum += A[i * M + k] * B[k * P + j];
                        #pragma omp atomic
                        C[i * P + j] += sum;
                    }
}
```

```
root@worker-0:~/matrix/asignment# ./test block
[Baseline] Time: 26.4611 seconds
matmul_block_tiling methods...
[Block Parallel] Valid: 1, Time: 2.01161 seconds
```

图 1.3: block 算法

这个函数采用块 tiling（分块）技术对矩阵乘法进行优化，通过将原始的大矩阵划分为小块，改善了数据的缓存局部性。在每一小块中进行局部矩阵乘法计算，从而更好地利用 CPU 缓存，减少内存访问延迟。同时使用 OpenMP 并行加速外层两个块循环（ii 和 jj），实现多线程并发计算。

```
void matmul_block_tiling_optimized(const std::vector<double>& A,
                                   const std::vector<double>& B,
                                   std::vector<double>& C,
                                   int N, int M, int P,
                                   int block_size = 64) {
    std::cout << "matmul_block_tiling_optimized methods..." << std::endl;

    // 先转置矩阵B，变成 B_t[P][M]
    std::vector<double> B_t(P * M);
    for (int i = 0; i < M; ++i)
        for (int j = 0; j < P; ++j)
            B_t[j * M + i] = B[i * P + j];

    #pragma omp parallel for collapse(2)
    for (int ii = 0; ii < N; ii += block_size)
        for (int jj = 0; jj < P; jj += block_size) {
            for (int i = ii; i < std::min(ii + block_size, N); ++i) {
                for (int j = jj; j < std::min(jj + block_size, P); ++j) {
                    double sum = 0;
```

```
                    for (int k = 0; k < M; ++k) {
                        sum += A[i * M + k] * B_t[j * M + k];   // B访问变成连续访问
                    }
                    C[i * P + j] += sum;
                }
            }
        }
}
```

同样的，我们这里也预先对矩阵 B 进行转置，使得矩阵乘法中对 B 的访问从原来的“按列访问”变为“按行访问”，大大提高了数据的缓存命中率。可以看到，优化后的加速比为 21.40。

```
root@worker-0:~/matrix/asignment# ./test block
[Baseline] Time: 27.7302 seconds
matmul_block_tiling_optimized methods...
[Block Parallel] Valid: 1, Time: 1.29552 seconds
```

图 1.4: block 优化

1.2.4 利用 MPI 消息传递，实现多进程并行优化

```
void matmul_mpi(int N, int M, int P) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int rows_per_proc = N / size;
    std::vector<double> A_local(rows_per_proc * M);
    std::vector<double> B(M * P);
    std::vector<double> C_local(rows_per_proc * P, 0);
    std::vector<double> A, C;

    if (rank == 0) {
        A.resize(N * M);
        C.resize(N * P);
        init_matrix(A, N, M);
        init_matrix(B, M, P);
    }

    MPI_Bcast(B.data(), M * P, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(A.data(), rows_per_proc * M, MPI_DOUBLE, A_local.data(),
        rows_per_proc * M, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        // 计时开始
    double start_time = MPI_Wtime();

    for (int i = 0; i < rows_per_proc; ++i)
        for (int j = 0; j < P; ++j) {
            for (int k = 0; k < M; ++k)
                C_local[i * P + j] += A_local[i * M + k] * B[k * P + j];
```

```
        }

    double local_elapsed = MPI_Wtime() - start_time;

    MPI_Gather(C_local.data(), rows_per_proc * P, MPI_DOUBLE, C.data(), rows_per_proc
        * P, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // 获取所有进程中最大的运行时间作为整个MPI计算的耗时
    double max_elapsed;
    MPI_Reduce(&local_elapsed, &max_elapsed, 1, MPI_DOUBLE, MPI_MAX, 0,
        MPI_COMM_WORLD);

    if (rank == 0) {
        std::vector<double> C_ref(N * P);
        matmul_baseline(A, B, C_ref, N, M, P);
        std::cout << "[MPI] Valid: " << validate(C, C_ref, N, P)
                  << ", Time: " << max_elapsed << " seconds\n";
    }
}
```

```
root@worker-0:~/matrix/asignment# mpirun --allow-run-as-root -np 4 ./test mpi
[MPI] Valid: 1, Time: 7.99416 seconds
```

图 1.5: mpi 算法

在这里，我们通过使用 MPI 实现了一个基于行划分的分布式矩阵乘法算法。具体做法是将输入矩阵 A 按行划分给多个进程（每个进程处理 N / size 行），而矩阵 B 由于在所有计算中都需要，采用广播（MPI_Bcast）的方式发给每个进程。每个进程只计算属于自己负责的那部分结果矩阵 C 的行（C_local），最后通过 MPI_Gather 收集结果。

为了避免不一致的本地运行时间对总性能分析造成影响，我们使用 MPI_Reduce 获取了所有进程中的最大运行时间作为全局耗时。

```
void matmul_mpi_optimized(int N, int M, int P) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int rows_per_proc = N / size;
    std::vector<double> A_local(rows_per_proc * M);
    std::vector<double> B(M * P);              // 原始B矩阵，广播用
    std::vector<double> B_T(P * M);            // 转置后的B矩阵，用于计算
    std::vector<double> C_local(rows_per_proc * P, 0);
    std::vector<double> A, C;

    if (rank == 0) {
        A.resize(N * M);
        C.resize(N * P);
        init_matrix(A, N, M);
        init_matrix(B, M, P);
```

```
18
19          // 对B进行转置，方便缓存访问
20          for (int m = 0; m < M; ++m) {
21              for (int p = 0; p < P; ++p) {
22                  B_T[p * M + m] = B[m * P + p];
23              }
24          }
25      }
26      // 广播原始的B矩阵，方便后面验证（验证时用）
27      MPI_Bcast(B.data(), M * P, MPI_DOUBLE, 0, MPI_COMM_WORLD);
28
29      // 广播转置后的B矩阵，计算时用
30      MPI_Bcast(B_T.data(), M * P, MPI_DOUBLE, 0, MPI_COMM_WORLD);
31
32      MPI_Scatter(A.data(), rows_per_proc * M, MPI_DOUBLE, A_local.data(),
            rows_per_proc * M, MPI_DOUBLE, 0, MPI_COMM_WORLD);
33
34      double start_time = MPI_Wtime();
35
36      // 计算 C_local = A_local * B_T^T = A_local * B
37      for (int i = 0; i < rows_per_proc; ++i) {
38          for (int j = 0; j < P; ++j) {
39              double sum = 0;
40              for (int k = 0; k < M; ++k) {
41                  sum += A_local[i * M + k] * B_T[j * M + k];   //
                        访问转置后的B，连续访问缓存友好
42              }
43              C_local[i * P + j] = sum;
44          }
45      }
46      double local_elapsed = MPI_Wtime() - start_time;
47
48      MPI_Gather(C_local.data(), rows_per_proc * P, MPI_DOUBLE, C.data(), rows_per_proc
            * P, MPI_DOUBLE, 0, MPI_COMM_WORLD);
49
50      double max_elapsed;
51      MPI_Reduce(&local_elapsed, &max_elapsed, 1, MPI_DOUBLE, MPI_MAX, 0,
            MPI_COMM_WORLD);
52
53      if (rank == 0) {
54          std::vector<double> C_ref(N * P);
55          matmul_baseline(A, B, C_ref, N, M, P);   // 这里用原始B做验证
56          std::cout << "[MPI] Valid: " << validate(C, C_ref, N, P)
57                    << ", Time: " << max_elapsed << " seconds\n";
58      }
59  }
```

这里也同样对 cache 进行了优化，将 B 矩阵进行了转置，由按列访问改为按行访问，提高缓存命中率。如果平凡算法的时间大概在 26s 的话，可以看到优化到了 2.2222s，加速比约为 11.70。

```
root@worker-0:~/matrix/asignment# mpirun --allow-run-as-root -np 4 ./test mpi
[MPI] Valid: 1, Time: 2.2222 seconds
```

图 1.6: MPI 优化

1.2.5 利用 SIMD 指令优化

```
void matmul_simd(const std::vector<double>& A,
                 const std::vector<double>& B,
                 std::vector<double>& C,
                 int N, int M, int P) {
    std::cout << "matmul_simd_transpose methods..." << std::endl;

    // 先转置 B 为 B_T, 使其按行访问连续
    std::vector<double> B_T(P * M);
    for (int i = 0; i < M; ++i)
        for (int j = 0; j < P; ++j)
            B_T[j * M + i] = B[i * P + j];

    // 主乘法逻辑, SIMD 向量化
    #pragma omp parallel for schedule(static)
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < P; ++j) {
            double sum = 0.0;

            // 向量化核心内层循环
            #pragma omp simd reduction(+:sum)
            for (int k = 0; k < M; ++k) {
                sum += A[i * M + k] * B_T[j * M + k];
            }

            C[i * P + j] = sum;
        }
    }
}
```

这段代码通过先转置矩阵 B,使得内存访问更加连续,提高缓存命中率;然后利用 OpenMP 的 #pragma omp simd 指令对内层循环进行自动向量化，充分利用 CPU 的 SIMD 指令集，实现数据的并行处理。这样我们就实现了 SIMD 优化，可以看到显著提高了效率，加速比约为 21.39。

```
root@worker-0:~/matrix/asignment# ./test simd
[Baseline] Time: 25.8131 seconds
matmul_simd_transpose methods...
[Simd] Valid: 1, Time: 1.20674 seconds
```

图 1.7: SIMD 优化

1.2.6 利用 DCU 高性能异构加速器

```
__global__ void matmul_kernel(const double* A, const double* B, double* C, int n, int
    m, int p) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < n && col < p) {
        double sum = 0.0;
        for (int k = 0; k < m; ++k)
            sum += A[row * m + k] * B[k * p + col];
        C[row * p + col] = sum;
    }
}

double matmul_dcu(const std::vector<double>& A, const std::vector<double>& B,
    std::vector<double>& C) {
    double *d_A, *d_B, *d_C;
    hipMalloc(&d_A, sizeof(double) * N * M);
    hipMalloc(&d_B, sizeof(double) * M * P);
    hipMalloc(&d_C, sizeof(double) * N * P);

    hipMemcpy(d_A, A.data(), sizeof(double) * N * M, hipMemcpyHostToDevice);
    hipMemcpy(d_B, B.data(), sizeof(double) * M * P, hipMemcpyHostToDevice);

    dim3 blockDim(16, 16);
    dim3 gridDim((P + blockDim.x - 1) / blockDim.x,
                 (N + blockDim.y - 1) / blockDim.y);

    hipEvent_t start, stop;
    hipEventCreate(&start);
    hipEventCreate(&stop);
    hipEventRecord(start);

    matmul_kernel<<<gridDim, blockDim>>>(d_A, d_B, d_C, N, M, P);

    hipEventRecord(stop);
    hipEventSynchronize(stop);
    float milliseconds = 0;
    hipEventElapsedTime(&milliseconds, start, stop);

    hipMemcpy(C.data(), d_C, sizeof(double) * N * P, hipMemcpyDeviceToHost);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    return static_cast<double>(milliseconds);  // 毫秒
}
```

在这一部分，我们利用 DCU 并行计算矩阵乘法，通过把矩阵 A 和矩阵 B 的数据先从 CPU 内存复制到 DCU 显存，然后启动大量线程同时计算结果矩阵 C 中的元素。每个 DCU 线程负责计算结果矩阵中一个具体的位置，通过循环累加对应行和列元素的乘积完成这一计算。通过这种设计，DCU 的并行架构让这种计算可以多线程同时进行，极大提升了速度。计算结束后，结果矩阵 C 的数据从 DCU 显存复制回 CPU 内存供后续使用。虽然在数据传输方面存在一定开销，但对于大规模矩阵运算来说，DCU 的并行计算优势远大于传输时间，从而整体提高了计算效率。但现在是从全局内存读取数据，还不能充分发挥 DCU 的优势，以下我们采用共享内存进行优化：

```
#define TILE_SIZE 16

__global__ void matmul_kernel_shared(const double* A, const double* B, double* C, int
    n, int m, int p) {
    __shared__ double tile_A[TILE_SIZE][TILE_SIZE];
    __shared__ double tile_B[TILE_SIZE][TILE_SIZE];

    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;

    double sum = 0.0;

    for (int t = 0; t < (m + TILE_SIZE - 1) / TILE_SIZE; ++t) {
        // 从全局内存加载子块 A 和 B 到共享内存（边界检查）
        if (row < n && t * TILE_SIZE + threadIdx.x < m)
            tile_A[threadIdx.y][threadIdx.x] = A[row * m + t * TILE_SIZE +
                threadIdx.x];
        else
            tile_A[threadIdx.y][threadIdx.x] = 0.0;

        if (t * TILE_SIZE + threadIdx.y < m && col < p)
            tile_B[threadIdx.y][threadIdx.x] = B[(t * TILE_SIZE + threadIdx.y) * p +
                col];
        else
            tile_B[threadIdx.y][threadIdx.x] = 0.0;

        __syncthreads();  // 同步线程块内所有线程

        for (int k = 0; k < TILE_SIZE; ++k)
            sum += tile_A[threadIdx.y][k] * tile_B[k][threadIdx.x];

        __syncthreads();  // 准备下一轮 tile 加载
    }

    if (row < n && col < p)
        C[row * p + col] = sum;
}
```

具体来说，它将矩阵 A 和矩阵 B 划分成多个小的子块（tile），每个线程块中的线程先把对应的子块数据从较慢的全局内存加载到速度更快的共享内存中。然后各线程在共享内存中完成子块乘法计算。每

次计算一个子块后，线程同步（___syncthreads()）确保所有线程都完成了数据加载和计算，避免数据冲突。由于共享内存的访问速度远快于全局内存，这种方式显著减少了内存访问延迟，提高了数据复用率。

运行结果如下所示：

```
[CPU] Time: 10388.9 ms
[DCU] Time: 3.29627 ms
[HIP] Valid: 1
[Speedup] CPU/DCU: 3151.7x
HIP_PROF:process end run totol cost:11(s)
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------
HIP PROF:hip API statistics
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------
|Name                                                |Calls          |TotalDurationNs            |AverageNs
|Percentage                  |
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------
|hipMalloc                                           |3              |387171695                  |129057231
|97.1515755497888            |
|hipMemcpy                                           |3              |7926613                    |2642204
|1.98899597173145            |
|hipEventSynchronize                                 |1              |2687611                    |2687611
|0.674392385824959           |
|hipLaunchKernel                                     |1              |596441                     |596441
|0.149662755880157           |
|hipFree                                             |3              |98821                      |32940
|0.0247967916337626          |
|hipEventRecord                                      |2              |21140                      |10570
|0.00530458278238169         |
|hipEventCreate                                      |2              |9620                       |4810
|0.00241391137022289         |
|hipEventElapsedTime                                 |1              |4610                       |4610
|0.00115677041753924         |
|__hipPopCallConfiguration                           |1              |3530                       |3530
|0.000885769972649355        |
|__hipPushCallConfiguration                          |1              |3250                       |3250
|0.000815510598048273        |
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------
|Total                                               |18             |398523331                  |22140185
|100.0                       |
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------
HIP PROF:hip kernel statistics
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------
|Name                                                |Pars                       |Calls          |TotalDurationNs            |AverageNs
            |Percentage      |
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------
|matmul_kernel_shared(double const*, double...|(1,64,32),(1,16,16)          |1              |2691031                    |2691031
            |100.0           |
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------
|Total                                               |                           |1              |2691031                    |2691031
```

图 1.8: DCU 结果

从总的计算时间来看，基于 cpu 完成的普通矩阵乘法运算运行时间为 10388.9ms，利用 DCU 高性能异构加速器完成的矩阵乘法的运行时间为 3.29627ms(测的是计算时间，不包括数据交换的消耗)，最后得到的加速比为 3151.7，可以看到利用 DCU 高性能异构加速器完成的矩阵乘法提升非常明显。

另外我们使用 hipprof 工具采集了详细的 API 调用耗时数据和核函数运行统计，分析如下：

我们可以发现，在所有 HIP API 调用里，hipMalloc 最耗时间，占了整个执行时间的 97%。这说明在运行时分配 DCU 内存是一个开销非常大的操作，如果我们在实际项目里反复执行矩阵乘法，最好是提前分配好显存，循环复用，这样可以省下很多时间。

hipMemcpy 只占了不到 2%，说明数据从主机拷贝到 DCU 的过程不是程序的瓶颈，内存带宽还够用。

而我们所关心的真正用于计算的核函数 matmul_kernel_shared 只用了 2.69 ms，说明在 DCU 上进行计算过程本身非常高效，虽然 hipMalloc 比较耗时间，但这主要是在初始化过程，正式计算时几乎都是在飞速运行，随着数据规模的不断增大，利用 DCU 进行矩阵运算的优势只会更加明显。

2 进阶题 1：基于矩阵乘法的多层感知机实现和性能优化挑战

2.1 问题重述

基于矩阵乘法，实现 MLP 神经网络计算，可进行前向传播、批处理，要求使用 DCU 加速卡，以及矩阵乘法优化方法，并进行全面评测，输入、权重矩阵由随机生成的双精度浮点数组成：

输入层：一个大小为 B × I 的随机输入矩阵（B 是 batch size=1024，I 是输入维度 =10）；

隐藏层：I × H 的权重矩阵 W1 + bias b1，激活函数为 ReLU（H 为隐含层神经元数量 =20）；

输出层：H × O 的权重矩阵 W2 + bias b2，无激活函数，（O 为输出层神经元数量 =5）；

2.2 整体结构

```
这是一个用 HIP 实现的简单多层感知机（MLP）前向传播程序，计算流程是：

*输入层维度 I，隐藏层维度 H，输出层维度 O，批量大小 BATCH。

*隐藏层计算：H = ReLU(X * W1 + B1)

*输出层计算：Y = H * W2 + B2

*计算使用 DCU 加速（基于矩阵乘法核函数和一些辅助核函数）

*同时实现了对应 CPU 版本作结果对比

*测量 CPU 和 DCU 执行时间和加速比，计算 GFLOPS。
```

2.3 算法设计

```
#include <hip/hip_runtime.h>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <cmath>
#include <chrono>

// 增大计算规模以更好体现GPU优势
#define BATCH 1024  // 增大batch size
#define I 10        // 增大输入维度
#define H 20        // 增大隐藏层维度
#define O 5         // 增大输出维度
```

首先在这一部分进行网络结构与数据准备，输入层大小为 10，隐藏层大小为 20，输出层大小为 5，输入样本批次为 1024，代表我们一次性处理 1024 个输入样本。

```
// 优化的矩阵乘法核函数（使用共享内存）
__global__ void matmul_kernel_optimized(const double* A, const double* B, double* C,
    int M, int N, int K) {
```

```
    __shared__ double As[16][16];
    __shared__ double Bs[16][16];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    double sum = 0.0;

    for (int tile = 0; tile < (K + blockDim.x - 1) / blockDim.x; ++tile) {
        // 加载数据到共享内存
        int k = tile * blockDim.x + threadIdx.x;
        if (row < M && k < K)
            As[threadIdx.y][threadIdx.x] = A[row * K + k];
        else
            As[threadIdx.y][threadIdx.x] = 0.0;

        k = tile * blockDim.y + threadIdx.y;
        if (k < K && col < N)
            Bs[threadIdx.y][threadIdx.x] = B[k * N + col];
        else
            Bs[threadIdx.y][threadIdx.x] = 0.0;

        __syncthreads();

        // 计算部分乘积
        for (int k = 0; k < blockDim.x; ++k) {
            sum += As[threadIdx.y][k] * Bs[k][threadIdx.x];
        }

        __syncthreads();
    }

    if (row < M && col < N) {
        C[row * N + col] = sum;
    }
}
```

在我们的核函数中，也同样使用到了共享内存利用来提升访问速度，减少全局内存访问。核函数的功能就是计算 C = A * B，核心操作就是先定义相关变量，再进行分块，利用共享内存提高访问速度，把 A、B 子矩阵块加载到共享内存。等待所有线程完成加载之后，我们开始进行部分乘积的计算，最后等待完所有线程完成计算之后将计算结果写回全局内存。

```
__global__ void relu_kernel(double* A, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size)
        A[idx] = fmax(0.0, A[idx]);
}
```

在这里我们实现了激活函数 ReLU 核函数，逻辑很简单，就是对输入数组 A 的每个元素应用 ReLU 激

活：fmax(0, A[idx]) 进行映射引入非线性，每个线程处理一个元素。

```
__global__ void add_bias_kernel(double* A, const double* bias, int rows, int cols) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < rows * cols) {
        int col = idx % cols;
        A[idx] += bias[col];
    }
}
```

在神经网络中，除了输入乘以权重外，还会加上一个偏置项。在这里添加了加偏置核函数，为二维矩阵 A 按列添加偏置向量 bias。

```
void random_init(std::vector<double>& mat) {
    for (auto& val : mat)
        val = static_cast<double>(rand()) / RAND_MAX * 2 - 1;
}
```

这里定义了 CPU 随机初始化函数，对向量中所有元素随机初始化到 [-1, 1] 区间，模拟随机权重和输入。这是神经网络中常见的初始化方式，从而保证网络可以正常学习。

```
// CPU baseline
void mlp_forward_cpu(const std::vector<double>& X, const std::vector<double>& W1,
                     const std::vector<double>& B1, const std::vector<double>& W2,
                     const std::vector<double>& B2, std::vector<double>& H_layer,
                     std::vector<double>& Y) {
    // 隐藏层计算
    for (int b = 0; b < BATCH; ++b) {
        for (int h = 0; h < H; ++h) {
            double sum = B1[h];
            for (int i = 0; i < I; ++i) {
                sum += X[b * I + i] * W1[i * H + h];
            }
            H_layer[b * H + h] = fmax(0.0, sum); // ReLU
        }
    }

    // 输出层计算
    for (int b = 0; b < BATCH; ++b) {
        for (int o = 0; o < O; ++o) {
            double sum = B2[o];
            for (int h = 0; h < H; ++h) {
                sum += H_layer[b * H + h] * W2[h * O + o];
            }
            Y[b * O + o] = sum;
        }
    }
}
```

在这里实现了 CPU 版本的 MLP 前向计算，逐元素计算隐藏层和输出层，遍历批量内样本，计算每个神经元的加权和加偏置，激活函数用 ReLU。这个 CPU 版本用于和 DCU 版本进行比较，以确定 DCU 正确性和优化效能。

```
int main() {
    std::vector<double> h_X(BATCH * I), h_W1(I * H), h_B1(H), h_W2(H * O), h_B2(O);
    std::vector<double> h_H(BATCH * H), h_Y(BATCH * O), h_Y_cpu(BATCH * O);

    random_init(h_X);
    random_init(h_W1);
    random_init(h_B1);
    random_init(h_W2);
    random_init(h_B2);

    std::cout << "计算规模: BATCH=" << BATCH << ", I=" << I << ", H=" << H << ", O="
        << O << std::endl;

    // CPU baseline - 多次运行取平均
    double cpu_time_total = 0.0;
    const int cpu_runs = 5;
    for (int run = 0; run < cpu_runs; ++run) {
        auto t0 = std::chrono::high_resolution_clock::now();
        mlp_forward_cpu(h_X, h_W1, h_B1, h_W2, h_B2, h_H, h_Y_cpu);
        auto t1 = std::chrono::high_resolution_clock::now();
        cpu_time_total += std::chrono::duration<double, std::milli>(t1 - t0).count();
    }
    double cpu_time = cpu_time_total / cpu_runs;
    std::cout << "[CPU时间] " << cpu_time << " ms (" << cpu_runs << "次平均)" <<
        std::endl;

    // GPU计算
    double *d_X, *d_W1, *d_B1, *d_H, *d_W2, *d_B2, *d_Y;

    // 分配设备内存
    hipMalloc(&d_X, BATCH * I * sizeof(double));
    hipMalloc(&d_W1, I * H * sizeof(double));
    hipMalloc(&d_B1, H * sizeof(double));
    hipMalloc(&d_H, BATCH * H * sizeof(double));
    hipMalloc(&d_W2, H * O * sizeof(double));
    hipMalloc(&d_B2, O * sizeof(double));
    hipMalloc(&d_Y, BATCH * O * sizeof(double));

    // 数据传输到GPU
    hipMemcpy(d_X, h_X.data(), BATCH * I * sizeof(double), hipMemcpyHostToDevice);
    hipMemcpy(d_W1, h_W1.data(), I * H * sizeof(double), hipMemcpyHostToDevice);
    hipMemcpy(d_B1, h_B1.data(), H * sizeof(double), hipMemcpyHostToDevice);
    hipMemcpy(d_W2, h_W2.data(), H * O * sizeof(double), hipMemcpyHostToDevice);
    hipMemcpy(d_B2, h_B2.data(), O * sizeof(double), hipMemcpyHostToDevice);

```

```
    // 优化的线程配置
    dim3 block(16, 16);
    dim3 grid1((H + block.x - 1) / block.x, (BATCH + block.y - 1) / block.y);
    dim3 grid2((O + block.x - 1) / block.x, (BATCH + block.y - 1) / block.y);

    int threads_per_block = 256;
    int blocks_h = (BATCH * H + threads_per_block - 1) / threads_per_block;
    int blocks_o = (BATCH * O + threads_per_block - 1) / threads_per_block;

    // GPU预热
    matmul_kernel_optimized<<<grid1, block>>>(d_X, d_W1, d_H, BATCH, H, I);
    hipDeviceSynchronize();

    // 精确计时GPU计算（不包含数据传输）
    hipEvent_t start, stop;
    hipEventCreate(&start);
    hipEventCreate(&stop);

    const int gpu_runs = 10;
    float gpu_time_total = 0.0;

    for (int run = 0; run < gpu_runs; ++run) {
        hipEventRecord(start);

        // 隐藏层: H = X * W1 + B1, ReLU
        matmul_kernel_optimized<<<grid1, block>>>(d_X, d_W1, d_H, BATCH, H, I);
        add_bias_kernel<<<blocks_h, threads_per_block>>>(d_H, d_B1, BATCH, H);
        relu_kernel<<<blocks_h, threads_per_block>>>(d_H, BATCH * H);

        // 输出层: Y = H * W2 + B2
        matmul_kernel_optimized<<<grid2, block>>>(d_H, d_W2, d_Y, BATCH, O, H);
        add_bias_kernel<<<blocks_o, threads_per_block>>>(d_Y, d_B2, BATCH, O);

        hipEventRecord(stop);
        hipEventSynchronize(stop);

        float run_time;
        hipEventElapsedTime(&run_time, start, stop);
        gpu_time_total += run_time;
    }

    double gpu_time = gpu_time_total / gpu_runs;
    std::cout << "[GPU时间] " << gpu_time << " ms (" << gpu_runs << "次平均, 纯计算)"
        << std::endl;

    // 数据传输回CPU
    hipMemcpy(h_Y.data(), d_Y, BATCH * O * sizeof(double), hipMemcpyDeviceToHost);

    // 验证结果
```

```
    bool valid = true;
    double max_diff = 0.0;
    for (int i = 0; i < BATCH * O; ++i) {
        double diff = std::abs(h_Y[i] - h_Y_cpu[i]);
        max_diff = std::max(max_diff, diff);
        if (diff > 1e-6) {
            valid = false;
        }
    }

    std::cout << "[验证结果] " << (valid ? "通过" : "失败") << std::endl;
    std::cout << "[最大误差] " << max_diff << std::endl;
    std::cout << "[加速比] " << cpu_time / gpu_time << "x" << std::endl;

    // 计算GFLOPS
    double flops = 2.0 * BATCH * (I * H + H * O); // 矩阵乘法的浮点运算数
    std::cout << "[CPU GFLOPS] " << flops / (cpu_time * 1e6) << std::endl;
    std::cout << "[GPU GFLOPS] " << flops / (gpu_time * 1e6) << std::endl;

    // 清理资源
    hipEventDestroy(start);
    hipEventDestroy(stop);
    hipFree(d_X);
    hipFree(d_W1);
    hipFree(d_B1);
    hipFree(d_H);
    hipFree(d_W2);
    hipFree(d_B2);
    hipFree(d_Y);

    return 0;
}
```

最后在我们的 main 函数中,首先会创建矩阵向量,为神经网络分配的主机端向量,用容器用 std::vector<double> 进行管理。接着会通过调用我们上面实现的 random_init 函数来给输入数据和网络的参数(权重、偏置等)初始化随机值,范围在 [-1, 1] 之间。它们会被用来执行前向传播。然后会执行 CPU 前向传播,运行 5 次取平均时间,作为 DCU 的参考。接着会通过 hipMalloc 在 DCU 上分配内存,并利用 hipMemcpy 把数据从主机拷贝到 DCU 上。再设置设置矩阵乘法核函数线程块和网格尺寸以及线程块数量用于偏置和 ReLU 核函数,并让 DCU 预热调用一次矩阵乘法核函数,防止第一次调用延迟影响计时。之后进行计时并多次执行 DCU 前向传播,计算平均时间。在 main 函数的最后,我们进行将 DCU 结果拷回 CPU、验证 DCU 计算结果与 CPU 结果的差异、计算并输出理论 GFLOPS 性能指标、释放资源等操作。

至此我们完成了基于矩阵乘法的多层感知机实现和性能优化挑战的算法设计层面。

2.4 结果分析

```
计算规模： BATCH=1024, I=10, H=20, O=5
[CPU时间] 0.14528 ms (5次平均)
[GPU时间] 0.109514 ms (10次平均，纯计算)
[验证结果] 通过
[最大误差] 1.77636e-15
[加速比] 1.32659x
[CPU GFLOPS] 4.22907
[GPU GFLOPS] 5.61026
HIP_PROF:process end run totol cost:1(s)
--------------------------------------------------------------------------------------------------------------------------------------------------------------
HIP PROF:hip API statistics
--------------------------------------------------------------------------------------------------------------------------------------------------------------
|Name                                                    |Calls          |TotalDurationNs            |AverageNs                  |Percentage
              |
--------------------------------------------------------------------------------------------------------------------------------------------------------------
|hipMalloc                                               |7              |422396931                  |60342418                   |99.525759608
4884          |
|hipMemcpy                                               |6              |811711                     |135285                     |0.1912564886
92542         |
|hipLaunchKernel                                         |51             |763441                     |14969                      |0.1798830433
29366         |
|hipEventRecord                                          |20             |131450                     |6572                       |0.0309724340
789204        |
|hipEventSynchronize                                     |10             |106120                     |10612                      |0.0250041438
147968        |
|__hipPopCallConfiguration                               |51             |54660                      |1071                       |0.0128790661
601658        |
|__hipPushCallConfiguration                              |51             |51320                      |1006                       |0.0120920906
575139        |
|hipFree                                                 |7              |35950                      |5135                       |0.0084705896
1686717       |
|hipEventElapsedTime                                     |10             |25450                      |2545                       |0.0059965648
3308121       |
|hipDeviceSynchronize                                    |1              |22720                      |22720                      |0.0053533183
8929686       |
|hipEventCreate                                          |2              |6320                       |3160                       |0.0014891272
9843117       |
|hipEventDestroy                                         |2              |3580                       |1790                       |0.0008435246
40567023      |
--------------------------------------------------------------------------------------------------------------------------------------------------------------
|Total                                                   |218            |424409653                  |1946833                    |100.0
              |
--------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------------------
HIP PROF:hip kernel statistics
--------------------------------------------------------------------------------------------------------------------------------------------------------------
|Name                                    |Pars                       |Calls          |TotalDurationNs            |AverageNs                  |
Percentage    |
```

图 2.9: 结果分析

由结果可知，从计算时间上来看，CPU 执行时间为 0.14528 ms（5 次测试平均），DCU 执行时间为 0.109514 ms（10 次测试平均，纯计算时间），加速比约为 1.33 倍，说明 DCU 实现相较于 CPU 有明显加速效果。通过计算准确性验证，最大误差约为 1.77636e-15，表明 DCU 计算结果与 CPU 结果高度一致，数值误差极小。从计算性能（GFLOPS）上来看，CPU 约 4.23 GFLOPS DCU 约 5.61 GFLOPS，提升明显。

另外通过 HIP Profiler 统计的 API 调用时间分布可见：hipMalloc 申请设备内存占总时间约 99.53%，为主要瓶颈。这表明内存分配操作耗时较长，在实际应用中我们应尽可能减少频繁的内存申请，以提高整体效率。

纯计算核函数（kernel）调用时间占比较低，其中：主要核函数 matmul_kernel_optimized 共调用 21 次，平均执行时间分别为 7010 ns 和 7375 ns，占核函数执行总时间的 54.45%。其他如加偏置和激活函数核执行时间分别占 17.15% 和 16.51%，总体核函数执行时间高效且均衡。

其他 API 调用时间占比较小，对整体性能影响较小。

总体来说基于 DCU 的矩阵乘法的多层感知机加速方案在保证精度的前提下，实现了超过 1.3 倍的加速，优于 CPU。

表 1: 不同参数组合下的网络规模与加速比变化

BATCH	I	H	O	CPU time	DCU time	加速比
1024	10	20	5	0.14528 ms	0.109514 ms	1.32659
256	10	20	5	0.03743 ms	0.0383178 ms	0.97683
1024	20	40	5	0.433263 ms	0.0490933 ms	8.82529
1024	10	20	20	0.223034 ms	0.0473252 ms	4.7128
2048	20	40	10	1.18482 ms	0.0571488 ms	20.7323

可以看到，在规模比较小的时候，DCU 由于数据传输或是并行资源没有得到充分利用等原因甚至出现了负优化，但当数据规模比较大的时候，其并行计算的优势就显现了出来，我们可以观察到，随着数据规模增大，CPU 时间随着计算量增加迅速上升，DCU 时间基本保持在较低水平，甚至出现了大规模运行时间比小规模低的情况（充分利用并行资源、计算核心），加速比从 0.97 提升至 20.73，呈指数级增长。这说明规模越大，DCU 加速效果越好，加速比越高。

3 总结感想

在这次实验中，我尝试了多种并行加速技术，包括 OpenMP、MPI、块级优化（Block）、SIMD 指令优化，以及基于 DCU 的加速方法，还应用了 DCU 加速神经网络的前向传播过程。

在矩阵乘法这类计算密集型任务中，CPU 并行化能在一定程度上提升性能，但其效果受限于核心数和内存带宽。相比之下，DCU 更适合处理大批量、高维度的数据任务，实验中也观察到了随着数据规模的增大，DCU 的加速效果越发明显。这也让我意识到并行方法的选择应充分考虑任务特征和硬件架构的匹配性。

总之，本次实验不仅加深了我对不同并行模型的理解，也让我意识到实际系统中计算与通信、代码复杂度与性能提升之间的权衡。