《并行计算》实验报告（正文）

姓名_____学名_____完成时间____25/5/4_____

一、实验名称与内容

DCU 编程实现通用矩阵乘
内容：使用 DCU 实现矩阵乘，并分析加速比、效率等指标。

二、实验环境的配置参数

（CPU/GPU 型号与参数、内存容量与带宽、互联网络参数等）
CPU：Hygon C86 7285 32-core Processor
内存：128GB DDR4
计算网络：200Gb IB
主频：2.0GHz
显存：16GB HBM2
性能数据：FP64:10.1Tflops
加速卡：4*异构加速卡 2
每张卡 64 个 CU，每个 CU 处理 40 个波前，每个波前最多处理 64 个线程

三、实验题目问题分析

（本题目是什么类型的计算问题，可以有哪些并行化方案等）

3.1 问题分析

本题目为计算密集型的问题，数学形式为：

$$C = A \times B$$

其中$A \in \mathbb{R}^{M \times K}, B \in \mathbb{R}^{K \times N}, C \in \mathbb{R}^{M \times N}$。时间复杂度为$O(M \times N \times K)$空间复杂度为$O(M \times N + M \times K + N \times K)$。

首先要了解矩阵在内存中的组织，如图 3-1 所示，为行优先的方式。可以通过索引计算来定位矩阵中的某个元素，比如第 i 行第 j 列的元素，在线性内存中的位置：

$$\mathrm{i} * \mathrm{w} + \mathrm{j}$$

其中，w 为矩阵的宽度。

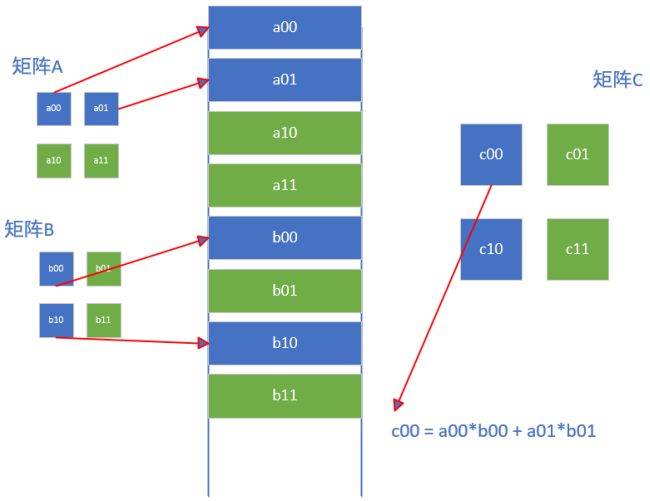

图 3-1　矩阵乘在内存中的组织

以 CPU 的串行实现为例，可以使用这种索引的方式来实现：

```
for (int m = 0; m < M; ++m) {
    for (int n = 0; n < N; ++n) {
        double sum = 0.0;
        for (int k = 0; k < K; ++k) {
            sum += A[m * K + k] * B[k * N + n];
        }
        C[m * N + n] = sum;
    }
}
```

图 3-2　CPU 矩阵乘实现

由于只有一个 CPU 线程在串行计算，所以矩阵越大耗时越久。为了优化这个过程，我们采用 GPU 来计算，GPU 有大量的线程，通过增加更多的线程来并行计算，降低运算时间。

3.2 优化方案设计

3.2.1 1 维块构建

使用 1 维块，也就是说，每个块内的线程呈 1 维的线性排列，获取每个线程的索引，如图 3-3 所示。

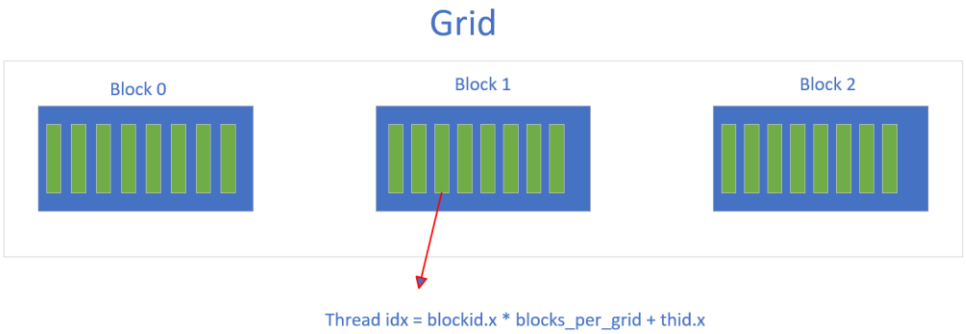

图 3-3 线程索引计算

然后要将每个线程的索引转换为矩阵的元素索引：

$$Cx = thID / wC$$
$$Cy = thID \% wC$$

然后就可以在对应的位置上计算结果了。

另外，如果不使用共享内存进行优化，那么使用 GPU 异构编程时，将矩阵拷贝到 GPU 的显存中之后，就可以像 CPU 多线程那样，之间访问对应的元素，不需要进行通信。

3.2.2 2 维块构建

计算索引的方式如图 3-4 所示，每个线程对应结果矩阵 C 中的一个元素 C[m][n]。

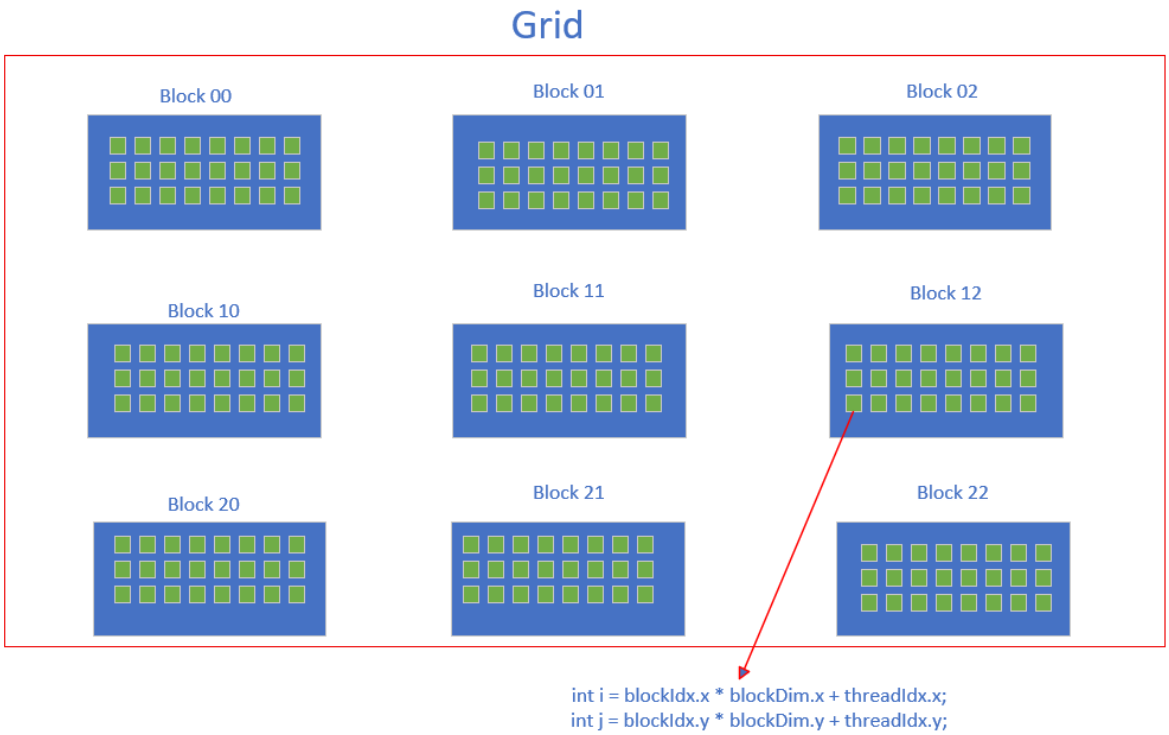

图 3-4 二维块组织

3.2.3 使用共享内存优化

使用上面的方式计算并没有有效的数据复用，在 GPU 中，共享内存是一个 block 内的所有线程共享的，此时，只需要访问一次全局内存，将数据加载到共享内存，就可以降低时间，优化性能。

假设共享内存 16KB，那就正好可以存下两个 32*32 的 double 类型的数据，对应 C 的子块。

在 A 中就是 32 行，在 B 中就是 32 列，每次加载 32 个，一次一次，将整行/整列计算完毕。

每次计算时，首先将子块加载到共享内存，然后每个线程使用共享内存的数据进行计算，最后将结果写回全局内存。

并且每一个 block 处理这样的一个子块，block 内的每个线程处理子块中的一个元素。

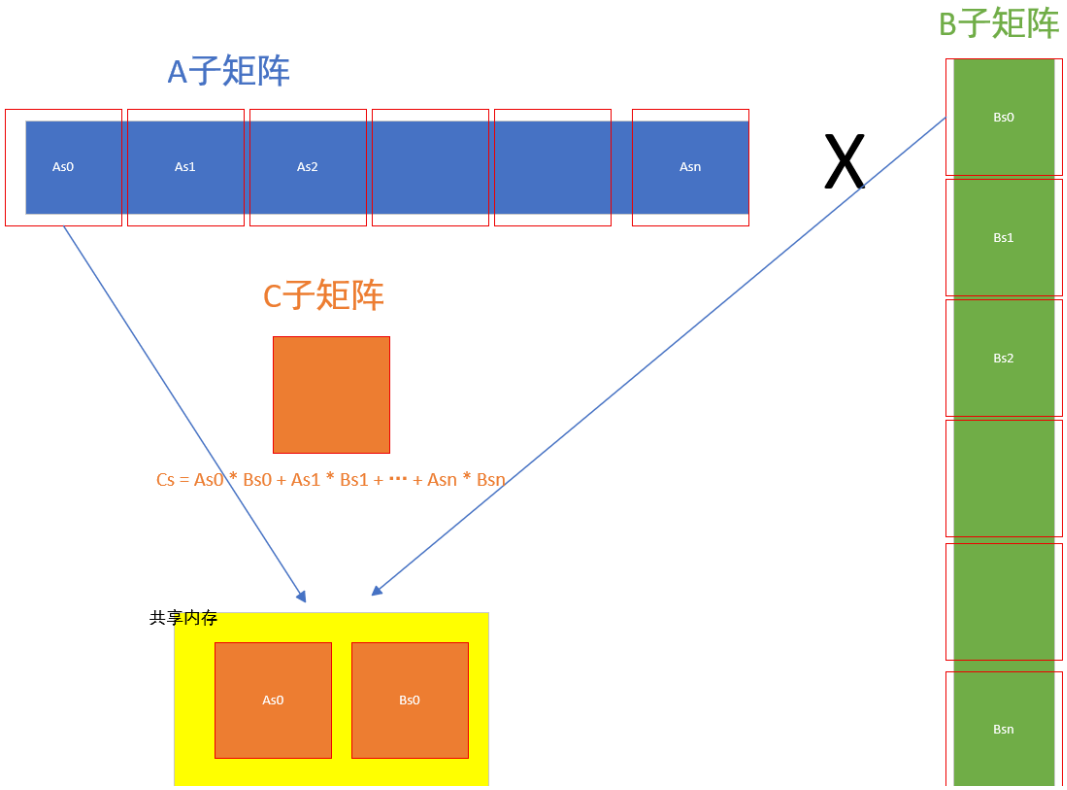

图 3-5 共享内存优化计算矩阵乘

3.3 指标计算

使用 GPU 实现矩阵乘计算加速比，效率这些指标与 CPU 的情况有些差异。

使用的基准串行时间依然是实验 2 中计算的串行时间，即使平台不同，但是为了之后的横向对比，因此，基准的串行实验依然使用实验 2 的结果，而不是在新的异构计算平台上的 CPU 串行时间。

加速比还是串行时间除以并行时间。

$$加速比 = \frac{串行时间}{并行时间}$$

3.3.1 效率

效率的计算需要用到使用到的核数，CPU 中的核数就是线程数，如果只使用 MPI，不使用多线程，那么核数就等于进程数。

在 GPU 中，情况有所不同，在 GPU 中，指令的执行最小的单位是 warp，在 DCU 中，最小单位是波前(Waves)，在本实验的平台上，一张加速卡有 64 个 CU(计算单元)，每个 CU 最多处理 40 个波前，那么一张卡最多就有 64*40＝2560 个波前，同时，每个波前最多含有 64 个线程，如果不在同一 block，那就不在同一波前。例如，有 2 个线程，在 2 个 block 中，算作 2 个波前，即使没有超过 64，如果有 128 个线程，在 1 个 block 中，就算做 2 个波前，因为都在 1 个 block 内。

因此，最后计算的结果与 2560 相比，如果大于 2560，那就使用 2560 个波前计算(因为一张卡最多 2560 个波前)，如果小于，就用实际用到的波前数。

$$效率 = \frac{加速比}{实际波前数}$$

3.3.2 每秒浮点计算数

使用 GPU 编程进行矩阵乘的实验中，也有一个指标可以关注，就是每秒计算浮点数，为便于表示通常使用 GFLOPS (= 10^9FLOPS) 和 TFLOPS (= 10^{12} FLOPS)。

要计算这个指标，就要统计核函数单独的运行时间而不考虑数据传输时间，假设这个时间为 t，对于矩阵乘，假设规模是 M，K，N，那么这个矩阵乘总共的计算数量就是

$$2 * M * N * K$$

然后使用计算量除以时间可以得到。

3.4 编程环境

平台是基于 HSA 的异构计算平台，编程模型是 HIP，这个 HIP 是一种 CUDA 兼容层，可将 CUDA 代码转换为支持 HSA 架构的代码。

因此，为了简化编程复杂性，以及在自己的电脑验证小样本，本实验的所有代码都是先采用 CUDA 编程模型，然后在平台上面通过 hipify-perl 指令，将.cu 的 CUDA 代码转换为 HIP 代码进行编译和运行。

四、方案设计

（以流程图、框图等方式给出并行方法的思路，以伪代码方式给出并行算法细节）

4.1 流程图

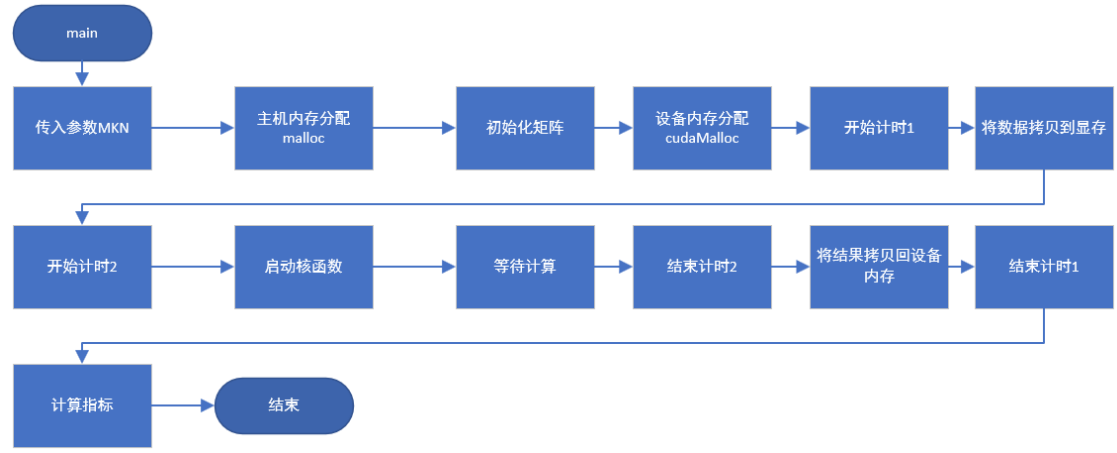

图 4-1　GPU 编程实现矩阵乘

4.2 伪代码

这一部分给出每种版本的核函数的伪代码，因为其他部分的代码较简单，容易理解，并且基本思路已经在流程图中给出。

4.2.1 1 维块

```
# 输入：A(hC×wh)，B(wh×wC)，输出：C(hC×wC)
# 每个线程可能计算多个 C 元素，通过网格跨步循环扩展
并行执行所有线程：
    total_size = wC * hC                # C 矩阵总元素数
    thID = blockIdx.x * blockDim.x + threadIdx.x   # 初始全局线程索引
    while thID < total_size:            # 网格跨步循环
        Cx = thID // wC                 # 计算 C 的行索引（整除）
        Cy = thID % wC                  # 计算 C 的列索引（取余）
        sum = 0.0
        for i in 0 到 wh-1:              # 沿 wh 维度累加点积
            a_val = A[Cx * wh + i]      # 取 A 的第 Cx 行第 i 列元素
            b_val = B[i * wC + Cy]      # 取 B 的第 i 行第 Cy 列元素
            sum += a_val * b_val
        C[Cx * wC + Cy] = sum           # 写入计算结果
        thID += gridDim.x * blockDim.x # 跨步到下一个待处理元素
```

4.2.2 2 维块

```
# 输入：A(M×K), B(K×N), 输出：C(M×N)
# 每个线程计算 C 中的一个元素 C[m][n]
```

```
并行执行所有线程：
  # 计算全局索引
  block_x = block.x      # 线程块的x坐标
  block_y = block.y      # 线程块的y坐标
  thread_x = 线程块内索引.x   # 线程在块内的x坐标
  thread_y = 线程块内索引.y   # 线程在块内的y坐标
  # 计算全局坐标
  m = block_y * 线程块高度 + thread_y   # 对应M维
  n = block_x * 线程块宽度 + thread_x   # 对应N维
  if m < M 且 n < N:
     sum = 0.0
     for k in 0 到 K-1: # 遍历K维做点积
        sum += A[m][k] * B[k][n]
     C[m][n] = sum    # 写入结果
```

4.2.3 共享内存

```
# 输入：A(M×K)，B(K×N)，输出：C(M×N)
# 每个线程块处理C的BLOCK_SIZE×BLOCK_SIZE分块，利用共享内存提升数据复用
并行执行所有线程块：
    # 共享内存声明（每个块独享）
    shared s_a[BLOCK_SIZE][BLOCK_SIZE]  # 缓存A的分块
    shared s_b[BLOCK_SIZE][BLOCK_SIZE]  # 缓存B的分块
    # 线程局部坐标
    tx = 线程块内 x 坐标
    ty = 线程块内 y 坐标
    # 当前块处理的C分块起始坐标
    bx = 块 x 索引 * BLOCK_SIZE  # 列方向起始
    by = 块 y 索引 * BLOCK_SIZE  # 行方向起始
    # 线程对应的全局坐标
    n = bx + tx  # C的列坐标
    m = by + ty  # C的行坐标
    psum = 0.0  # 局部累加器
    # 分阶段处理K维度
    for k_base in 0 到 K 步长 BLOCK_SIZE:
        # 阶段1：协作加载A的分块到 s_a ----------------------------
        a_row = by + ty         # 全局行坐标
        a_col = k_base + tx     # 全局列坐标
        if a_row < M 且 a_col < K:
            s_a[ty][tx] = A[a_row][a_col]  # 行优先加载
        else:
            s_a[ty][tx] = 0.0  # 边界填充
        # 阶段2：协作加载B的分块到 s_b ----------------------------
        b_row = k_base + ty     # 全局行坐标
        b_col = bx + tx         # 全局列坐标
        if b_row < K 且 b_col < N:
            s_b[ty][tx] = B[b_row][b_col]  # 行优先加载
```

```
        else:
            s_b[ty][tx] = 0.0  # 边界填充
        同步块内所有线程  # 等待数据加载完成
        # 阶段 3: 共享内存矩阵乘累加 ------------------------------
        for kk in 0 到 BLOCK_SIZE-1:
            # s_a 按行访问，s_b 按列访问（矩阵乘法则）
            psum += s_a[ty][kk] * s_b[kk][tx]
        同步块内所有线程  # 等待计算完成
    # 阶段 4: 结果写回全局内存 ----------------------------------
    if m < M 且 n < N:
        C[m][n] = psum
```

五、实现方法

（结合所用计算环境，给出具体的编程实现方案）

为了方便的与实验 2，3 进行对比分析，又实现了一个版本，使用 n 个 block，每个 block 仅有 1 个线程，尝试不同的 n，得到各个指标，这是因为使用 GPU 编程实现的矩阵乘一般要用到的线程数非常多，达到上千。而使用多线程，多进程的情况下，线程数，进程数很少，小于 100 个。在这种情况下，实现了一个仅使用较少线程的版本，分析其性能。

5.1 小线程版本

```cpp
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define TESTCOUNT 1024
void InitMat(double *data, int size)
{
    srand(time(NULL));
    for (int i = 0; i < size; ++i) {
        data[i] = (rand() % 10) / 10.0;;
    }
}
__global__ void MatMulKernel1D(double *C, double *A, double *B, int wh, int wC, int hC)
{
    int totalSize = wC * hC;
    int thID = threadIdx.x + blockIdx.x * blockDim.x; // 索引计算
    while (thID < totalSize) {
        int Cx = thID / wC;             //数据坐标 与 thread索引的映射
        int Cy = thID % wC;
        float rst = 0.0;
        #pragma unroll
        for (int i = 0; i < wh; i++) {
            rst += A[Cx * wh + i] * B[i * wC + Cy];
        }
        C[Cx * wC + Cy] = rst;
        thID += gridDim.x * blockDim.x;
    }
}
```

```
67  int main(int argc, char **argv) {
68      // 矩阵维度设置
69      int hC = 256;  // 矩阵A的行数，结果矩阵C的行数
70      int wh = 256;   // 矩阵A的列数，矩阵B的行数
71      int wC = 256;  // 矩阵B的列数，结果矩阵C的列数
72
73      struct timespec time_start, time_end;
74      struct timespec time_start1, time_end1;
75      //printf("Entering matrix multiplication...\n");
76      if (argc != 4) {
77          fprintf(stderr, "Usage: %s <M> <K> <N>\n", argv[0]);
78          exit(0);
79      }
80      hC = atoi(argv[1]);
81      wh = atoi(argv[2]);
82      wC = atoi(argv[3]);
83      //printf("");
84      // 主机内存分配
85      size_t size_A = hC * wh * sizeof(double);
86      size_t size_B = wh * wC * sizeof(double);
87      size_t size_C = hC * wC * sizeof(double);
88
89      double *h_A = (double *)malloc(size_A);
90      double *h_B = (double *)malloc(size_B);
91      double *h_C = (double *)malloc(size_C);
92      double *h_C_ref = (double *)malloc(size_C);
93
94      // 初始化数据
95      InitMat(h_A, hC * wh);
96      InitMat(h_B, wh * wC);
```

```
99    // 设备内存分配
100   double *d_A, *d_B, *d_C;
101   cudaMalloc(&d_A, size_A);
102   cudaMalloc(&d_B, size_B);
103   cudaMalloc(&d_C, size_C);
104
105   //clock_gettime(CLOCK_MONOTONIC, &time_start);
106   //double sum_elapsed1 = 0;
107
108   for(int q = 1; q <= TESTCOUNT; q*=2){
109       clock_gettime(CLOCK_MONOTONIC, &time_start);
110
111       // 数据拷贝到设备
112       cudaMemcpy(d_A, h_A, size_A, cudaMemcpyHostToDevice);
113       cudaMemcpy(d_B, h_B, size_B, cudaMemcpyHostToDevice);
114
115       // 执行核函数
116       const int threadsPerBlock = 1;
117
118       int blocksPerGrid = q;
119       clock_gettime(CLOCK_MONOTONIC, &time_start1);
120       MatMulKernel1D<<<blocksPerGrid, threadsPerBlock>>>(d_C, d_A, d_B, wh, wC, hC);
121
122       // 检查核函数执行错误
123       cudaError_t err = cudaGetLastError();
124       if (err != cudaSuccess) {
125           fprintf(stderr, "Kernel execution error: %s\n", cudaGetErrorString(err));
126           exit(EXIT_FAILURE);
127       }
128       cudaDeviceSynchronize();
129       clock_gettime(CLOCK_MONOTONIC, &time_end1);
```

```
        // 拷贝结果回主机
        cudaMemcpy(h_C, d_C, size_C, cudaMemcpyDeviceToHost);
        clock_gettime(CLOCK_MONOTONIC, &time_end);
        double elapsed = ((time_end.tv_sec - time_start.tv_sec) + (time_end.tv_nsec - time_start.tv_nsec) / 1e9 );
        double elapsed1 = ((time_end1.tv_sec - time_start1.tv_sec) + (time_end1.tv_nsec - time_start1.tv_nsec) / 1e9);
        double speedup = getser_time(hC) / elapsed;
        int sps = (threadsPerBlock * blocksPerGrid) > 2560 ? 2560 :  (threadsPerBlock * blocksPerGrid);

        double eff = speedup / sps;
        double computing = 2.0 * hC * hC * hC;
        double Gflops = computing / elapsed1 / 1000000000;
        printf("1dnoshare,%d,%d,%d,%d,%f,%f,%f,%f\n",hC,wh,wC,sps,elapsed,speedup,eff,Gflops);

    }
    // 资源释放
    free(h_A);
    free(h_B);
    free(h_C);
    free(h_C_ref);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    return 0;
}
```

```
double getser_time(int hc){
    if(hc == 100)
        return 0.015778;
    if(hc == 200)
        return 0.244741;
    if(hc == 300)
        return 0.349207;
    if(hc == 400)
        return 0.730528;
    if(hc == 500)
        return 1.651484;
    if(hc == 600)
        return 4.098283;
    if(hc == 700)
        return 6.249801;
    if(hc == 800)
        return 10.354726;
    if(hc == 900)
        return 13.621626;
    if(hc == 1000)
        return 20.700472;
    if(hc == 2000)
        return 158.748486;
    if(hc == 3000)
        return 546.788188;
    if(hc == 4000)
        return 1315.157002;
    if(hc == 5000)
        return 2613.591440;
    if(hc == 6000)
        return 4570.909423;
    if(hc == 7000)
        return 6941.371477;
    if(hc == 8000)
        return 11169.167989;
```

5.2 1 维块

```cpp
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


#define TESTCOUNT 20


void InitMat(double *data, int size)
{
    srand(time(NULL));
    for (int i = 0; i < size; ++i) {
        data[i] = (rand() % 10) / 10.0;;
    }
}


__global__ void MatMulKernel1D(double *C, double *A, double *B, int wh, int wC, int hC)
{
    int totalSize = wC * hC;
    int thID = threadIdx.x + blockIdx.x * blockDim.x; // 索引计算
    while (thID < totalSize) {
        int Cx = thID / wC;             //数据坐标 与 thread索引的映射
        int Cy = thID % wC;
        double rst = 0.0;
        #pragma unroll
        for (int i = 0; i < wh; i++) {
            rst += A[Cx * wh + i] * B[i * wC + Cy];
        }
        C[Cx * wC + Cy] = rst;
        thID += gridDim.x * blockDim.x;
    }
}
```

```
109  int main(int argc, char **argv) {
110      // 矩阵维度设置
111      int hC = 256;  // 矩阵A的行数，结果矩阵C的行数
112      int wh = 256;   // 矩阵A的列数，矩阵B的行数
113      int wC = 256;  // 矩阵B的列数，结果矩阵C的列数
114
115      struct timespec time_start, time_end;
116      struct timespec time_start1, time_end1;
117      //printf("Entering matrix multiplication...\n");
118      if (argc != 4) {
119          fprintf(stderr, "Usage: %s <M> <K> <N>\n", argv[0]);
120          exit(0);
121      }
122      hC = atoi(argv[1]);
123      wh = atoi(argv[2]);
124      wC = atoi(argv[3]);
125      printf("1dnoshare,%d,%d,%d,", hC,wh,wC);
126      // 主机内存分配
127      size_t size_A = hC * wh * sizeof(double);
128      size_t size_B = wh * wC * sizeof(double);
129      size_t size_C = hC * wC * sizeof(double);
130
131      double *h_A = (double *)malloc(size_A);
132      double *h_B = (double *)malloc(size_B);
133      double *h_C = (double *)malloc(size_C);
134      double *h_C_ref = (double *)malloc(size_C);
135
136      // 初始化数据
137      InitMat(h_A, hC * wh);
138      InitMat(h_B, wh * wC);
```

```
141      // 设备内存分配
142      double *d_A, *d_B, *d_C;
143      cudaMalloc(&d_A, size_A);
144      cudaMalloc(&d_B, size_B);
145      cudaMalloc(&d_C, size_C);
146
147      clock_gettime(CLOCK_MONOTONIC, &time_start);
148      double sum_elapsed1 = 0;
149
150      for(int q = 0; q < TESTCOUNT; q++){
151
152
153          // 数据拷贝到设备
154          cudaMemcpy(d_A, h_A, size_A, cudaMemcpyHostToDevice);
155          cudaMemcpy(d_B, h_B, size_B, cudaMemcpyHostToDevice);
156
157          // 执行核函数
158          const int threadsPerBlock = 256;
159          int totalElements = hC * wC;
160          int blocksPerGrid = (totalElements + threadsPerBlock - 1) / threadsPerBlock;
161          clock_gettime(CLOCK_MONOTONIC, &time_start1);
162          MatMulKernel1D<<<blocksPerGrid, threadsPerBlock>>>(d_C, d_A, d_B, wh, wC, hC);
163
164          // 检查核函数执行错误
165          cudaError_t err = cudaGetLastError();
166          if (err != cudaSuccess) {
167              fprintf(stderr, "Kernel execution error: %s\n", cudaGetErrorString(err));
168              exit(EXIT_FAILURE);
169          }
170          cudaDeviceSynchronize();
171          clock_gettime(CLOCK_MONOTONIC, &time_end1);
172          sum_elapsed1 +=((time_end1.tv_sec - time_start1.tv_sec) + (time_end1.tv_nsec - time_start1.tv_nsec) / 1e9);
```

```
        // 拷贝结果回主机
        cudaMemcpy(h_C, d_C, size_C, cudaMemcpyDeviceToHost);
    }
    clock_gettime(CLOCK_MONOTONIC, &time_end);

    double elapsed = ((time_end.tv_sec - time_start.tv_sec) + (time_end.tv_nsec - time_start.tv_nsec) / 1e9 )/ 20;
    double elapsed1 = sum_elapsed1 / 20;
    double speedup = getser_time(hC) / elapsed;

    const int threadsPerBlock = 256;
    int totalElements = hC * wC;
    int blocksPerGrid = (totalElements + threadsPerBlock - 1) / threadsPerBlock;
    int wave = 4 * blocksPerGrid;


    int core = wave > 2560 ? 2560 :  wave;

    double eff = speedup / core;
    double computing = 2.0 * hC * hC * hC;
    double Gflops = computing / elapsed1 / 1000000000;
    printf("%d,%f,%f,%f,%f\n",core,elapsed,speedup,eff,Gflops);

    // 资源释放
    free(h_A);
    free(h_B);
    free(h_C);
    free(h_C_ref);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

5.3 2 维块

```cpp
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void InitMat(double *data, int size)
{
    srand(time(NULL));
    for (int i = 0; i < size; ++i) {
        data[i] = (rand() % 10) / 10.0;;
    }
}

#define OFFSET(row, col, ld) ((row) * (ld) + (col))
__global__ void naiveDgemm(
    double * __restrict__ a, double * __restrict__ b, double * __restrict__ c,
    const int M, const int N, const int K) {

    int n = blockIdx.x * blockDim.x + threadIdx.x;
    int m = blockIdx.y * blockDim.y + threadIdx.y;
    if (m < M && n < N) {
        double psum = 0.0;
        #pragma unroll
        for (int k = 0; k < K; k++) {
            psum += a[OFFSET(m, k, K)] * b[OFFSET(k, n, N)];
        }
        c[OFFSET(m, n, N)] = psum;
    }
}

```

```cpp
int main(int argc, char **argv) {
    // 矩阵维度设置
    int M = 256;  // 矩阵A的行数，结果矩阵C的行数
    int K = 256;  // 矩阵A的列数，矩阵B的行数
    int N = 256;  // 矩阵B的列数，结果矩阵C的列数

    struct timespec time_start, time_end;
    struct timespec time_start1, time_end1;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s <M> <K> <N>\n", argv[0]);
        exit(0);
    }
    M = atoi(argv[1]);
    K = atoi(argv[2]);
    N = atoi(argv[3]);
    printf("2dnoshare,%d,%d,%d,", M, K, N);

    // 主机内存分配
    size_t size_A = M * K * sizeof(double);
    size_t size_B = K * N * sizeof(double);
    size_t size_C = M * N * sizeof(double);

    double *h_A = (double *)malloc(size_A);
    double *h_B = (double *)malloc(size_B);
    double *h_C = (double *)malloc(size_C);
    double *h_C_ref = (double *)malloc(size_C);

    // 初始化双精度数据
    InitMat(h_A, M * K);
    InitMat(h_B, K * N);
```

```cpp
    // 设备内存分配
    double *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size_A);
    cudaMalloc(&d_B, size_B);
    cudaMalloc(&d_C, size_C);

    clock_gettime(CLOCK_MONOTONIC, &time_start);
    double sum_elapsed1 = 0;

    // 定义测试次数常量
    const int TESTCOUNT = 20;
    for(int q = 0; q < TESTCOUNT; q++){
        // 数据拷贝到设备
        cudaMemcpy(d_A, h_A, size_A, cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B, size_B, cudaMemcpyHostToDevice);

        // 核函数配置
        dim3 blockSize(16, 16);  // 256 threads per block
        dim3 gridSize((N + blockSize.x - 1) / blockSize.x,
                      (M + blockSize.y - 1) / blockSize.y);

        // 执行双精度核函数
        clock_gettime(CLOCK_MONOTONIC, &time_start1);

        naiveDgemm<<<gridSize, blockSize>>>(d_A, d_B, d_C, M, N, K);
```

```
        cudaError_t err = cudaGetLastError();
        if (err != cudaSuccess) {
            fprintf(stderr, "Kernel execution error: %s\n", cudaGetErrorString(err));
            exit(EXIT_FAILURE);
        }
        cudaDeviceSynchronize();
        clock_gettime(CLOCK_MONOTONIC, &time_end1);
        sum_elapsed1 += (time_end1.tv_sec - time_start1.tv_sec) +
                        (time_end1.tv_nsec - time_start1.tv_nsec) / 1e9;

        cudaMemcpy(h_C, d_C, size_C, cudaMemcpyDeviceToHost);
    }
    clock_gettime(CLOCK_MONOTONIC, &time_end);

    // 性能计算
    double elapsed = ((time_end.tv_sec - time_start.tv_sec) +
                      (time_end.tv_nsec - time_start.tv_nsec) / 1e9) / TESTCOUNT;
    double elapsed1 = sum_elapsed1 / TESTCOUNT;
    double computing_ops = 2.0 * M * N * K;  // 正确计算运算量
    double Gflops = computing_ops / elapsed1 / 1e9;

    dim3 blockSize(16, 16);  // 256 threads per block
    dim3 gridSize((N + blockSize.x - 1) / blockSize.x,
                  (M + blockSize.y - 1) / blockSize.y);

    int active_blocks = gridSize.x * gridSize.y;
    int wave_size = 4 * active_blocks;
    int max_cores = 2560;
    int used_cores = (wave_size > max_cores) ? max_cores : wave_size;
    double speedup = getser_time(M) / elapsed;
    double efficiency = speedup / used_cores;
```

```
    // 输出结果
    printf("%d,%f,%f,%f,%f\n",used_cores,elapsed,speedup,efficiency,Gflops);

    // 资源释放
    free(h_A);
    free(h_B);
    free(h_C);
    free(h_C_ref);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

5.4 共享内存

```cpp
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void InitMat(double *data, int size)
{
    srand(time(NULL));
    for (int i = 0; i < size; ++i) {
        data[i] = (rand() % 10) / 10.0;;
    }
}

#define BLOCK_SIZE 32  // 每个块使用 2*(32*32)*8B = 16KB 共享内存

__global__ void __launch_bounds__(1024) sharedMemDgemm(
    double * __restrict__ a,
    double * __restrict__ b,
    double * __restrict__ c,
    const int M, const int N, const int K)
{
    // 定义共享内存块（按行优先存储）
    __shared__ double s_a[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double s_b[BLOCK_SIZE][BLOCK_SIZE];

    // 线程在块内的局部坐标
    const int tx = threadIdx.x;
    const int ty = threadIdx.y;

    // 计算当前块处理的C矩阵坐标范围
    const int bx = blockIdx.x * BLOCK_SIZE;  // 列方向起始位置
    const int by = blockIdx.y * BLOCK_SIZE;  // 行方向起始位置

    // 当前线程处理的全局坐标
    const int n = bx + tx;  // 列坐标
    const int m = by + ty;  // 行坐标
```

```
    double psum = 0.0;

    // 分阶段加载数据并计算
    for (int k_base = 0; k_base < K; k_base += BLOCK_SIZE) {
        // 协作加载A的子块到共享内存 ----------------------------------------
        int a_col = k_base + tx;                    // A的列坐标
        int a_row = by + ty;                        // A的行坐标
        if (a_row < M && a_col < K) {
            s_a[ty][tx] = a[a_row * K + a_col];     // 行优先访问
        } else {
            s_a[ty][tx] = 0.0;  // 越界填充零
        }

        // 协作加载B的子块到共享内存 ----------------------------------------
        int b_row = k_base + ty;                    // B的行坐标
        int b_col = bx + tx;                        // B的列坐标
        if (b_row < K && b_col < N) {
            s_b[ty][tx] = b[b_row * N + b_col];     // 行优先访问
        } else {
            s_b[ty][tx] = 0.0;  // 越界填充零
        }
        void __syncthreads()
        __syncthreads();  // 等待块内所有线程完成数据加载

        // 计算当前分块的矩阵乘累加 ----------------------------------------
        #pragma unroll
        for (int kk = 0; kk < BLOCK_SIZE; ++kk) {
            // s_a按行访问，s_b按列访问（通过kk遍历共享内存的行）
            psum += s_a[ty][kk] * s_b[kk][tx];
        }

        __syncthreads();  // 等待计算完成再加载下一批数据
    }

    // 将结果写回全局内存 ------------------------------------------------
    if (m < M && n < N) {
        c[m * N + n] = psum;
    }
}
```

```cpp
int main(int argc, char **argv) {
    // 矩阵维度设置
    int M = 256;  // 矩阵A的行数，结果矩阵C的行数
    int K = 256;  // 矩阵A的列数，矩阵B的行数
    int N = 256;  // 矩阵B的列数，结果矩阵C的列数

    struct timespec time_start, time_end;
    struct timespec time_start1, time_end1;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s <M> <K> <N>\n", argv[0]);
        exit(0);
    }
    M = atoi(argv[1]);
    K = atoi(argv[2]);
    N = atoi(argv[3]);
    printf("2dshare,%d,%d,%d,", M, K, N);

    // 主机内存分配
    size_t size_A = M * K * sizeof(double);
    size_t size_B = K * N * sizeof(double);
    size_t size_C = M * N * sizeof(double);

    double *h_A = (double *)malloc(size_A);
    double *h_B = (double *)malloc(size_B);
    double *h_C = (double *)malloc(size_C);
    double *h_C_ref = (double *)malloc(size_C);

    // 初始化双精度数据
    InitMat(h_A, M * K);
    InitMat(h_B, K * N);

    // 设备内存分配
    double *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size_A);
    cudaMalloc(&d_B, size_B);
    cudaMalloc(&d_C, size_C);

    clock_gettime(CLOCK_MONOTONIC, &time_start);
    double sum_elapsed1 = 0;

    // 定义测试次数常量
    const int TESTCOUNT = 20;
    for(int q = 0; q < TESTCOUNT; q++){
        // 数据拷贝到设备
        cudaMemcpy(d_A, h_A, size_A, cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B, size_B, cudaMemcpyHostToDevice);

        // 核函数配置
        dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);  // 256 threads per block
        dim3 gridSize((N + blockSize.x - 1) / blockSize.x,
                      (M + blockSize.y - 1) / blockSize.y);

        // 执行双精度核函数
        clock_gettime(CLOCK_MONOTONIC, &time_start1);

        sharedMemDgemm<<<gridSize, blockSize>>>(d_A, d_B, d_C, M, N, K);
```

```
        cudaError_t err = cudaGetLastError();
        if (err != cudaSuccess) {
            fprintf(stderr, "Kernel execution error: %s\n", cudaGetErrorString(err));
            exit(EXIT_FAILURE);
        }
        cudaDeviceSynchronize();
        clock_gettime(CLOCK_MONOTONIC, &time_end1);
        sum_elapsed1 += (time_end1.tv_sec - time_start1.tv_sec) +
                        (time_end1.tv_nsec - time_start1.tv_nsec) / 1e9;

        cudaMemcpy(h_C, d_C, size_C, cudaMemcpyDeviceToHost);
    }
    clock_gettime(CLOCK_MONOTONIC, &time_end);

    // 性能计算
    double elapsed = ((time_end.tv_sec - time_start.tv_sec) +
                      (time_end.tv_nsec - time_start.tv_nsec) / 1e9) / TESTCOUNT;
    double elapsed1 = sum_elapsed1 / TESTCOUNT;
    double computing_ops = 2.0 * M * N * K;  // 正确计算运算量
    double Gflops = computing_ops / elapsed1 / 1e9;

    dim3 blockSize(16, 16);  // 256 threads per block
    dim3 gridSize((N + blockSize.x - 1) / blockSize.x,
                  (M + blockSize.y - 1) / blockSize.y);

    int active_blocks = gridSize.x * gridSize.y;
    int wave_size = 4 * active_blocks;
    int max_cores = 2560;
    int used_cores = (wave_size > max_cores) ? max_cores : wave_size;
    double speedup = getser_time(M) / elapsed;
    double efficiency = speedup / used_cores;

    // 输出结果
    printf("%d,%f,%f,%f,%f\n",used_cores,elapsed,speedup,efficiency,Gflops);

    // 资源释放
    free(h_A);
    free(h_B);
    free(h_C);
    free(h_C_ref);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

六、结果分析

（在结果正确的前提下，分析所实现方案的加速比、效率等指标）

将. cu 文件传入平台后，使用 hipify-perl 指令转换为 mat1.cpp，然后使用模板提交作业：

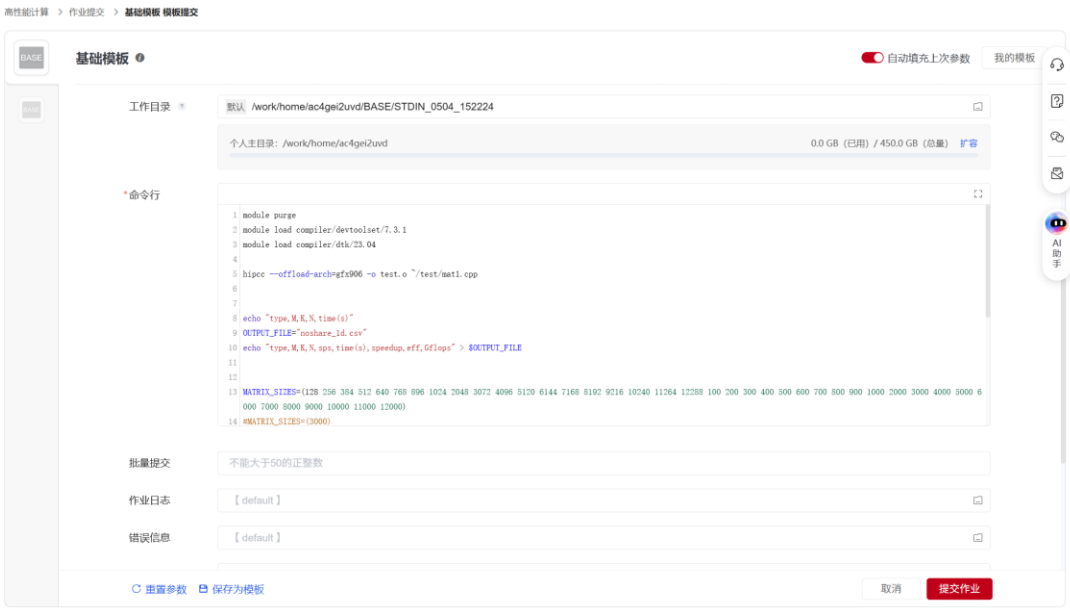

图 6-1　作业提交

得到的结果如下：

```
type,M,K,N,sps,time(s),speedup,eff,Gflops
2dshare,128,128,128,256,0.000456,73.570383,0.287384,35.284561
2dshare,256,256,256,1024,0.000738,382.570729,0.373604,204.039832
2dshare,384,384,384,2304,0.001458,672.555085,0.291908,392.980894
2dshare,512,512,512,2560,0.002153,1102.275609,0.430576,508.812449
2dshare,640,640,640,2560,0.003080,1528.873948,0.597216,635.597255
2dshare,768,768,768,2560,0.004176,1974.597819,0.771327,761.237580
2dshare,896,896,896,2560,0.005595,2366.283809,0.924330,791.570792
2dshare,1024,1024,1024,2560,0.007429,2686.036654,1.049233,825.463210
2dshare,2048,2048,2048,2560,0.034573,4853.495948,1.895897,981.829814
2dshare,3072,3072,3072,2560,0.095249,6121.777326,2.391319,1023.192897
2dshare,4096,4096,4096,2560,0.199628,7068.461260,2.761118,1043.982898
2dshare,5120,5120,5120,2560,0.360385,7771.128961,3.035597,1058.137903
2dshare,6144,6144,6144,2560,0.589850,8312.887320,3.247222,1068.029361
2dshare,7168,7168,7168,2560,0.898840,8759.312310,3.421606,1083.495128
2dshare,8192,8192,8192,2560,1.328646,8930.842382,3.488610,1056.585991
2dshare,9216,9216,9216,2560,1.805935,9434.930253,3.685520,1090.764419
```

图 6-2　测试结果

6.1 小线程版本

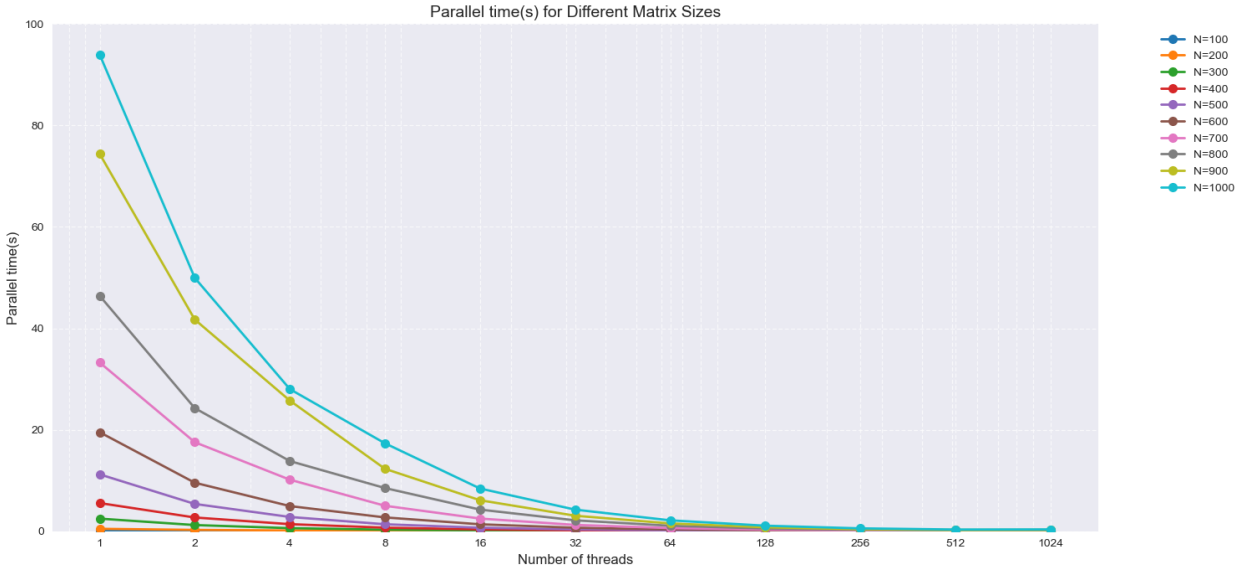

图 6-3　不同规模矩阵不同线程数执行时间

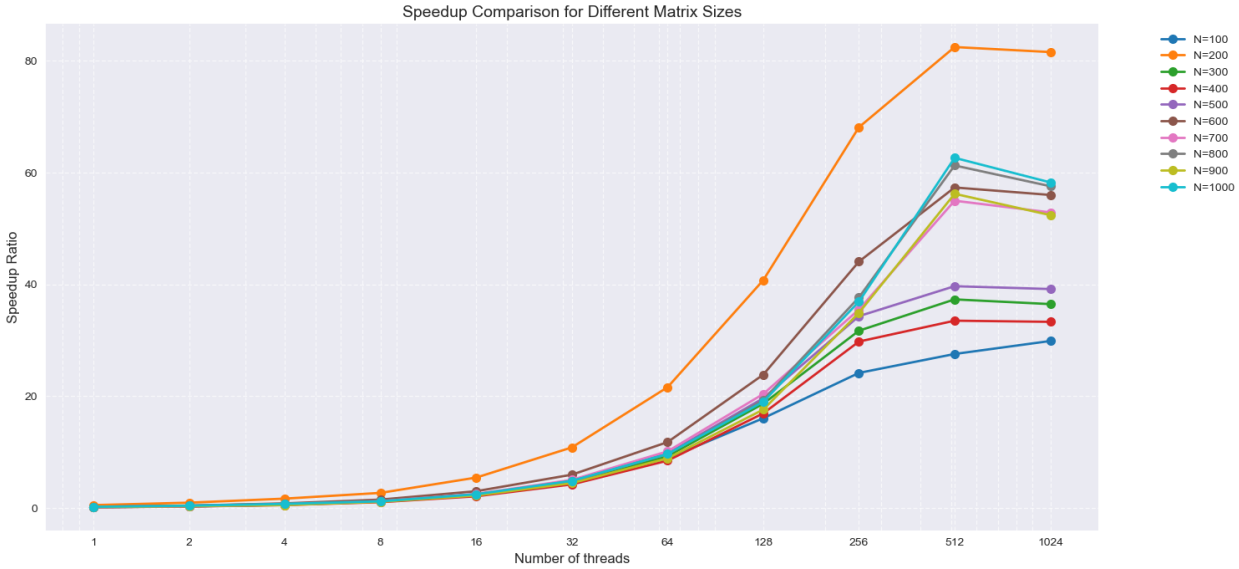

图 6-4　不同规模矩阵不同线程数加速比

由图 6-4 所示，即使线程数已经很高，继续扩大线程数依然可以提高加速比，直到 1024 线程，加速比不再明显上升。

这表明 GPU 的并行计算架构对于矩阵乘法这种可以高度并行化的任务非常有效，在较低线程数时，GPU 的计算资源未得到充分利用，增加线程数能够让更多的计算单元同时工作，从而有效提高矩阵乘法的计算速度，当线程数达到 1024 时，加速比不再明显上升，说明此时 GPU 的计算资源已接近饱和。

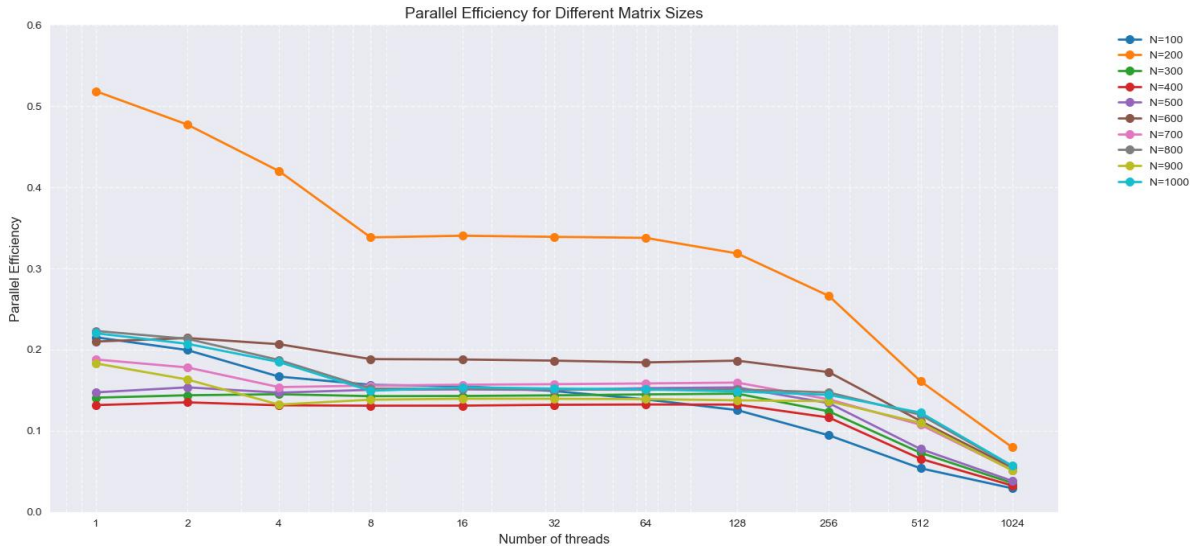

图 6-5　不同规模矩阵不同线程数效率

由图 6-5 所示，随着线程数增加，效率比较平缓，直到 256 线程增加到 512 以及继续增加到 1024 时，效率明显下降。在每个 Block 仅 1 个线程的情况下，效率很低，这是因为无法利用每个 block 内多线程的并行，DCU 的线程调度单位是波前，每个波前 64 个线程，这里仅有 1 个线程，导致很多资源被浪费。

256 线程增加到 512 以及继续增加到 1024 时，效率明显下降是因为，GPU 的流多处理器（SM）可并行处理的 Block 数量有限，在 DCU 中，每个 CU 有 4 个 SM，统共 64 个 CU，即 256 个 SM，因此继续增加 block 效率降低很快。

6.2 1 维块

从这里开始，后面画的图都是以矩阵规模为自变量，这是因为要最大的发挥 DCU 的并行潜力，那就要尽量大的使用线程以及 block，因此，几乎每种情况使用到的波前数都是 2560 个，这种情况下，再以线程数为自变量的意义不大。

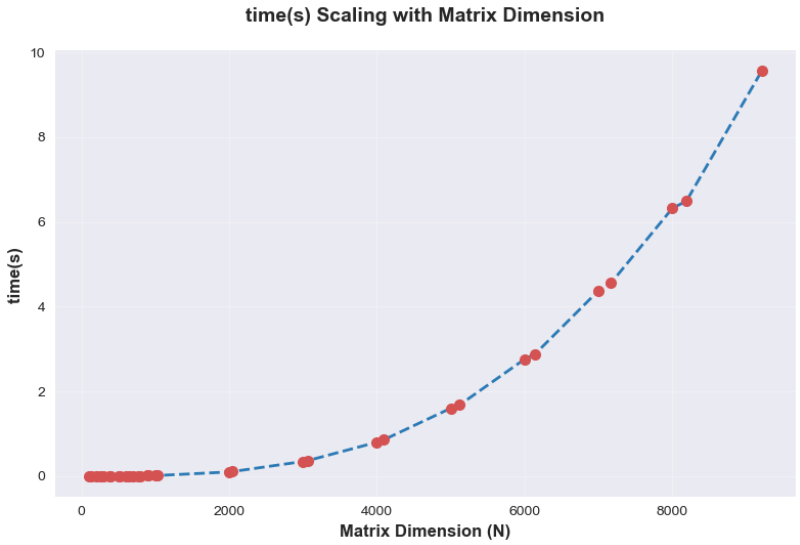

图 6-7 不同规模运行时间

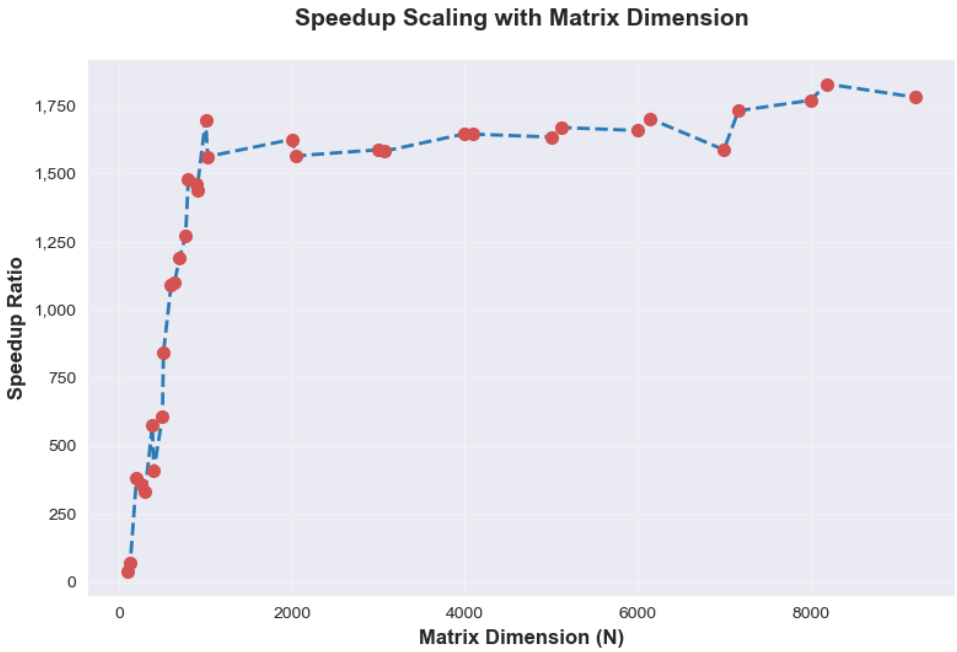

图 6-6 不同规模加速比

由图所示，随着矩阵规模增大，加速比越来越大，并在规模达到 2000 左右达到饱和。体现出 GPU 并行计算在处理大规模矩阵乘法时的显著优势。矩阵乘法运算具有高度的并行性，GPU 拥有众多计算核心，可以同时处理大量矩阵元素的计算。当矩阵规模较小时，GPU 的部分计算资源可能处于闲置状态；而随着矩阵规模增加，更多的计算任务能被并行执行，从而有效加快计算速度，使得加速比增大。

当矩阵规模达到 2000 左右时加速比饱和，说明 GPU 的并行计算能力存在一定的上限，比如最多只有 2560 个波前，只能同时处理一定的数据量，如果规模过大，那么后面的任务就要排队，不能有效的增加加速比。

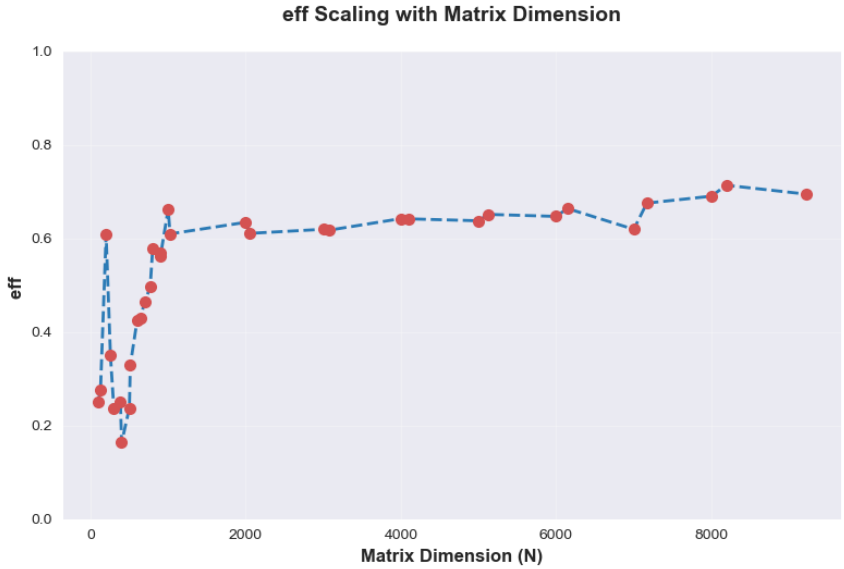

图 6-7 不同规模效率

效率随矩阵规模增大而呈提升的趋势，表明大规模计算能更高效地利用资源，并且在规模很大时，效率可以得到保持，并且此时用到的波前数也达到了单块加速卡的上限。

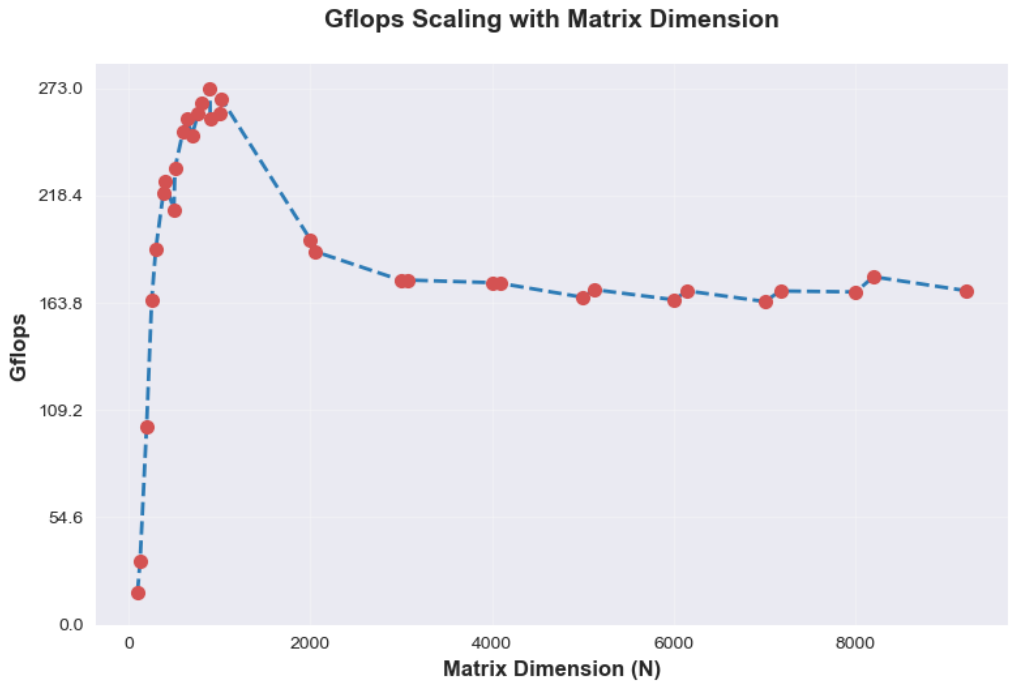

图 6-7 不同规模的每秒计算浮点数

Gflops 随规模先增后降，较小规模矩阵达到峰值 272.7 Gflops，而超大矩阵降至 177 Gflops。这说明，较小规模下，计算与内存访问比最优，计算单元满载，超大规模时，数据搬运时间占比增加，内存带宽成为瓶颈，导致实际计算量下降。

6.3 2 维块

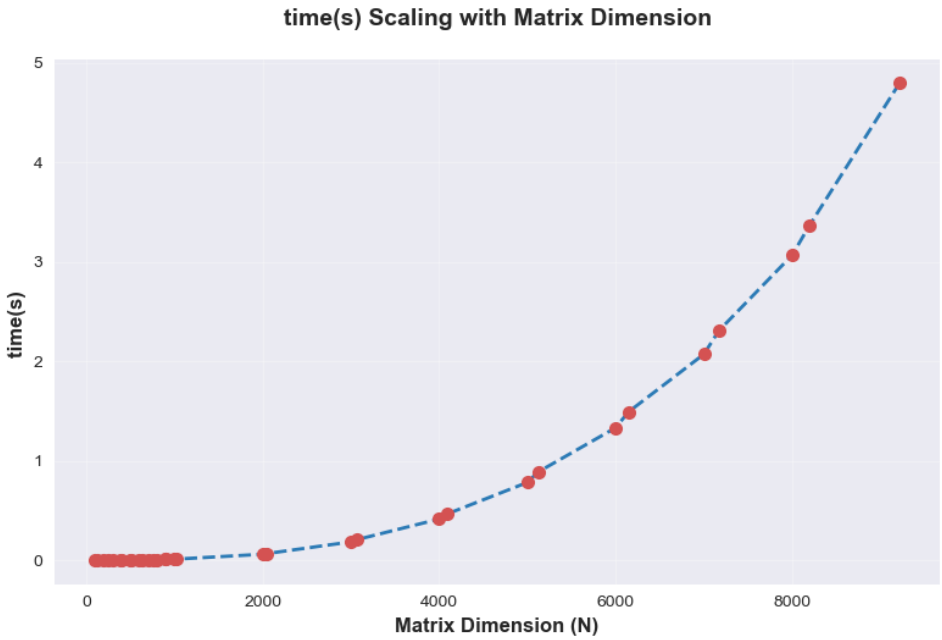

图 6-8 二维块不同规模下的运行时间

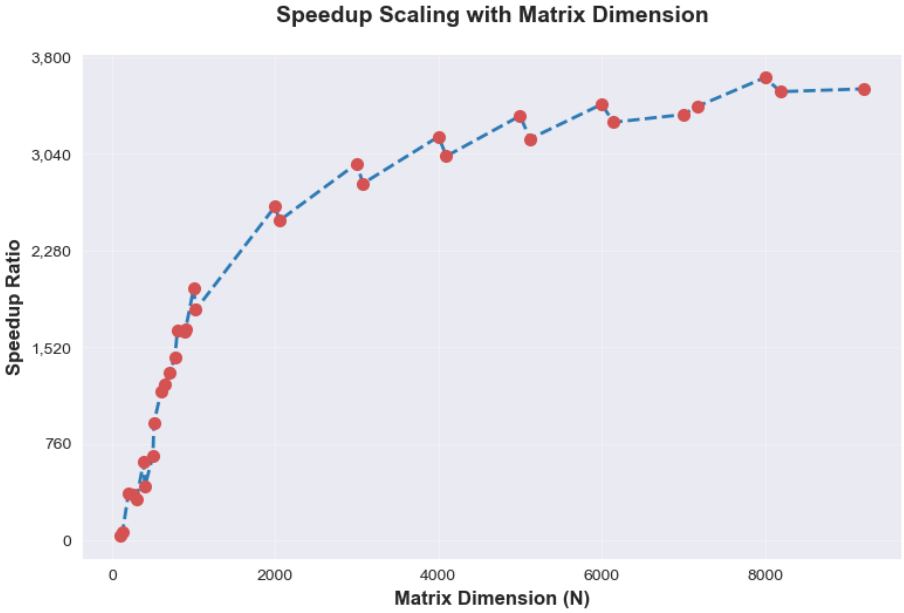

图 6-9 二维块不同规模下的加速比

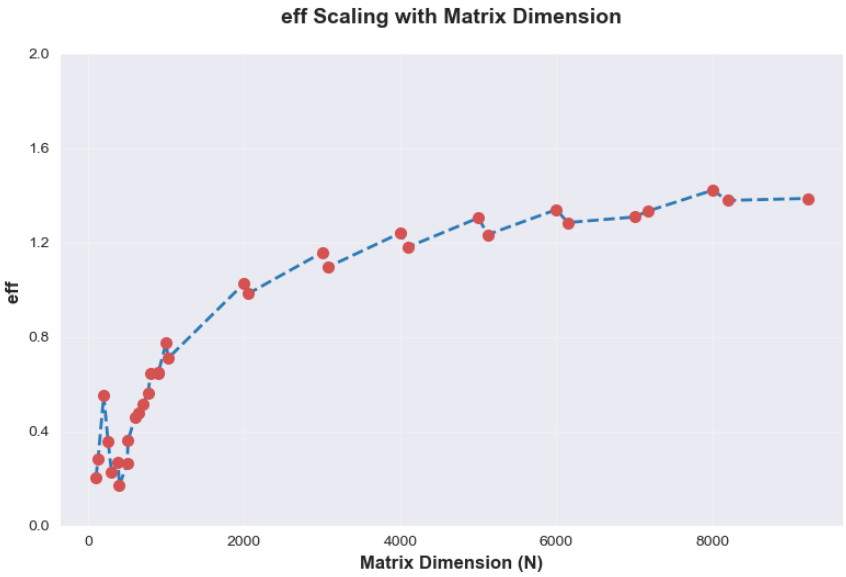

图 6-9 二维块不同规模下的效率

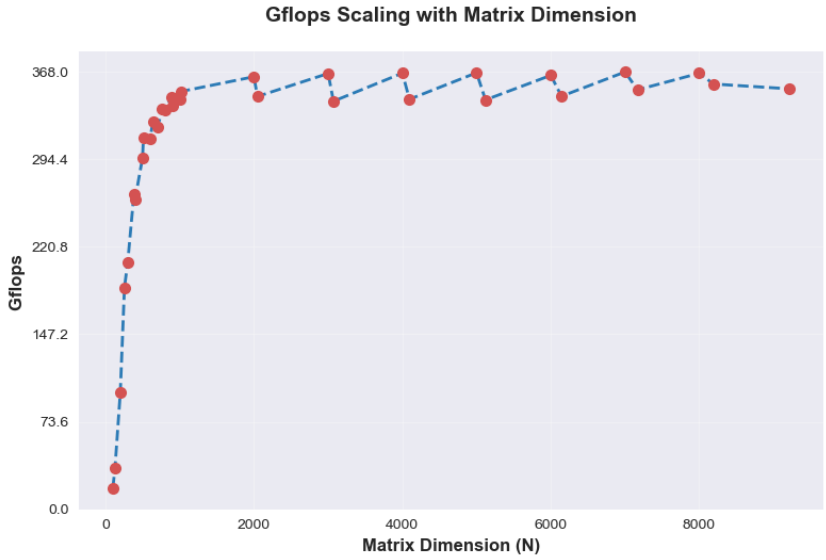

图 6-10 二维块不同规模下的每秒浮点计算数

在 8192x8192 矩阵下，二维块的加速比和效率与 1 维块想比均翻倍，GFlops 几近翻倍，表明二维块能更高效利用 GPU 资源。

在 4096x4096 及以上规模，二维块效率超 1，表明二维块即使不适用共享内存优化，也可以减少访问全局内存的次数，达到优化效果。

并且二维块的 Gflops 在矩阵规模较大时，一直保持较高水平，说明显存访问确实存在优化。

6.4 共享内存

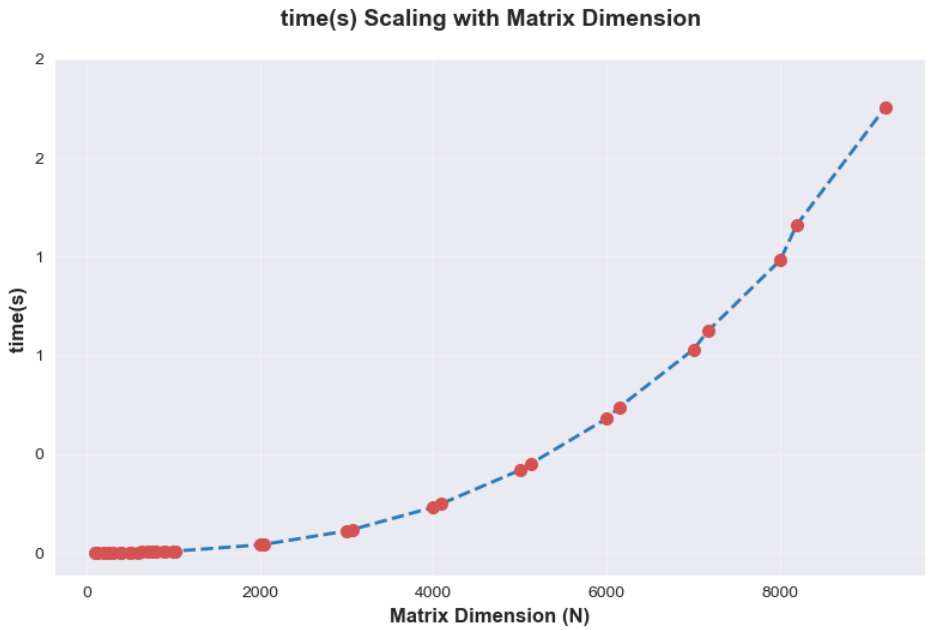

图 6-11 共享内存优化下不同规模下的运行时间

图 6-12 共享内存优化下不同规模下的加速比

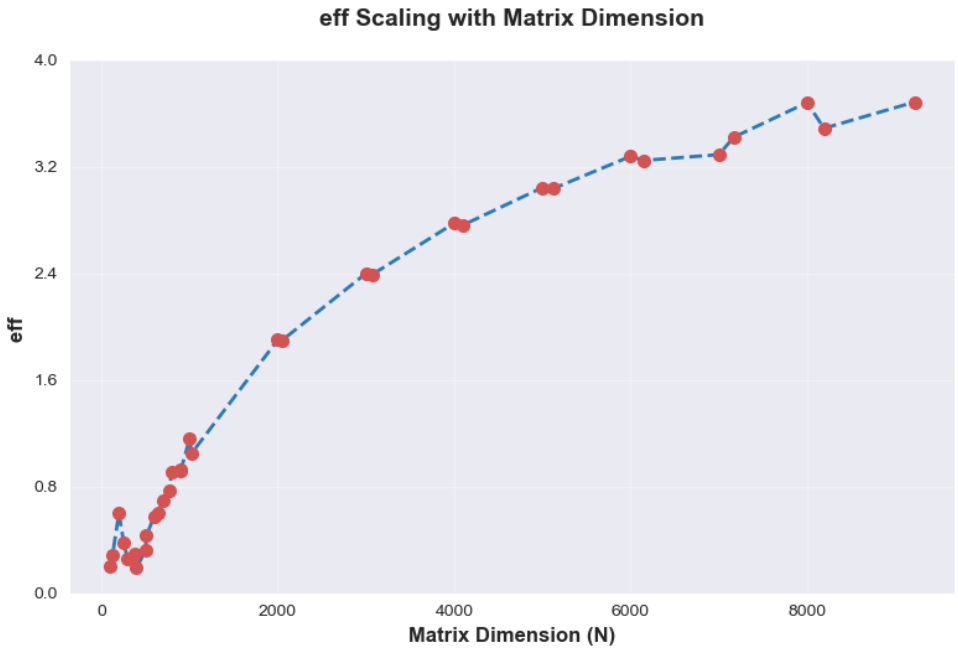

图 6-13 共享内存优化下不同规模下的效率

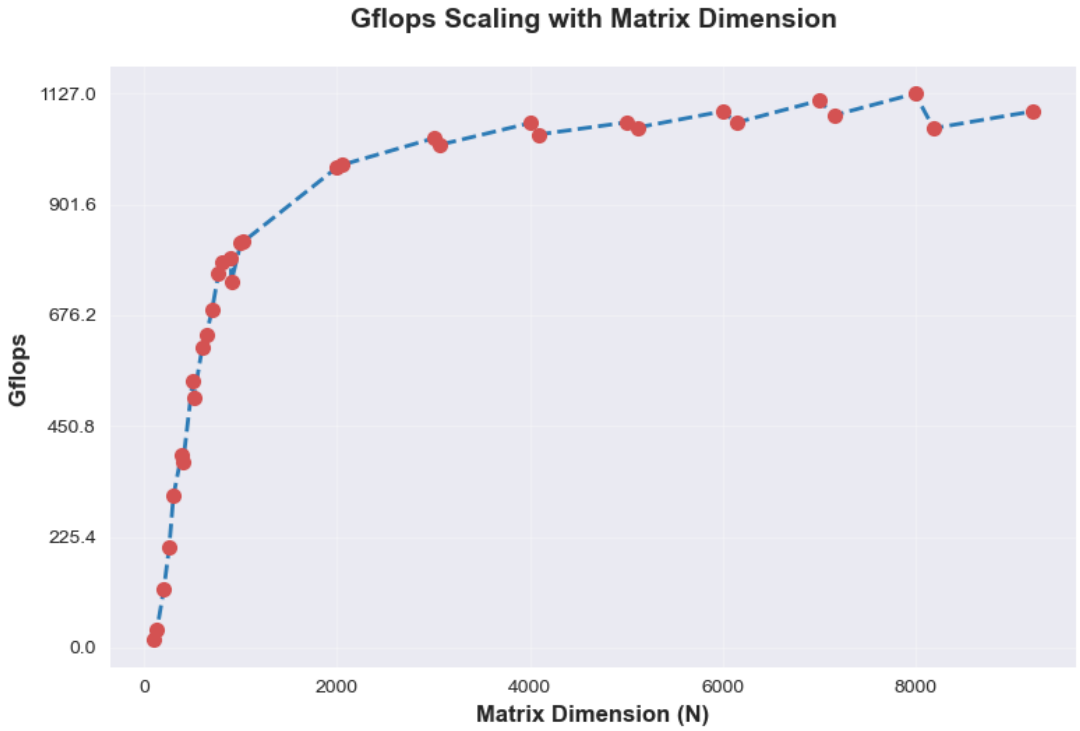

图 6-14 共享内存优化下不同规模下的每秒浮点计算数

与没有共享内存的二维块相比，使用了共享内存优化的矩阵乘运行时间大幅缩短，并且 GFlops 显著提升，加速比与效率也有显著提升，表明，共享内存通过分块减少全局内存访问次数，显著提高带宽利用率，优化数据局部性，显著提升性能。

七、个人总结

通过本次实验，使我掌握了基本的 CUDA 编程的方法，以及对于矩阵乘这个任务的一些基本实现方法，包括 1 维，2 维，共享内存，以及如何分析性能，在波前数固定的情况下，如何分析不同规模的性能等。为我后续的学习和工作打下基础。