《先导杯》实验报告

学号：2313211　　姓名：2313211

实验平台

1.1pc电脑

个人`PC`电脑实验要求如下：

1. 使用个人电脑完成，不仅限于`visual studio`、`vscode`等。
2. 在完成矩阵乘法优化后，测试矩阵规模在$1024-4096$，或更大维度上，至少进行4个矩阵规模维度的测试。如`PC`电脑有`Nvidia`显卡，建议尝试`CUDA`代码。
3. 在作业中需总结出不同层次，不同规模下的矩阵乘法优化对比，对比指标包括计算耗时、运行性能、加速比等。
4. 在作业中总结优化过程中遇到的问题和解决方式。

1.2“超算互联网”平台

在”超算互联网“平台上进行计算资源的申请，即可获得计算资源，使用`mpic++ -fopenmp -o outputfile sourcefile.cpp`类型的编译语句即可编译文件，使用`./outputfile`进行文件的运行。

基础题目

2.1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；

2.2理论分析

2.2.1 OpenMP 多线程并行优化

- **并行度**：在最外层循环进行并行化，将矩阵的行分配给不同线程
- **理论加速比**：在理想情况下，速度提升与CPU核心数成正比
- 限制因素
  - 内存带宽瓶颈：多线程同时访问内存可能导致带宽饱和
  - 线程创建和管理开销
  - 非连续内存访问（特别是对矩阵B的列访问）
- **适用场景**：单机多核系统，中小规模矩阵

2.2.2 块状分块(Block Tiling)优化

理论性能提升分析：

- **缓存利用率**：将大矩阵分解为适合CPU缓存大小的小块进行处理
- **内存访问优化**：减少缓存缺失和主内存访问次数
- 理论加速比：取决于系统的内存层次结构和缓存大小
  - L1/L2缓存命中率显著提高可带来2-5倍性能提升
- 关键参数：块大小(block_size)需要根据CPU缓存大小调整
  - 太小：增加循环开销
  - 太大：无法充分利用缓存
- **适用场景**：各种规模矩阵，特别是大矩阵

2.2.3 MPI 多进程并行优化

理论性能提升分析：

- **分布式计算能力**：可以利用多台机器的计算资源
- **内存分布**：每个进程只需存储部分数据，可处理超大矩阵
- 理论加速比：
  - 计算密集型情况：接近线性加速（与进程数成正比）
  - 通信密集型情况：随进程数增加而收益递减
- 限制因素：
  - 进程间通信开销
  - 负载均衡问题
  - 网络带宽和延迟
- **适用场景**：大规模矩阵，多节点集群环境

2.2.4循环顺序优化(ikj loop order)

理论性能提升分析：

- **内存访问模式优化**：改变循环顺序，使内存访问更加连续
- 缓存友好性：
  - 缓存A中的单个元素(val_A)并在内层循环重用
  - 对B和C的访问变得更加连续（行优先）
- **理论加速比**：1.5-3倍，取决于矩阵大小和内存架构
- 优势：
  - 实现简单
  - 无需额外内存
  - 适合任何环境
- **适用场景**：各种规模矩阵，特别是缓存敏感的系统

2.2.5综合对比与最优组合

优化方法	主要优势	限制因素	理想加速比
OpenMP	易实现、多线程并行	内存带宽、线程开销	~核心数
Block Tiling	缓存友好、内存效率	块大小选择敏感	2-5倍
MPI	分布式、大规模	通信开销、负载均衡	~进程数
循环顺序	内存访问优化	优化空间有限	1.5-3倍

理论上最佳性能应是**多种方法的组合**:

- MPI在节点间分布计算负载
- 每个节点内部使用OpenMP多线程
- 每个线程内部采用块分块和优化循环顺序

2.3关键代码展示

2.3.1 lesson1.cpp

```
// 方式1：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) {
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < P; ++j) {
            double sum = 0.0;
            for (int k = 0; k < M; ++k) {
                sum += A[i * M + k] * B[k * P + j];
            }
            C[i * P + j] = sum;
        }
    }
}

// 方式2：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) {
    for (int i0 = 0; i0 < N; i0 += block_size) {
        for (int j0 = 0; j0 < P; j0 += block_size) {
            for (int k0 = 0; k0 < M; k0 += block_size) {
                for (int i = i0; i < std::min(i0 + block_size, N); ++i) {
                    for (int j = j0; j < std::min(j0 + block_size, P); ++j) {
                        for (int k = k0; k < std::min(k0 + block_size, M); ++k) {
                            C[i * P + j] += A[i * M + k] * B[k * P + j];
                        }
                    }
                }
            }
```

```
        }
    }
}

// 方式3：利用MPI消息传递，实现多进程并行优化
// C_ref_main is passed for rank 0 to optionally validate against it.
void matmul_mpi(int N, int M, int P, const std::vector<double>&
C_ref_main_for_validation) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    std::chrono::high_resolution_clock::time_point mpi_start_time, mpi_end_time;

    if (rank == 0) {
        mpi_start_time = std::chrono::high_resolution_clock::now();
    }

    int rows_per_proc_base = N / size;
    int remainder_rows = N % size;

    int local_N = rows_per_proc_base + (rank < remainder_rows ? 1 : 0);

    std::vector<double> local_A(local_N * M);
    std::vector<double> B_all(M * P);
    std::vector<double> local_C(local_N * P, 0.0);

    std::vector<double> A_all_mpi;
    std::vector<double> C_result_mpi_gathered;

    std::vector<int> sendcounts_A(size);
    std::vector<int> displs_A(size);
    std::vector<int> recvcounts_C(size);
    std::vector<int> displs_C(size);

    if (rank == 0) {
        A_all_mpi.resize(N * M);
        C_result_mpi_gathered.resize(N * P);

        // Initialize A and B on rank 0. Since init_matrix uses a fixed seed,
        // A_all_mpi and B_all here will be consistent with A and B in main (rank
0).
        init_matrix(A_all_mpi, N, M);
        init_matrix(B_all, M, P);

        int current_displ_A_val = 0;
        int current_displ_C_val = 0;
        for (int i = 0; i < size; ++i) {
            int num_rows_for_proc = rows_per_proc_base + (i < remainder_rows ? 1
: 0);

            sendcounts_A[i] = num_rows_for_proc * M;
            displs_A[i] = current_displ_A_val;
            current_displ_A_val += sendcounts_A[i];

            recvcounts_C[i] = num_rows_for_proc * P;
```

```
            displs_C[i] = current_displ_C_val;
            current_displ_C_val += recvcounts_C[i];
        }
    }

    MPI_Bcast(B_all.data(), M * P, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    MPI_Scatterv(A_all_mpi.data(), sendcounts_A.data(), displs_A.data(),
MPI_DOUBLE,
                 local_A.data(), local_N * M, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    for (int i = 0; i < local_N; ++i) {
        for (int j = 0; j < P; ++j) {
            double sum = 0.0;
            for (int k = 0; k < M; ++k) {
                sum += local_A[i * M + k] * B_all[k * P + j];
            }
            local_C[i * P + j] = sum;
        }
    }

    MPI_Gatherv(local_C.data(), local_N * P, MPI_DOUBLE,
                C_result_mpi_gathered.data(), recvcounts_C.data(),
displs_C.data(), MPI_DOUBLE,
                0, MPI_COMM_WORLD);

    if (rank == 0) {
        mpi_end_time = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double, std::milli> duration_ms_chrono =
mpi_end_time - mpi_start_time;
        double duration_s = duration_ms_chrono.count() / 1000.0;
        double gflops = (2.0 * N * M * P) / (duration_s * 1e9);

        std::cout << "[MPI] Computation complete on root." << std::endl;
        std::cout << "[MPI] Time: " << duration_ms_chrono.count() << " ms" <<
std::endl;
        std::cout << "[MPI] GFLOPS: " << std::fixed << std::setprecision(2) <<
gflops << std::endl;

        bool is_valid = validate(C_result_mpi_gathered,
C_ref_main_for_validation, N, P);
        std::cout << "[MPI] Valid: " << is_valid << std::endl;
    }
}

// 方式4: 其他方式（ikj loop order）
void matmul_other(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 k = 0; k < M; ++k) {
            double val_A = A[i * M + k];
            for (int j = 0; j < P; ++j) {
                C[i * P + j] += val_A * B[k * P + j];
            }
        }
```

```
    }
}
```

2.3.2 lesson1_duc.cpp

```
// 主要修改函数
__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;
    }
}

// 主要修改部分
    std::cout << "\nRunning HIP GPU computation..." << std::endl;
    double *d_A, *d_B, *d_C;
    size_t size_A = (size_t)n_val * m_val * sizeof(double);
    size_t size_B = (size_t)m_val * p_val * sizeof(double);
    size_t size_C = (size_t)n_val * p_val * sizeof(double);

    // Allocate device memory
    HIP_CHECK(hipMalloc(&d_A, size_A));
    HIP_CHECK(hipMalloc(&d_B, size_B));
    HIP_CHECK(hipMalloc(&d_C, size_C));

    // Copy data from host to device
    HIP_CHECK(hipMemcpy(d_A, h_A.data(), size_A, hipMemcpyHostToDevice));
    HIP_CHECK(hipMemcpy(d_B, h_B.data(), size_B, hipMemcpyHostToDevice));

    // Kernel launch parameters
    // Typically 16x16 or 32x32 for 2D kernels, adjust based on GPU
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((p_val + threadsPerBlock.x - 1) / threadsPerBlock.x,
                   (n_val + threadsPerBlock.y - 1) / threadsPerBlock.y);

    // Use hipEvent_t for accurate GPU timing
    hipEvent_t start_event, stop_event;
    HIP_CHECK(hipEventCreate(&start_event));
    HIP_CHECK(hipEventCreate(&stop_event));

    // Record start event
    HIP_CHECK(hipEventRecord(start_event, 0));

    // Launch kernel
    hipLaunchKernelGGL(matmul_kernel, numBlocks, threadsPerBlock, 0, 0, d_A, d_B,
d_C, n_val, m_val, p_val);
    HIP_CHECK(hipGetLastError()); // Check for kernel launch errors
```

```
    // Record stop event and synchronize
    HIP_CHECK(hipEventRecord(stop_event, 0));
    HIP_CHECK(hipEventSynchronize(stop_event)); // Wait for the event to complete

    float milliseconds = 0;
    HIP_CHECK(hipEventElapsedTime(&milliseconds, start_event, stop_event));

    double gpu_duration_s = milliseconds / 1000.0;
    double gpu_gflops = (gpu_duration_s > 0) ? (total_flops / (gpu_duration_s *
1e9)) : 0;

    std::cout << "[HIP] Kernel Time: " << milliseconds << " ms" << std::endl;
    std::cout << "[HIP] GFLOPS: " << std::fixed << std::setprecision(2) <<
gpu_gflops << std::endl;


    // Copy result from device to host
    HIP_CHECK(hipMemcpy(h_C_gpu.data(), d_C, size_C, hipMemcpyDeviceToHost));

    // Validate
    if (validate(h_C_cpu, h_C_gpu, n_val, p_val)) {
        std::cout << "[HIP] Valid: 1 (Results match CPU)" << std::endl;
    } else {
        std::cout << "[HIP] Valid: 0 (Results DO NOT match CPU)" << std::endl;
    }

    // Free device memory
    HIP_CHECK(hipEventDestroy(start_event));
    HIP_CHECK(hipEventDestroy(stop_event));
    HIP_CHECK(hipFree(d_A));
    HIP_CHECK(hipFree(d_B));
    HIP_CHECK(hipFree(d_C));

    std::cout << "\nFinished." << std::endl;
    return 0;
```

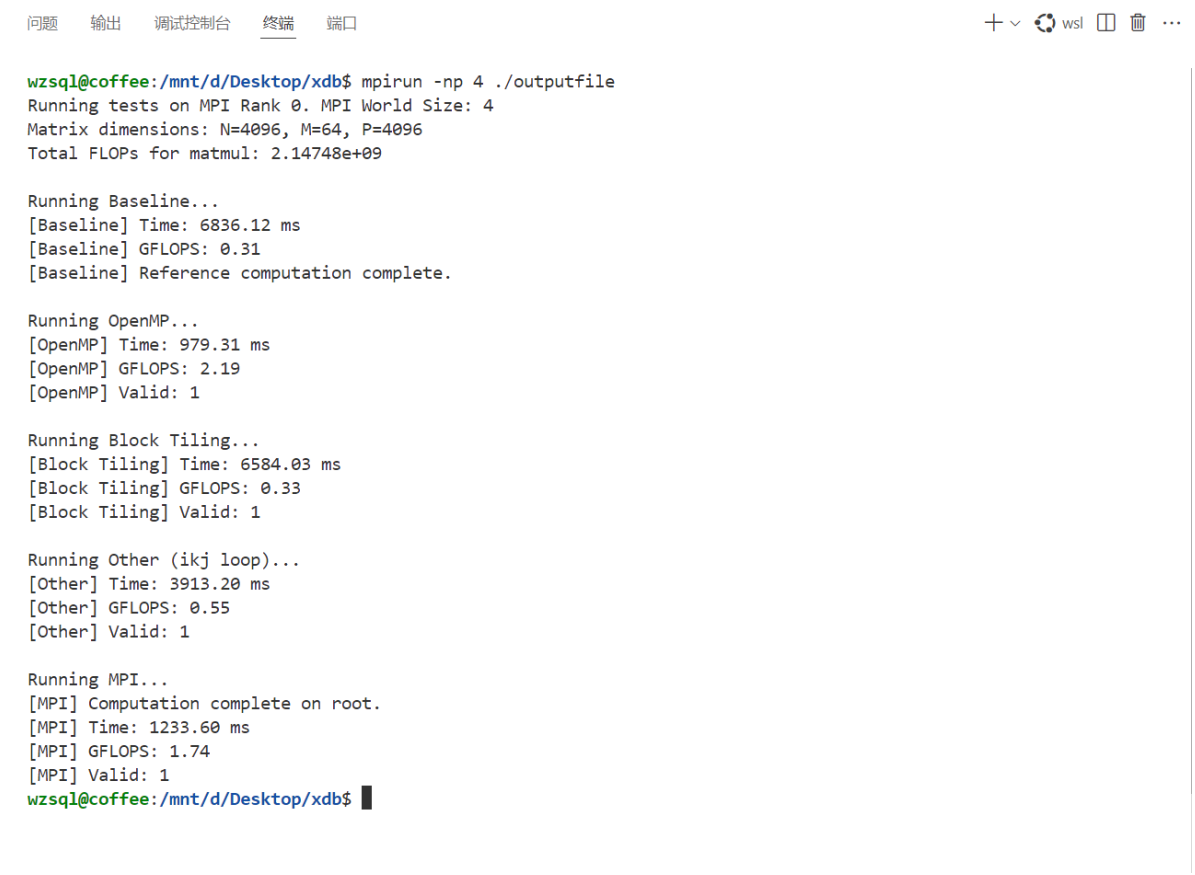
2.4结果展示

2.4.1 问题1

我们引入一个验证正确性的函数,由于涉及到浮点数的计算，所以只要误差在一定范围内我们就可以判定结果是正确的。

```cpp
bool validate(const std::vector<double>& A, const std::vector<double>& B, int
rows, int cols, double tol = 1e-6) {
    // 检查矩阵维度是否匹配
    if (A.size() != B.size() || A.size() != (size_t)rows * cols) {
        std::cerr << "Validation error: Matrix dimensions mismatch..." <<
std::endl;
        return false;
    }
    // 逐元素比较，允许1e-6的误差容差
    for (int i = 0; i < rows * cols; ++i)
        if (std::abs(A[i] - B[i]) > tol) {
            return false;
        }
    return true;
}
```

当我们的结果正确时`valid`输出`1`，否则输出`0`。



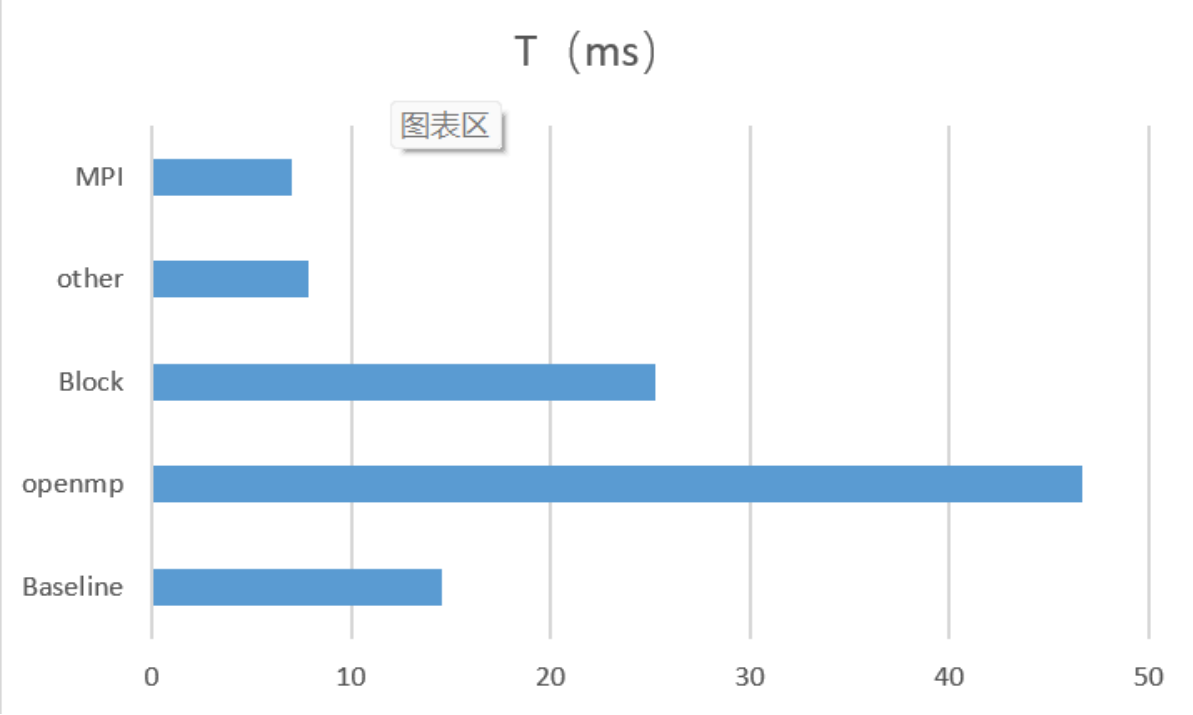
我们可以看到所有的结果均为1，则证明我们的实现的代码是正确的。

2.4.2 问题2

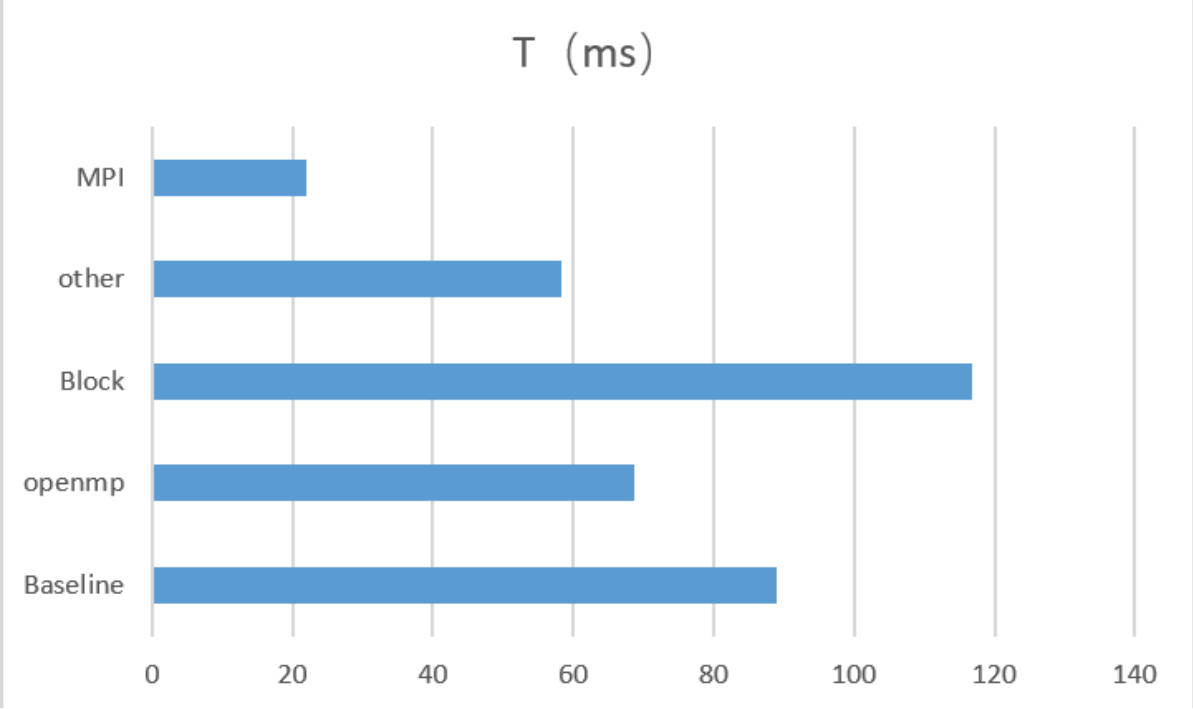
在问题二中我们选择了不同类型的矩阵来观察对于每一种加速方法适用于什么样的数据类型。

2.4.2.1正方形矩阵

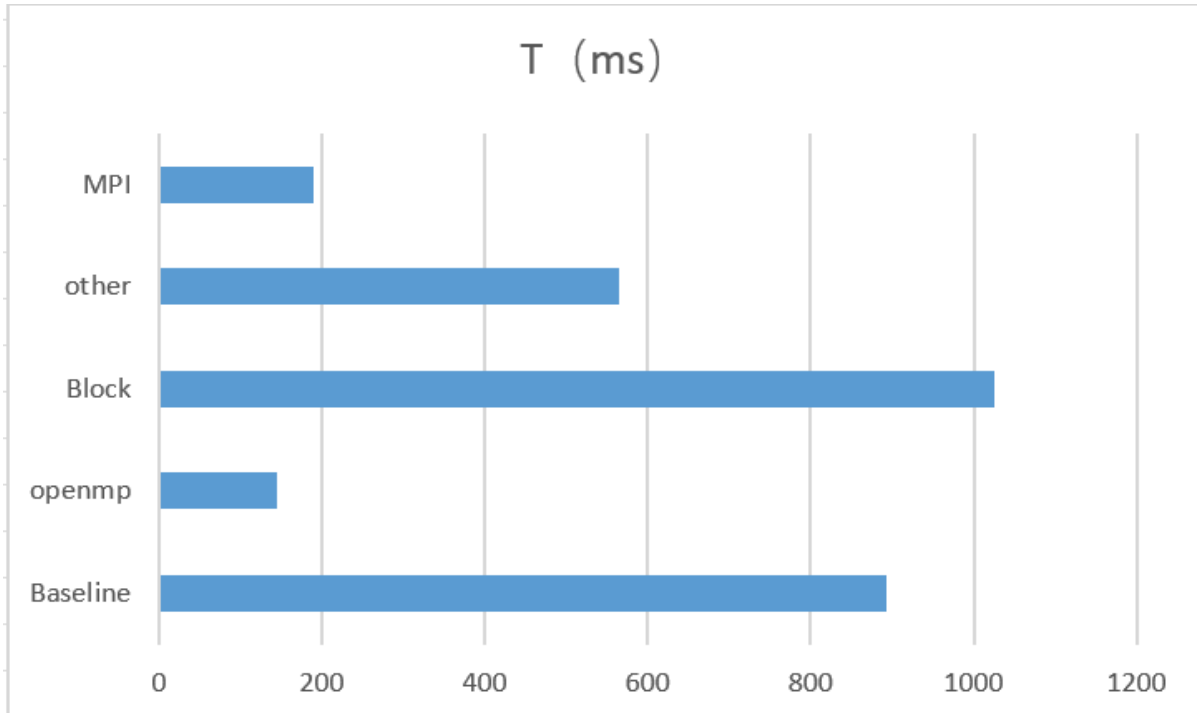
2.4.2.1.1 128 128 128



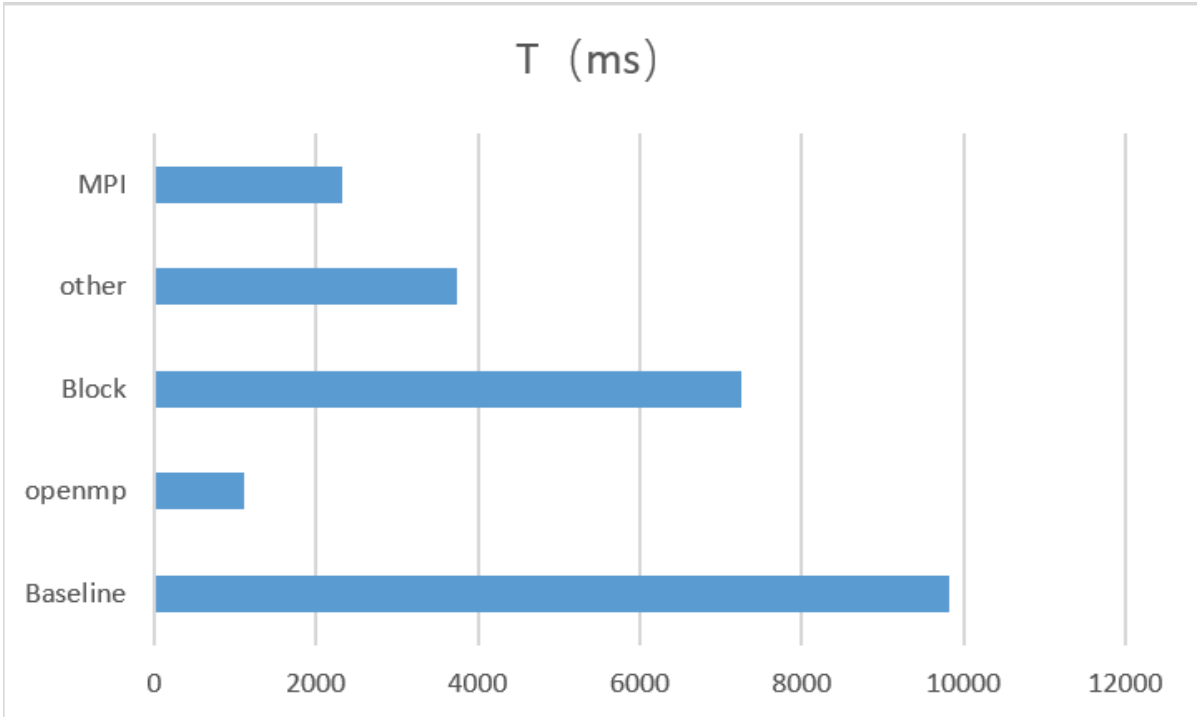
2.4.2.1.2 256 256 256



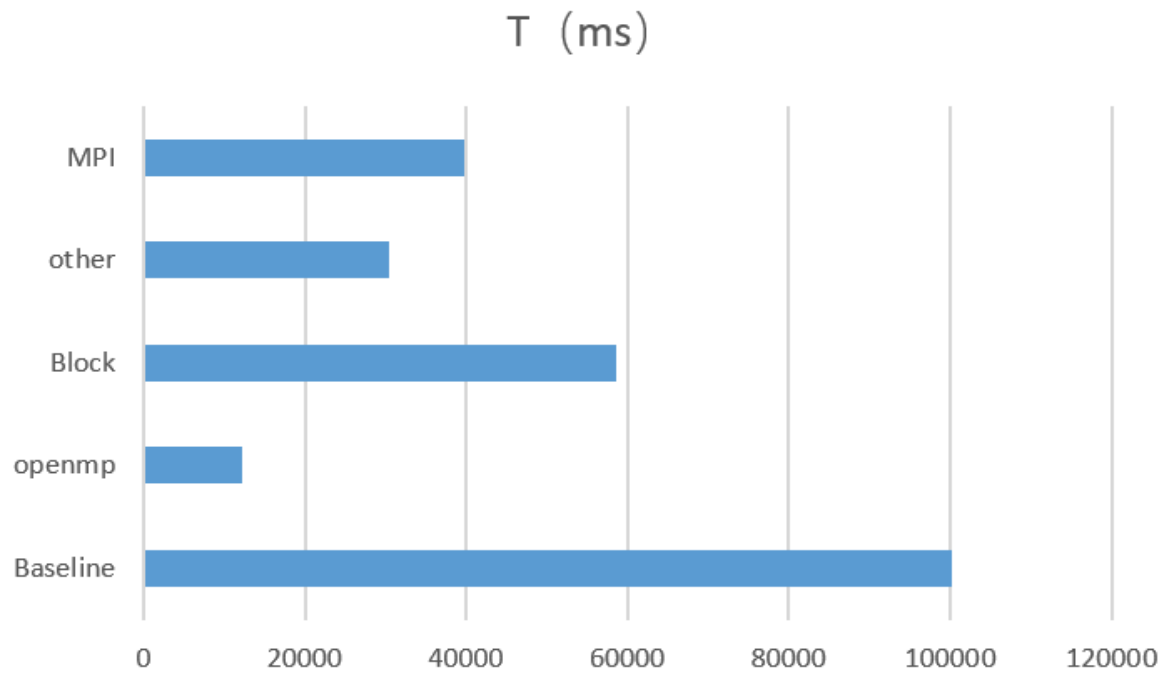
2.4.2.1.3 512 512 512



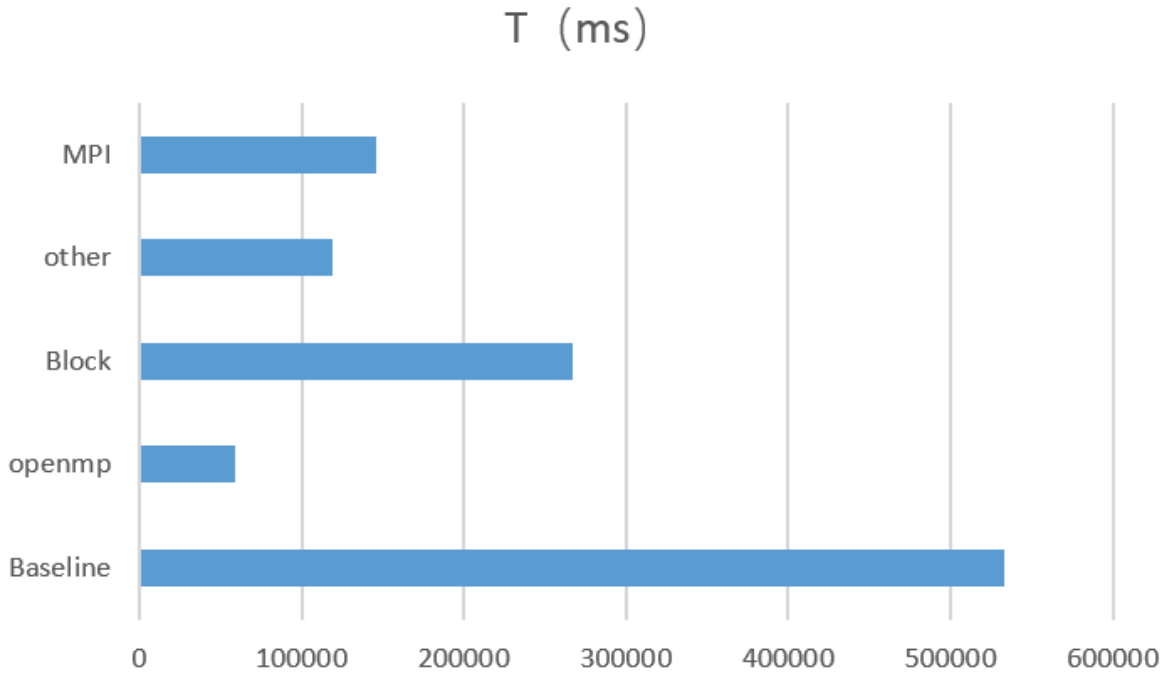
2.4.2.1.4 1024 1024 1024



2.4.2.1.5 2048 2048 2048

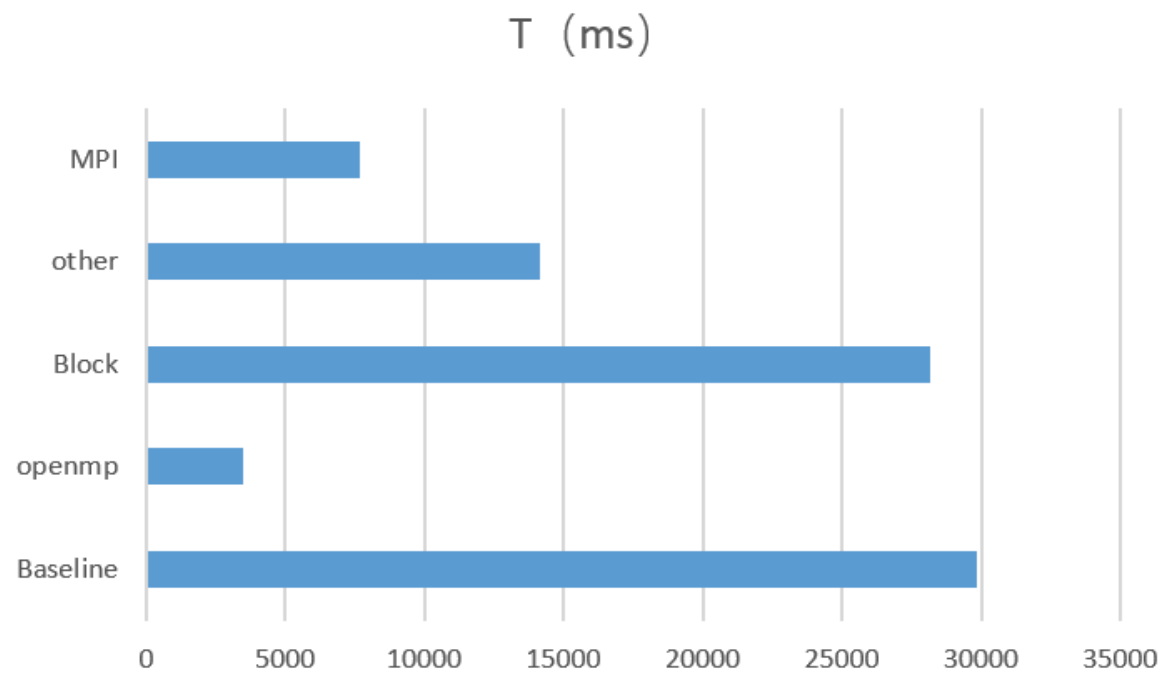


2.4.2.1.6 3072 3072 3072



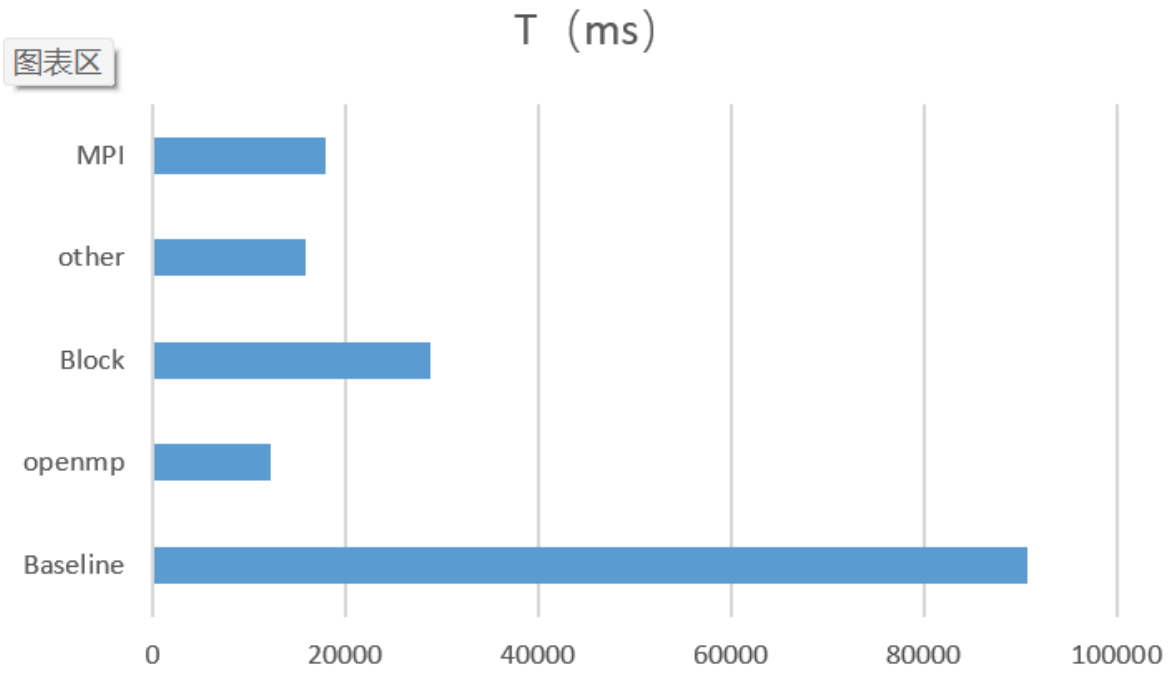
我们可以看出来，在矩阵的大小越来越大的时候，$openmp$算法越来越显示出多线程算法的优势出来了。而$block$算法很显然并不能够很好地克服各部分收集的损耗所带来的时间增长。但是我们同时也可以看出来，当矩阵变得特别小的时候，$openmp$算法反而变得比$Baseline$还要慢，这显示出了多线程的线程之间的合并占了总消耗时长的非常大的一部分。

2.4.2.2 高瘦型（4096 1024 1024）



在这个类型中*openmp*的优势变得极其巨大，而*Block*算法显得比较不尽人意。

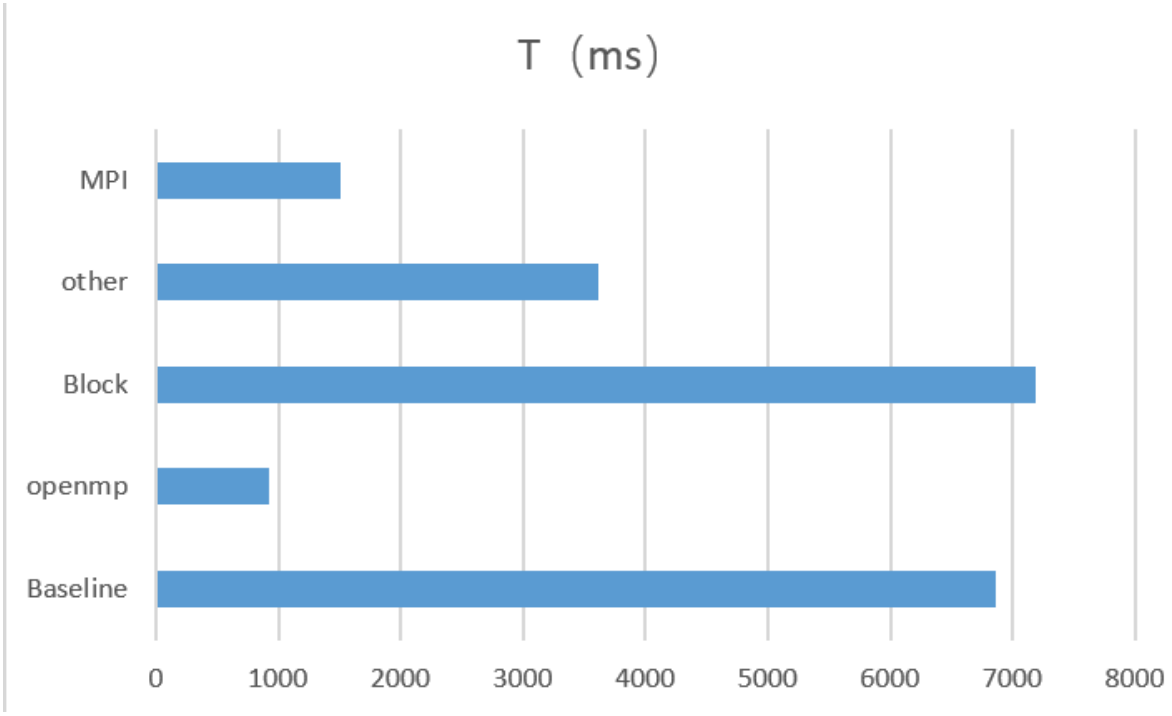
2.4.2.3 宽扁形（1024 4096 1024）



在这种类型的矩阵中，每种加速算法表现得都不错，并且都显著降低了运算时间。

2.4.2.4 一个维度显著小

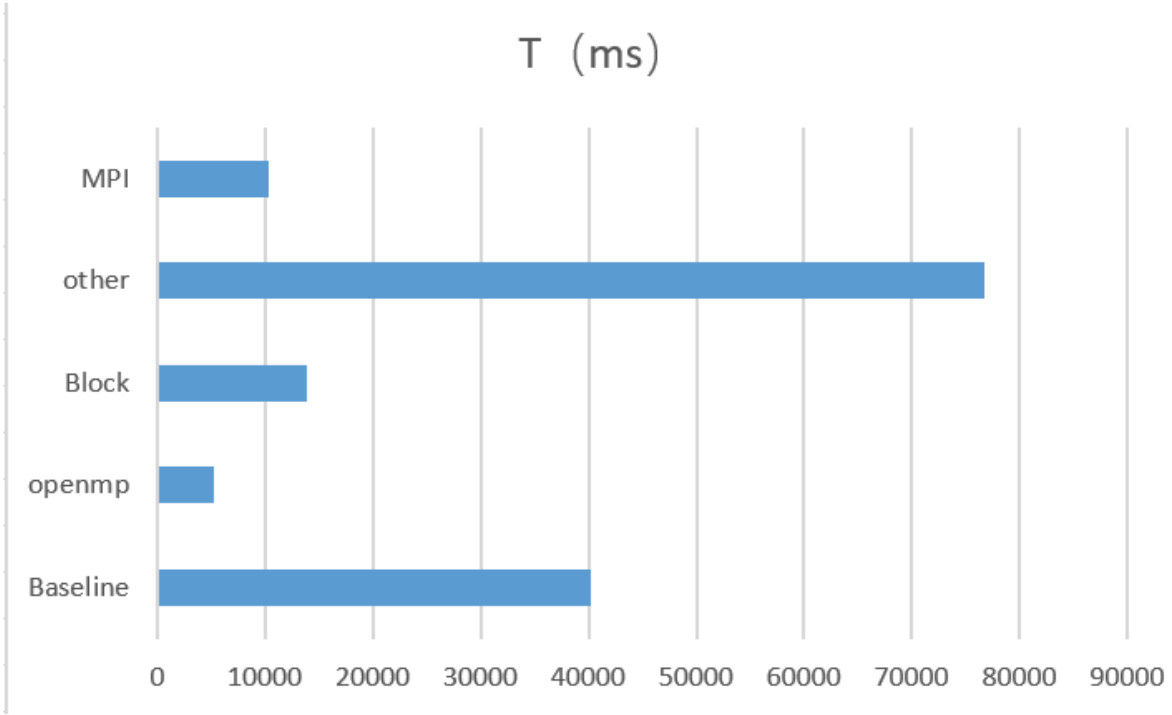
2.4.2.4.1 4096 256 4096



我们发现在这种情况下，$Block$算法居然运算所需要的时间增加了，说明在这种矩阵情况下对矩阵做分块处理并不是一个很好的选择。

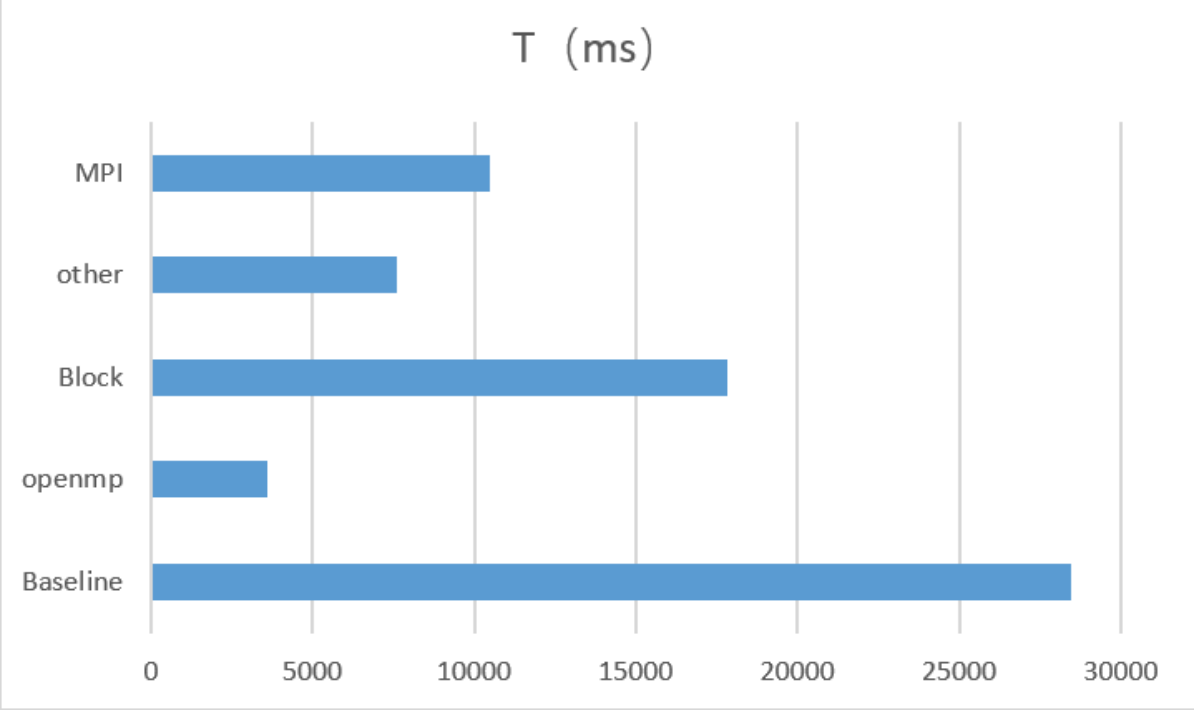
2.4.2.5 一个维度显著大

2.4.2.5.1 512 8192 215（内积主导）

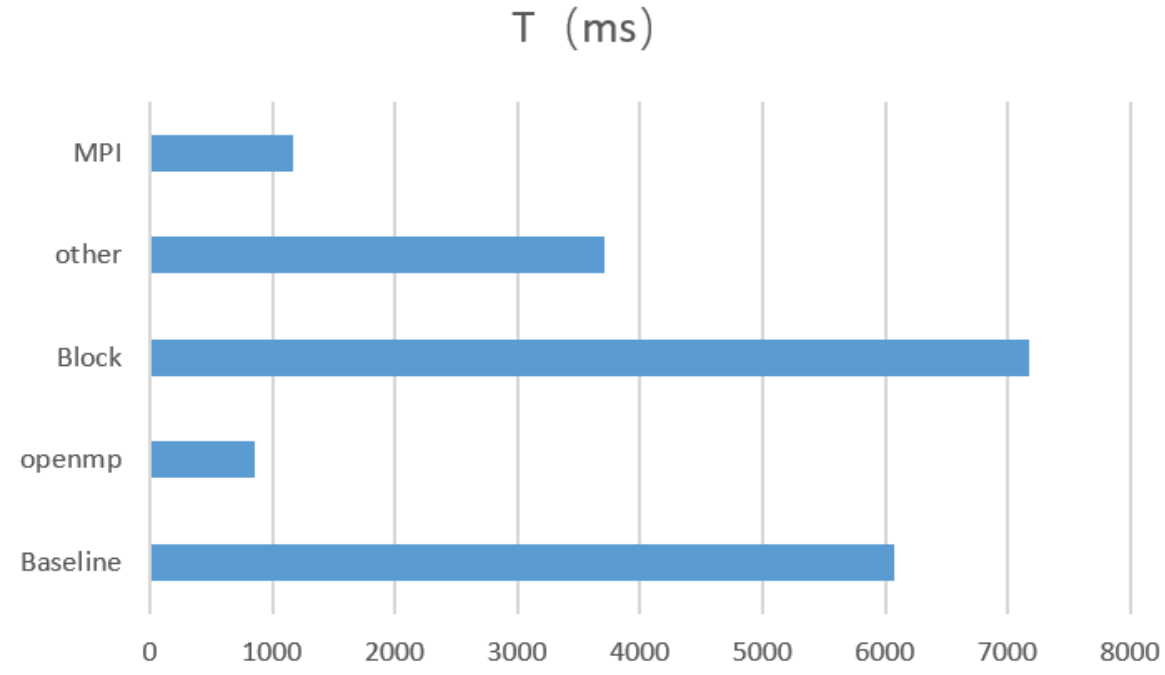


现在是$other$的算法比较慢，其他的都效果很好

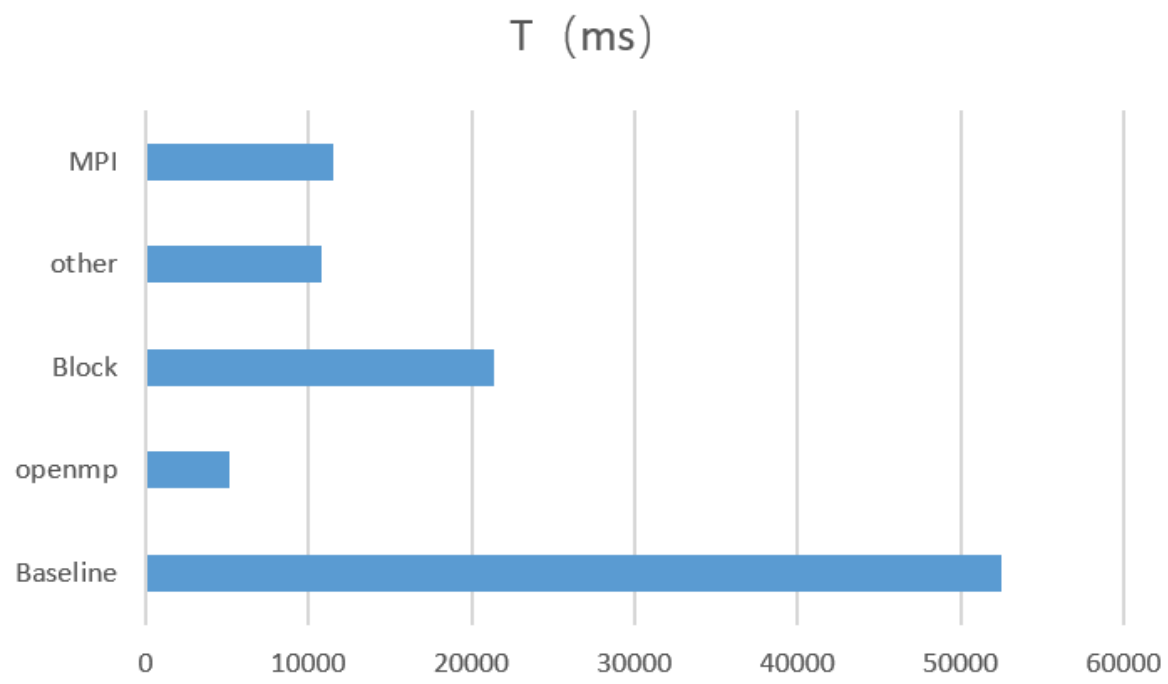
2.4.2.5.2 512 512 8192（外积主导）



2.4.2.5.3 4096 64 4096



2.4.2.6 随机矩阵（1536 2048 1024）



2.5 归纳总结

我们从计算耗时、运行性能以及加速比这三个方面来进行优化的评估。

> **计算耗时：**
>
> 按照定义计算的矩阵乘法耗时最长，因为它需要执行三重循环来逐个元素进行计算；优化后的矩阵乘法采用分块矩阵乘法的方法，在矩阵乘法计算中减少了不必要的访存操作，从而提高了计算效率，耗时相对较短；并行计算矩阵乘法利用多线程进行计算，可以同时进行多个乘法运算，因此具有更快的计算速度，耗时最短。

> **运行性能：**
>
> 按照定义计算的矩阵乘法的性能较低，因为它使用了三重循环的嵌套，导致计算复杂度较高；优化的矩阵乘法通过采用分块矩阵乘法的优化方法，减少了不必要的访存操作，提高了运行性能；并行计算矩阵乘法利用多线程实现并行计算，充分利用多核处理器的计算能力，因此具有更好的运行性能。

> **加速比：**
>
> 优化的矩阵乘法和并行计算矩阵乘法相对于按照定义计算的矩阵乘法都能够取得较好的加速比；优化的矩阵乘法通过减少不必要的访存操作和利用分块矩阵乘法的优化策略，加速比相对较高；并行计算矩阵乘法通过利用多线程并行计算的特点，能够进一步提高计算速度，加速比最高。

以下维度的正方形矩阵：

- **128×128×128**
- **256×256×256**
- **512×512×512**
- **1024×1024×1024**
- **2048×2048×2048**
- **3072×3072×3072**

结论：

- 随着矩阵尺寸增大，**多线程并行算法（如 OpenMP）** 表现出明显优势。
- 在小矩阵情况下，多线程反而不如单线程，因为线程创建和调度开销占比较大。

高瘦型矩阵（4096×1024×1024）

- 表现：
  - **MPI 多进程并行** 展现出显著优势。
  - Block Tiling 等其他方法表现一般。

  宽扁形矩阵（1024×4096×1024）

  - 表现：
    - 所有优化方法都显著减少了运算时间，说明适用于这种结构的数据。

  单一维度显著小的情况

  测试案例：

  - **4096×256×4096**

  结论：分块优化（Block Tiling）效果不佳，表明在这种矩阵结构下分块策略不适合。

  单一维度显著大的情况

  测试案例：

  - **512×8192×215**（内积主导）
  - **512×512×8192**（外积主导）
  - **4096×64×4096**

  结论：

  - **MPI 和 Block Tiling** 表现良好。
  - **OpenMP** 在某些情况下效率较低，可能受内存带宽限制。

  随机矩阵（1536×2048×1024）

  - 表现： 各种优化方法都能有效提升性能，适合复杂数据分布。

进阶题目1

3.1 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）；

3.2算法设计

3.2.1网络架构

这是一个两层的MLP网络：

- **输入层**: 10维 (I=10)
- **隐藏层**: 20维 (H=20) + ReLU激活函数
- **输出层**: 5维 (O=5)
- **批处理大小**: 1024 (BATCH=1024)

3.2.2前向传播算法

数学表达式：

```
隐藏层: H = ReLU(X * W1 + B1)
输出层: Y = H * W2 + B2
```

其中：

- `X`: 输入矩阵 [1024 × 10]
- `W1`: 第一层权重矩阵 [10 × 20]
- `B1`: 第一层偏置向量 [20]
- `W2`: 第二层权重矩阵 [20 × 5]
- `B2`: 第二层偏置向量 [5]

3.2.3GPU核函数实现

3.2.3.1基础矩阵乘法核函数

- 每个线程计算结果矩阵的一个元素
- 使用朴素的三重循环算法
- 线程布局：2D grid + 2D block

```
__global__ void matmul_kernel(const double* A, const double* B, double* C, int M,
int N, int K)
```

3.2.3.2优化的分块矩阵乘法核函数

优化原理：

- 使用**共享内存**减少全局内存访问
- 将矩阵分成16×16的**分块（tiles）**
- 每个线程块处理一个输出分块
- 通过__syncthreads()确保数据同步

分块算法流程：

1. 加载A和B的分块到共享内存

2. 同步等待所有线程加载完成

3. 计算当前分块的部分积

4. 移动到下一个分块，重复步骤1-3

```
__global__ void matmul_kernel_tiled(const double* A, const double* B, double* C,
int M, int N, int K)
```

3.2.3.3偏置加法核函数

将偏置向量广播到矩阵的每一行

```
__global__ void add_bias_kernel(double* matrix, const double* bias, int M_rows,
int N_cols)
```

3.2.3.4ReLU激活函数核函数

ReLU激活函数核函数

```
__global__ void relu_kernel(double* A, int size)
```

3.3关键代码

```
// Tiled Matrix Multiplication Kernel (Optimized)
__global__ void matmul_kernel_tiled(const double* A, const double* B, double* C,
int M, int N, int K) {
    __shared__ double tile_A[TILE_WIDTH][TILE_WIDTH];
    __shared__ double tile_B[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Calculate the row and col of the C element to work on
    int row = by * TILE_WIDTH + ty;
    int col = bx * TILE_WIDTH + tx;

    double sum = 0.0;

    // Loop over the tiles of A and B required to compute the C element
    for (int t = 0; t < (K + TILE_WIDTH - 1) / TILE_WIDTH; ++t) {
        // Load a tile of A into shared memory
        if (row < M && (t * TILE_WIDTH + tx) < K) {
            tile_A[ty][tx] = A[row * K + (t * TILE_WIDTH + tx)];
        } else {
            tile_A[ty][tx] = 0.0;
        }

        // Load a tile of B into shared memory
        if (col < N && (t * TILE_WIDTH + ty) < K) {
            tile_B[ty][tx] = B[(t * TILE_WIDTH + ty) * N + col];
        } else {
```

```
            tile_B[ty][tx] = 0.0;
        }
        __syncthreads(); // Ensure all threads in the block have loaded their
data

        // Multiply the two tiles
        for (int i = 0; i < TILE_WIDTH; ++i) {
            sum += tile_A[ty][i] * tile_B[i][tx];
        }
        __syncthreads(); // Ensure all threads in the block have finished using
the current tiles
    }

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

// ReLU Kernel
__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]);
    }
}

// --- Utility Functions ---
void random_init(std::vector<double>& mat) {
    for (auto& val : mat) {
        val = static_cast<double>(rand()) / RAND_MAX * 2.0 - 1.0;
    }
}

bool validate_results(const std::vector<double>& gpu_res, const
std::vector<double>& cpu_res, double tolerance = 1e-5) {
    if (gpu_res.size() != cpu_res.size()) {
        std::cerr << "Validation Error: Size mismatch! GPU: " << gpu_res.size()
<< ", CPU: " << cpu_res.size() << std::endl;
        return false;
    }
    for (size_t i = 0; i < gpu_res.size(); ++i) {
        if (std::fabs(gpu_res[i] - cpu_res[i]) > tolerance) {
            std::cerr << "Validation Error at index " << i << ": GPU=" <<
gpu_res[i] << ", CPU=" << cpu_res[i]
                      << ", Diff=" << std::fabs(gpu_res[i] - cpu_res[i]) <<
std::endl;
            return false;
        }
    }
    return true;
}

// --- GPU Setup ---
    double *d_X, *d_W1, *d_B1, *d_H_gpu, *d_W2, *d_B2, *d_Y_gpu;
    size_t size_X = BATCH * I * sizeof(double);
    size_t size_W1 = I * H * sizeof(double);
```

```
    size_t size_B1 = H * sizeof(double);
    size_t size_d_H_gpu = BATCH * H * sizeof(double);
    size_t size_W2 = H * O * sizeof(double);
    size_t size_B2 = O * sizeof(double);
    size_t size_d_Y_gpu = BATCH * O * sizeof(double);

    hipMalloc((void**)&d_X, size_X);
    hipMalloc((void**)&d_W1, size_W1);
    hipMalloc((void**)&d_B1, size_B1);
    hipMalloc((void**)&d_H_gpu, size_d_H_gpu);
    hipMalloc((void**)&d_W2, size_W2);
    hipMalloc((void**)&d_B2, size_B2);
    hipMalloc((void**)&d_Y_gpu, size_d_Y_gpu);

    hipMemcpy(d_X, h_X.data(), size_X, hipMemcpyHostToDevice);
    hipMemcpy(d_W1, h_W1.data(), size_W1, hipMemcpyHostToDevice);
    hipMemcpy(d_B1, h_B1.data(), size_B1, hipMemcpyHostToDevice);
    hipMemcpy(d_W2, h_W2.data(), size_W2, hipMemcpyHostToDevice);
    hipMemcpy(d_B2, h_B2.data(), size_B2, hipMemcpyHostToDevice);

    const int TPB_2D_BASIC = 16; // Threads per block for basic matmul
    const int TPB_1D_RELU = 256;


    // --- GPU MLP Forward Pass (Basic matmul_kernel) ---
    std::cout << "Running GPU MLP (Basic matmul_kernel)..." << std::endl;
    hipDeviceSynchronize();
    auto gpu_basic_start_time = std::chrono::high_resolution_clock::now();

    dim3 threads_matmul1_basic(TPB_2D_BASIC, TPB_2D_BASIC);
    dim3 blocks_matmul1_basic((H + TPB_2D_BASIC - 1) / TPB_2D_BASIC, (BATCH +
TPB_2D_BASIC - 1) / TPB_2D_BASIC);
    matmul_kernel<<<blocks_matmul1_basic, threads_matmul1_basic>>>(d_X, d_W1,
d_H_gpu, BATCH, H, I);

    dim3 threads_bias1_basic(TPB_2D_BASIC, TPB_2D_BASIC); // Can reuse
TPB_2D_BASIC
    dim3 blocks_bias1_basic((H + TPB_2D_BASIC - 1) / TPB_2D_BASIC, (BATCH +
TPB_2D_BASIC - 1) / TPB_2D_BASIC);
    add_bias_kernel<<<blocks_bias1_basic, threads_bias1_basic>>>(d_H_gpu, d_B1,
BATCH, H);

    dim3 threads_relu1(TPB_1D_RELU);
    dim3 blocks_relu1((BATCH * H + TPB_1D_RELU - 1) / TPB_1D_RELU);
    relu_kernel<<<blocks_relu1, threads_relu1>>>(d_H_gpu, BATCH * H);

    dim3 threads_matmul2_basic(TPB_2D_BASIC, TPB_2D_BASIC);
    dim3 blocks_matmul2_basic((O + TPB_2D_BASIC - 1) / TPB_2D_BASIC, (BATCH +
TPB_2D_BASIC - 1) / TPB_2D_BASIC);
    matmul_kernel<<<blocks_matmul2_basic, threads_matmul2_basic>>>(d_H_gpu, d_W2,
d_Y_gpu, BATCH, O, H);

    dim3 threads_bias2_basic(TPB_2D_BASIC, TPB_2D_BASIC);
    dim3 blocks_bias2_basic((O + TPB_2D_BASIC - 1) / TPB_2D_BASIC, (BATCH +
TPB_2D_BASIC - 1) / TPB_2D_BASIC);
```

```
    add_bias_kernel<<<blocks_bias2_basic, threads_bias2_basic>>>(d_Y_gpu, d_B2,
BATCH, O);

    hipDeviceSynchronize();
    auto gpu_basic_end_time = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> gpu_basic_duration_ms =
gpu_basic_end_time - gpu_basic_start_time;

    hipMemcpy(h_Y_gpu.data(), d_Y_gpu, size_d_Y_gpu, hipMemcpyDeviceToHost);
    bool basic_valid = validate_results(h_Y_gpu, h_Y_cpu);
    double gflops_basic = (total_flops / (gpu_basic_duration_ms.count() /
1000.0)) / 1e9;

    std::cout << "GPU (Basic) Execution Time: " << gpu_basic_duration_ms.count()
<< " ms" << std::endl;
    std::cout << "GPU (Basic) GFLOPS: " << std::fixed << std::setprecision(2) <<
gflops_basic << std::endl;
    std::cout << "GPU (Basic) Validation: " << (basic_valid ? "PASSED" :
"FAILED") << std::endl;
    std::cout << "-----------------------------------" << std::endl;

    // --- GPU MLP Forward Pass (Tiled matmul_kernel_tiled) ---
    std::cout << "Running GPU MLP (Tiled matmul_kernel_tiled)..." << std::endl;
    // Reset intermediate and output GPU buffers if necessary (though they are
overwritten)
    // hipMemset(d_H_gpu, 0, size_d_H_gpu); // Optional: clear if needed
    // hipMemset(d_Y_gpu, 0, size_d_Y_gpu); // Optional: clear if needed

    hipDeviceSynchronize();
    auto gpu_tiled_start_time = std::chrono::high_resolution_clock::now();

    // For matmul_kernel_tiled, threads per block must match TILE_WIDTH
    dim3 threads_matmul_tiled(TILE_WIDTH, TILE_WIDTH);
    dim3 blocks_matmul1_tiled((H + TILE_WIDTH - 1) / TILE_WIDTH, (BATCH +
TILE_WIDTH - 1) / TILE_WIDTH);
    matmul_kernel_tiled<<<blocks_matmul1_tiled, threads_matmul_tiled>>>(d_X,
d_W1, d_H_gpu, BATCH, H, I);

    // Bias and ReLU kernels remain the same
    add_bias_kernel<<<blocks_bias1_basic, threads_bias1_basic>>>(d_H_gpu, d_B1,
BATCH, H); // Reusing basic launch params
    relu_kernel<<<blocks_relu1, threads_relu1>>>(d_H_gpu, BATCH * H);

    dim3 blocks_matmul2_tiled((O + TILE_WIDTH - 1) / TILE_WIDTH, (BATCH +
TILE_WIDTH - 1) / TILE_WIDTH);
    matmul_kernel_tiled<<<blocks_matmul2_tiled, threads_matmul_tiled>>>(d_H_gpu,
d_W2, d_Y_gpu, BATCH, O, H);

    add_bias_kernel<<<blocks_bias2_basic, threads_bias2_basic>>>(d_Y_gpu, d_B2,
BATCH, O); // Reusing basic launch params

    hipDeviceSynchronize();
    auto gpu_tiled_end_time = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> gpu_tiled_duration_ms =
gpu_tiled_end_time - gpu_tiled_start_time;
```

```
    hipMemcpy(h_Y_gpu.data(), d_Y_gpu, size_d_Y_gpu, hipMemcpyDeviceToHost);
    bool tiled_valid = validate_results(h_Y_gpu, h_Y_cpu);
    double gflops_tiled = (total_flops / (gpu_tiled_duration_ms.count() /
1000.0)) / 1e9;
```

3.4结果展示

```
9 warnings generated when compiling for host.
root@worker-0:/public/home/xdzs2025coffe# ./mlp_forward
MLP Configuration:
Batch Size (BATCH): 1024
Input Dim (I):    10
Hidden Dim (H):   20
Output Dim (O):   5
-------------------------------------
Total Theoretical FLOPs: 660480
-------------------------------------
Running CPU MLP Forward Pass for reference...
CPU Execution Time: 0 ms
-------------------------------------
Running GPU MLP (Basic matmul_kernel)...
GPU (Basic) Execution Time: 1 ms
GPU (Basic) GFLOPS: 0.83
GPU (Basic) Validation: PASSED
-------------------------------------
Running GPU MLP (Tiled matmul_kernel_tiled)...
GPU (Tiled) Execution Time: 0.09 ms
GPU (Tiled) GFLOPS: 7.41
GPU (Tiled) Validation: PASSED
-------------------------------------
root@worker-0:/public/home/xdzs2025coffe#
```

3.4.1基础数据分析

GPU MLP (Basic matmul_kernel):

- **GPU (Basic) 执行时间:** 1 ms
- **GPU (Basic) GFLOPS:** 0.83
- GPU (Basic) 验证:

  PASSED
  - 分析:
    - 基础的GPU矩阵乘法核函数耗时1毫秒。
    - 计算性能为0.83 GFLOPS。相对于现代GPU的理论峰值性能来说，这个数值较低。这通常意味着基础核函数没有很好地利用GPU的并行能力和内存层次结构。
    - 验证通过，说明计算结果是正确的。

GPU MLP (Tiled matmul_kernel_tiled):

- **GPU (Tiled) 执行时间:** 0.09 ms

- **GPU (Tiled) GFLOPS:** 7.41
- GPU (Tiled) 验证:

  PASSED
  - 分析:
    - 使用分块优化（Tiled）的矩阵乘法核函数后，执行时间显著降低到0.09毫秒。
    - 计算性能提升到7.41 GFLOPS。
    - 验证通过，说明优化后的计算结果依然正确。

3.4.2原理分析

性能对比与结论:

优化效果显著:

分块优化的 matmul_kernel_tiled 相对于基础的 matmul_kernel 带来了巨大的性能提升。执行时间从 1 ms 降低到 0.09 ms (约 **11倍** 的提速)。

GFLOPS 从 0.83 提升到 7.41 (约 **8.9倍** 的提升)。

这清晰地展示了利用共享内存进行分块（Tiling）是GPU矩阵乘法中一种非常有效的优化手段，它能显著减少对全局内存的访问，提高数据复用率和计算效率。

3.4.2.1GPU 与 CPU (在此特定小规模问题上):

即使是优化后的GPU版本 (0.09 ms)，其执行时间也比CPU报告的0 ms要长（尽管CPU的0ms可能不完全精确）。

** 重要提示:** 对于这种总计算量非常小 (约66万FLOPs) 的问题，GPU的优势可能无法完全体现。GPU的强大之处在于其大规模并行处理能力。当问题规模（如BATCH、I、H、O）增大时，GPU相对于CPU的性能优势会更加明显。此外，数据从CPU拷贝到GPU（hipMemcpyHostToDevice）以及从GPU拷回CPU（hipMemcpyDeviceToHost）的开销，对于小问题来说，占比可能会比较大，甚至超过实际计算时间。在这个评测中，计时器主要测量的是GPU计算时间，不包括数据拷贝时间。

3.4.2.2GFLOPS 仍然有提升空间:

虽然7.41 GFLOPS远高于基础版本的0.83 GFLOPS，但对于现代DCU加速卡来说，这个数值可能仍远低于其理论双精度浮点运算峰值。这表明即使是分块优化，也可能还有进一步优化的空间，例如：

- 更细致地调整 TILE_WIDTH。
- 优化线程块和网格的配置。
- 考虑内存访问合并。
- 对于更复杂的场景，可能会使用更高级的库（如rocBLAS）。

3.4.2.3验证的重要性:

- 两次GPU运行的验证结果都是 "PASSED"，这确保了我们进行的性能优化没有引入计算错误，这是非常关键的一步。

3.4.3总结:

实验成功地展示了通过矩阵乘法优化（分块技术）可以显著提升MLP前向传播在DCU上的计算性能。对于当前的小规模问题，CPU表现也非常好。若要进一步体现GPU的优势和进行更深入的优化分析，建议在更大的矩阵维度和批处理大小上进行测试。

进阶题目2

4.1问题重述

> 完成MLP网络设计，要求能够进行前向传播，反向传播和通过梯度下降方法训练，并实现准确的LEO卫星网络下行带宽预测，需使用DCU加速卡，并对训练和推理性能进行全面的评测：
>
> 输入：每次输入t0, t0, ..., tN时刻的网络带宽值（N=10）；
>
> 输出：每次输出tN+1时刻的网络带宽值；
>
> MLP：输入层、隐藏层、输出层神经元数量和层数，以及训练参数、损失函数可自行设计优化；
>
> 数据集：一维的带宽记录，每个数据对应一个时刻的带宽值（已上传到测试环境中）；

4.2算法实现

4.2.1整体架构

这个实现包含了一个4层的深度神经网络：

- **输入层**: 10维 (INPUT_DIM)
- **隐藏层1**: 256个神经元 (HIDDEN_DIM1)
- **隐藏层2**: 128个神经元 (HIDDEN_DIM2)
- **隐藏层3**: 64个神经元 (HIDDEN_DIM3)
- **输出层**: 1维 (OUTPUT_DIM)

4.2.2核心GPU内核函数

4.2.2.1矩阵运算内核

实现矩阵乘法 C = A × B，使用2D线程块并行计算

```
__global__ void matmul_kernel(const double* A, const double* B, double* C, int M,
int N, int K)
```

4.2.2.2激活函数内核

ReLU前向传播: f(x) = max(0, x)，ReLU反向传播: 梯度传递

```
__global__ void relu_forward_kernel(double* data, int size)
__global__ void compute_relu_backward(double* delta, const double* activ, int
size)
```

4.2.2.3损失函数内核

计算MSE损失的梯度: grad = pred - target

```
__global__ void compute_output_grad(const double* pred, const double* target,
double* grad, int size)
```

4.2.3算法流程

4.2.3.1数据预处理

从文件加载带宽数据，使用滑动窗口创建时间序列数据集，Min-Max归一化到[0,1]范围，80/20划分训练/测试集

4.2.3.2前向传播

对每个样本执行以下计算序列：

```
Z1 = X × W1 + B1
A1 = ReLU(Z1)
Z2 = A1 × W2 + B2
A2 = ReLU(Z2)
Z3 = A2 × W3 + B3
A3 = ReLU(Z3)
Y_pred = A3 × W4 + B4
```

4.2.3.3反向传播

使用链式法则计算梯度：

```
dL/dY_pred = Y_pred - Y_target
dL/dW4 = A3^T × dL/dY_pred
dL/dA3 = dL/dY_pred × W4^T
dL/dZ3 = dL/dA3 ⊙ ReLU'(Z3)
... (依次向前传播)
```

4.2.3.4参数更新

使用SGD优化器：

```
weights = weights - learning_rate × gradients
```

4.3代码微调

在第一次实验代码之后我们先将以下参数设置为如下所示：

```
#define INPUT_DIM 10
#define HIDDEN_DIM 32
#define OUTPUT_DIM 1
#define BATCH_SIZE 256
#define EPOCHS 200
#define LEARNING_RATE 1e-4
```

结果如下所示：

```
[INFO] Test MSE (denormalized): 1561.8

[INFO] Sample Predictions (denormalized):
Predicted: 236.424, Actual: 259.88
Predicted: 238.928, Actual: 256.84
Predicted: 243.572, Actual: 252.1
Predicted: 242.752, Actual: 246.65
Predicted: 244.029, Actual: 258.93
Predicted: 241.823, Actual: 239.48
Predicted: 242.734, Actual: 179.27
Predicted: 234.237, Actual: 169.52
Predicted: 220.012, Actual: 174.92
Predicted: 216.524, Actual: 171.29

[INFO] Total execution time: 1 seconds.
[INFO] MLP training and inference complete.
root@worker-0:/public/home/xdzs2025coffe#
```

粘贴文本　删除单词

很显然结果并不是很好，许多数值存在着明显的偏差。所以我们对其中的许多数值进行了长期的一系列的微调。

```
#define INPUT_DIM 10
#define HIDDEN_DIM1 256
#define OUTPUT_DIM 1
#define BATCH_SIZE 256
#define EPOCHS 700
#define LEARNING_RATE 1e-3
```

```
[Epoch 689/700] Loss: 0.00493937, Time: 7 ms
[Epoch 690/700] Loss: 0.00496417, Time: 7 ms
[Epoch 691/700] Loss: 0.0049523, Time: 7 ms
[Epoch 692/700] Loss: 0.0049357, Time: 7 ms
[Epoch 693/700] Loss: 0.00495726, Time: 7 ms
[Epoch 694/700] Loss: 0.00495098, Time: 7 ms
[Epoch 695/700] Loss: 0.004945, Time: 7 ms
[Epoch 696/700] Loss: 0.00493846, Time: 7 ms
[Epoch 697/700] Loss: 0.00493935, Time: 7 ms
[Epoch 698/700] Loss: 0.00494778, Time: 7 ms
[Epoch 699/700] Loss: 0.00495005, Time: 7 ms
[Epoch 700/700] Loss: 0.00494344, Time: 7 ms

[INFO] Starting Inference on Test Set...
[INFO] Test MSE (normalized): 0.00596786
[INFO] Variance of predictions (normalized): 0.0211564
[INFO] Variance of targets (normalized): 0.025407
[INFO] Variance of errors (normalized): 0.00590557
[INFO] Test MSE (denormalized): 943.718

[INFO] Sample Predictions (denormalized):
Predicted: 220.396, Actual: 259.88
Predicted: 248.402, Actual: 256.84
Predicted: 249.511, Actual: 252.1
Predicted: 247.48, Actual: 246.65
Predicted: 242.627, Actual: 258.93
Predicted: 252.893, Actual: 239.48
Predicted: 240.914, Actual: 179.27
Predicted: 196.757, Actual: 169.52
Predicted: 182.681, Actual: 174.92
Predicted: 178.689, Actual: 171.29
```

可以看出来在我们选择输出的是个样本中，除了两个样本差别比较大，剩下的样本基本上都是一个非常接近的状态了。但是只观察是个样本未免太过于不严谨，所以我们新增加了几个评价指标对我们模型的整体性能进行评价。

4.4结果分析

4.4.1Test MSE (normalized): 0.00596786

含义: 这是在归一化数据（通常在0-1范围）上计算的均方误差。它衡量的是模型预测值与真实目标值之间平方差的平均值。数值越低越好。

评价: 这个值本身相对较小，表明在归一化的尺度上，模型的预测与实际值相当接近。这是一个积极的信号，说明模型学习到了一些数据的基本模式。

4.4.2Variance of predictions (normalized): 0.0211564

含义: 模型在测试集上做出的归一化预测值的方差。它反映了模型预测结果的分散程度或多样性。

评价: 这个值本身提供预测输出的分布信息。

4.4.3Variance of targets (normalized): 0.025407

含义: 测试集中归一化真实目标值的方差。它反映了真实数据本身固有的波动性或分散程度。

评价: 比较预测方差 (0.0211564) 和目标方差 (0.025407)：

预测值的方差略低于目标值的方差。这可能意味着模型的预测结果相比真实数据稍微平滑一些，或者未能完全捕捉到目标数据的所有波动范围，尤其可能是在极值处的预测。理想情况下，两者越接近，说明模型越能复现真实数据的分布特性。

4.4.4Variance of errors (normalized): 0.00590557

含义: 这是预测误差（预测值 - 真实值）在归一化尺度上的方差。它衡量的是模型预测误差大小的一致性。

评价: 这个值与归一化的测试MSE (0.00596786) 非常接近。MSE是误差平方的期望，而误差方差是误差减去其均值后平方的期望。两者接近通常意味着预测误差的均值接近于0，即模型没有明显的系统性偏差（总是高估或低估）。这是一个好的迹象。

4.4.5Test MSE (denormalized): 943.718

含义: 这是在原始数据尺度（去归一化后）上计算的均方误差。这个指标更具有实际业务意义，因为它直接反映了在真实单位下的预测误差。

评价:

为了更好地理解，我们可以计算均方根误差 (RMSE): `sqrt(943.718) ≈ 30.72`。这意味着，平均而言，模型的带宽预测值与实际值相差约 30.72 个单位（例如 Mbps，取决于原始数据单位）。这个误差是否可接受，取决于具体的应用场景和带宽数据的通常范围。从下面的抽样预测来看，实际带宽值在170-260之间波动，30.72的误差在这个范围内是比较显著的。

4.4.6Sample Predictions (denormalized):

含义: 展示了10个具体的去归一化后的预测值与实际值的对比。

评价:

好的预测: 例如 `Predicted: 247.48, Actual: 246.65` (误差 ≈ 0.83)，非常接近。

中等误差的预测: 例如 `Predicted: 248.402, Actual: 256.84` (误差 ≈ -8.42)。

较大误差的预测: 例如 `Predicted: 220.396, Actual: 259.88` (误差 ≈ -39.48) 或 `Predicted: 240.914, Actual: 179.27` (误差 ≈ 61.64)。后者是一个相当大的误差，模型预测值远高于实际值。

这些样本显示模型在某些情况下能够做出准确预测，但在其他情况下则偏差较大。特别是对于那些波动较大或与近期模式不符的数据点，模型可能难以准确捕捉。

4.5总结:

优点:

模型在归一化尺度上表现尚可（较低的归一化MSE），误差的均值可能接近于零（误差方差接近MSE）。模型能够捕捉到数据一定程度的变异性。

待改进/关注点:

去归一化后的MSE (943.718) 及其对应的RMSE (约30.72) 表明在实际应用中，预测误差可能仍然较大，具体取决于业务对精度的要求。样本预测显示模型在某些点上的预测性能不佳，存在较大的偏差，这可能影响模型的可靠性。预测值的方差略低于目标值的方差，可能意味着模型对数据的波动捕捉不够充分。

总体而言，模型学到了一些规律，但在预测精度和稳定性方面还有提升空间。进一步的分析可以集中在那些误差较大的样本上，尝试理解模型在何种情况下表现不佳，并据此调整模型结构、特征工程或训练策略。