总 评 成 绩	

《高性能计算》实验报告册

学年学期：　2023 - 2024 学年第 2 学期

学生姓名：　　高星杰

学生学号：　 2021307220712

专业年级：　计算机科学与技术 21 级

任课教师：　　郑　芳

华中农业大学信息学院

2024 年 4 月

实 验 报 告 册

实验名称	实验四　CUDA 并行程序设计	实 验 成 绩	
实验日期	2024 年 4 月 10 日星期三 第 9-12 节		

一、**实验目的**（描述本实验的学习目的及你对本实验的学习预期。）

1、掌握 CUDA 中线程、线程块、网格以及不同类型内存的基本概念。

2、学习定义和调用核函数（Kernel）以及管理并发执行。

3、编写 CUDA 程序，计算一个 32 维向量与 1024 个 32 维向量的欧式距离，并对其排序。

4、使用 CUDA 编译和调试工具，解决常见问题。

5、熟悉 CUDA 开发环境配置，编译和运行 CUDA 程序。

6、进行数据分配、传输和处理，实现高效计算。

二、**实验环境**（请描述本实验教学活动所使用的实际环境。）

1、电脑主机一台

2、Linux 系统

3、ITC CUDA 实验环境

```
root@1ade2852dd8d:/mnt/cgshare# nvidia-smi
Sun May 19 09:45:41 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.64.00    Driver Version: 440.64.00    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-PCIE...  Off  | 00000000:00:03.0 Off |                    0 |
| N/A   29C    P0    34W / 250W |      0MiB / 32510MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
root@1ade2852dd8d:/mnt/cgshare#
```

三、**实验任务**（本实验要求的实验任务完成情况，未完成注明原因。）

计算一个 32 维度向量与 1024 个 32 维向量的欧式距离，并对其距离进行排序。

四、**实验内容与过程**

（1）核函数结构

在本实验中，实现了四种不同的核函数来进行矩阵向量乘法。每个核函数都有特定的结构和实现方式。

①使用全局内存的核函数

- 每个线程计算矩阵的一行与向量的乘积，并将结果存储在输出向量中。
- 这种方式利用全局内存进行数据存取，适用于基本的矩阵向量乘法。

```
__global__ void MatVecMulGlobalMemory(const lf* A, const lf* B, lf* C, size_t nRow, size_t nCol) {
    size_t i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < nRow) {
        lf res = 0; // 将结果先存在寄存器里，减少对向量 C 的访存
        for (size_t j = 0; j < nCol; ++j)
            res += A[i * nCol + j] * B[j];
        C[i] = res;
    }
}
```

②使用合并访存的核函数

- 对矩阵进行转置，使得相邻线程能够合并访问内存，减少访存开销。
- 这种方式通过优化内存访问模式来提高性能。

```
 __global__ void MatVecMulGlobalMemoryAlign(const lf* At, const lf* B, lf*
C, size_t nRow, size_t nCol) {
    size_t i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < nRow) {
        lf res = 0;
        for (size_t j = 0; j < nCol; ++j)
            res += At[j * nRow + i] * B[j];
        C[i] = res;
    }
}
```

③使用常量内存的核函数

- 将向量存储在常量内存中，利用常量内存的高效读取特性。
- 这种方式适用于向量数据较小且不变的情况。

```
__global__ void MatVecMulConstantMemory(const lf* At, const lf* C, size_t
nRow, size_t nCol) {
    size_t i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < nRow) {
        lf res = 0;
        for (size_t j = 0; j < nCol; ++j)
            res += At[j * nRow + i] * d_Bc[j];
        C[i] = res;
    }
}
```

④使用共享访存的核函数

- 将向量分块加载到共享内存，通过共享内存提高访存效率。
- 这种方式适用于向量数据较小且可以分块处理的情况。

```
__global__ void MatVecMulSharedMemory(const lf* At, const lf* B, lf* C, size_t
nRow, size_t nCol) {
    extern __shared__ lf Bs[];
    size_t i = blockDim.x * blockIdx.x + threadIdx.x;
    lf res = 0;
    for (size_t jBeg = 0, jEnd = blockDim.x < nCol ? blockDim.x : nCol;
         jBeg < nCol;
```

```
        jBeg += blockDim.x, jEnd += blockDim.x) {
        __syncthreads(); // 防止有的进程还在读 Bs
        if (jBeg + threadIdx.x < nCol)
            Bs[threadIdx.x] = B[jBeg + threadIdx.x];
        __syncthreads();
        if (i < nRow)
            for (size_t j = jBeg; j < jEnd; ++j)
                res += At[j * nRow + i] * Bs[j - jBeg];
    }
    if (i < nRow)
        C[i] = res;
}
```

通过这四种核函数结构的设计和实现，我们可以看到不同内存访问和优化策略在 CUDA 编程中的应用。每种核函数都有其特定的优势和适用场景，通过合理选择和优化，可以显著提高矩阵向量乘法的计算效率。

（2）核函数线程分配

①确定处理的向量数目 numVectors 和每个线程块中的线程数目 numThreadsPerBlock

・为了确保所有的向量都能被处理，我们首先需要确定需要处理的向量数目 numVectors。在本实验中，numVectors 即为矩阵的行数 nRow。然后我们需要设定每个线程块中的线程数目 numThreadsPerBlock。这个数目一般根据具体的硬件特性和任务需求来设定，在本实验中设定为 256。

```
const size_t numThreadsPerBlock = 256;
```

②计算需要启动的线程块数目 numBlocks

・为了确定需要启动的线程块数目，我们使用以下公式进行计算：
int numBlocks = (nRow + numThreadsPerBlock - 1) / numThreadsPerBlock;

・这个公式确保了即使在 nRow 不能被 numThreadsPerBlock 整除的情况下，也能启动足够的线程块来处理所有的向量。例如，如果 nRow 为 1025，numThreadsPerBlock 为 256，那么计算出的 numBlocks 为 5，这样能确保最后一个线程块也能处理多余的行。

```
int numBlocks = (nRow + numThreadsPerBlock - 1) / numThreadsPerBlock;
```

③启动核函数

・在确定了线程块的数量和每个线程块中的线程数目之后，我们使用<<<numBlocks, numThreadsPerBlock>>>语法启动 CUDA kernel。通过这种方式，CUDA kernel 会在 GPU 上并行处理向量之间的距离计算，从而加快计算速度。

・以下是四种核函数的启动方式：

```
// 启动全局内存核函数
MatVecMulGlobalMemory<<<numBlocks, numThreadsPerBlock>>>(d_A, d_B, d_C,
```

```
nRow, nCol);

// 启动合并访存核函数
MatVecMulGlobalMemoryAlign<<<numBlocks, numThreadsPerBlock>>>(d_At, d_B,
d_C, nRow, nCol);

// 将向量 B 复制到常量内存并启动常量内存核函数
checkCudaError(cudaMemcpyToSymbol(d_Bc, B, nCol * sizeof(lf)));
MatVecMulConstantMemory<<<numBlocks,  numThreadsPerBlock>>>(d_At,  d_C,
nRow, nCol);

// 启动共享内存核函数
size_t sharedMemSize = numThreadsPerBlock * sizeof(lf); // 计算共享内存的大小
MatVecMulSharedMemory<<<numBlocks,                numThreadsPerBlock,
sharedMemSize>>>(d_At, d_B, d_C, nRow, nCol);
```

（3）优化策略

①使用共享内存

·共享内存是每个线程块共享的高速内存，可以显著减少全局内存访问的延迟。在核函数中，可以将部分数据加载到共享内存中，以减少全局内存访问延迟。例如，在矩阵向量乘法中，可以将向量数据加载到共享内存中，以便多个线程可以共享访问，从而减少全局内存的访问次数。

```
__shared__ lf Bs[];
```

②使用常量内存

·常量内存是 GPU 上的只读内存，适合存储在核函数执行期间不会发生变化的数据。使用常量内存可以减少全局内存的访问，提升读取效率。适用于向量数据较小且不变的情况。

```
__constant__ lf d_Bc[1 << 16 / sizeof(lf)];
```

③使用纹理内存

·如果向量数据具有一定的空间局部性，可以将其存储在纹理内存中。纹理内存具有高速缓存和插值功能，适用于某些数据访问模式。

④优化数据布局和访问模式

·优化向量数据的布局和访问方式，使内存访问更加连续和合并，减少全局内存的访问延迟。例如，使用结构数组或结构体分离数据，可以使内存访问更高效。

⑤减少全局内存访问

・尽量减少对全局内存的读写操作，通过计算和存储中间结果，减少不必要的内存访问。在核函数中，使用寄存器存储中间结果，可以显著减少对全局内存的访问。

⑥使用流处理器

・如果计算任务可以并行化为多个独立的操作，可以使用流处理器并行执行这些操作。流处理器可以执行多个线程块，从而提高计算性能。

⑦调整线程块大小和网格大小

・优化线程块大小和网格大小，以充分利用 GPU 的并行性和资源利用率。不同的设备和应用程序可能对最佳大小有不同的要求。通过实验调整 numThreadsPerBlock 和 numBlocks，找到最佳配置。

⑧内存预取和数据重用

・合理利用缓存和寄存器，预取和重用数据，减少内存访问和数据传输的延迟。在核函数中，使用共享内存或常量内存存储频繁访问的数据，可以显著提高内存访问效率。

（4）完整代码

```
#include <iostream>
#include <cmath>
#include <algorithm>
#include <cuda_runtime.h>
#include <ctime>

#define lf float

// Kernel using global memory
__global__ void MatVecMulGlobalMemory(const lf *A, const lf *B, lf *C, size_t nRow, size_t nCol)
{
    size_t i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < nRow)
    {
        lf res = 0; // 将结果先存在寄存器里，减少对向量 C 的访存
        for (size_t j = 0; j < nCol; ++j)
            res += A[i * nCol + j] * B[j];
        C[i] = res;
    }
}

// Kernel using global memory with coalesced access
__global__ void MatVecMulGlobalMemoryAlign(const lf *At, const lf *B, lf *C, size_t nRow, size_t nCol)
{
    size_t i = blockDim.x * blockIdx.x + threadIdx.x;
```

```
    if (i < nRow)
    {
        lf res = 0;
        for (size_t j = 0; j < nCol; ++j)
            res += At[j * nRow + i] * B[j];
        C[i] = res;
    }
}

// Constant memory for storing the vector
__constant__ lf d_Bc[1024]; // 假设最大维度为 1024

// Kernel using constant memory
__global__ void MatVecMulConstantMemory(const lf *At, lf *C, size_t nRow, size_t nCol)
{
    size_t i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < nRow)
    {
        lf res = 0;
        for (size_t j = 0; j < nCol; ++j)
            res += At[j * nRow + i] * d_Bc[j];
        C[i] = res;
    }
}

// Kernel using shared memory
__global__ void MatVecMulSharedMemory(const lf *At, const lf *B, lf *C, size_t nRow, size_t nCol)
{
    extern __shared__ lf Bs[];
    size_t i = blockDim.x * blockIdx.x + threadIdx.x;
    lf res = 0;
    for (size_t jBeg = 0, jEnd = blockDim.x < nCol ? blockDim.x : nCol;
         jBeg < nCol;
         jBeg += blockDim.x, jEnd += blockDim.x)
    {
        __syncthreads(); // 防止有的进程还在读 Bs
        if (jBeg + threadIdx.x < nCol)
            Bs[threadIdx.x] = B[jBeg + threadIdx.x];
        __syncthreads();
        if (i < nRow)
            for (size_t j = jBeg; j < jEnd; ++j)
                res += At[j * nRow + i] * Bs[j - jBeg];
```

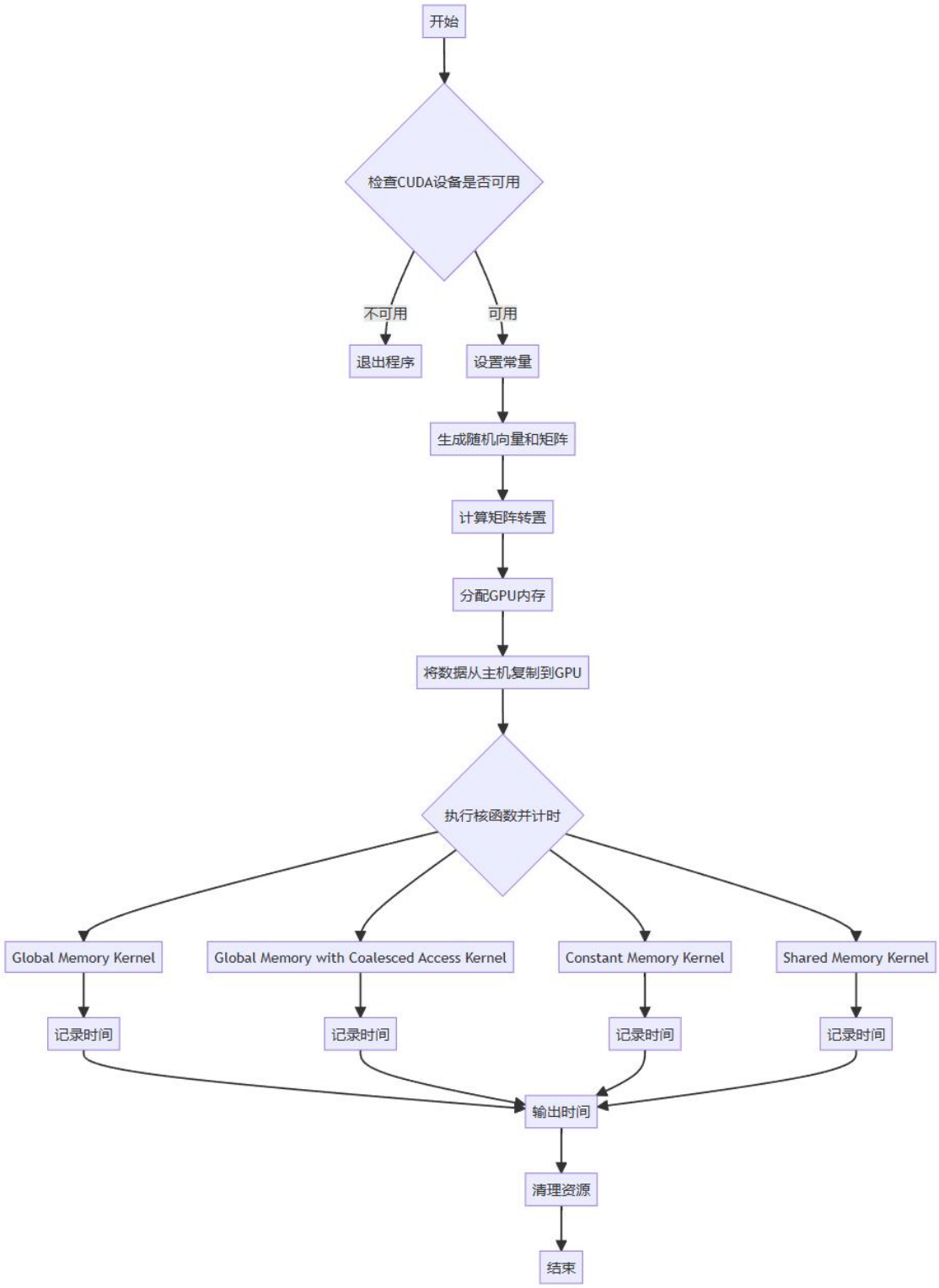
```
    }
    if (i < nRow)
        C[i] = res;
}

// 错误检查函数
void checkCudaError(cudaError_t err)
{
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Error: " << cudaGetErrorString(err) << std::endl;
        exit(EXIT_FAILURE);
    }
}

// 初始化数据
void initializeData(lf *A, lf *B, size_t nRow, size_t nCol)
{
    srand(time(NULL));
    for (size_t i = 0; i < nRow; ++i)
    {
        for (size_t j = 0; j < nCol; ++j)
        {
            A[i * nCol + j] = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
        }
    }
    for (size_t j = 0; j < nCol; ++j)
    {
        B[j] = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
    }
}

// 矩阵转置
void transposeMatrix(const lf *A, lf *At, size_t nRow, size_t nCol)
{
    for (size_t i = 0; i < nRow; ++i)
    {
        for (size_t j = 0; j < nCol; ++j)
        {
            At[j * nRow + i] = A[i * nCol + j];
        }
```

```
    }
}

int main()
{
    const size_t nRow = 1024;
    const size_t nCol = 32;
    const size_t numThreadsPerBlock = 256;

    lf *A = new lf[nRow * nCol];
    lf *B = new lf[nCol];
    lf *C = new lf[nRow];
    lf *At = new lf[nRow * nCol];

    // 初始化数据
    initializeData(A, B, nRow, nCol);
    transposeMatrix(A, At, nRow, nCol);

    lf *d_A, *d_B, *d_C, *d_At;
    checkCudaError(cudaMalloc((void **)&d_A, nRow * nCol * sizeof(lf)));
    checkCudaError(cudaMalloc((void **)&d_At, nRow * nCol * sizeof(lf)));
    checkCudaError(cudaMalloc((void **)&d_B, nCol * sizeof(lf)));
    checkCudaError(cudaMalloc((void **)&d_C, nRow * sizeof(lf)));

    checkCudaError(cudaMemcpy(d_A, A, nRow * nCol * sizeof(lf), cudaMemcpyHostToDevice));
    checkCudaError(cudaMemcpy(d_At, At, nRow * nCol * sizeof(lf), cudaMemcpyHostToDevice));
    checkCudaError(cudaMemcpy(d_B, B, nCol * sizeof(lf), cudaMemcpyHostToDevice));

    // 创建CUDA 事件用于计时
    cudaEvent_t start, stop;
    checkCudaError(cudaEventCreate(&start));
    checkCudaError(cudaEventCreate(&stop));

    // 计算使用全局内存的核函数执行时间
    checkCudaError(cudaEventRecord(start));
    int numBlocks = (nRow + numThreadsPerBlock - 1) / numThreadsPerBlock;
    MatVecMulGlobalMemory<<<numBlocks, numThreadsPerBlock>>>(d_A, d_B, d_C, nRow, nCol);
    checkCudaError(cudaEventRecord(stop));

    checkCudaError(cudaEventSynchronize(stop));
    float milliseconds = 0;
    checkCudaError(cudaEventElapsedTime(&milliseconds, start, stop));
    std::cout << "Global Memory Kernel Time: " << milliseconds << " ms" << std::endl;
```

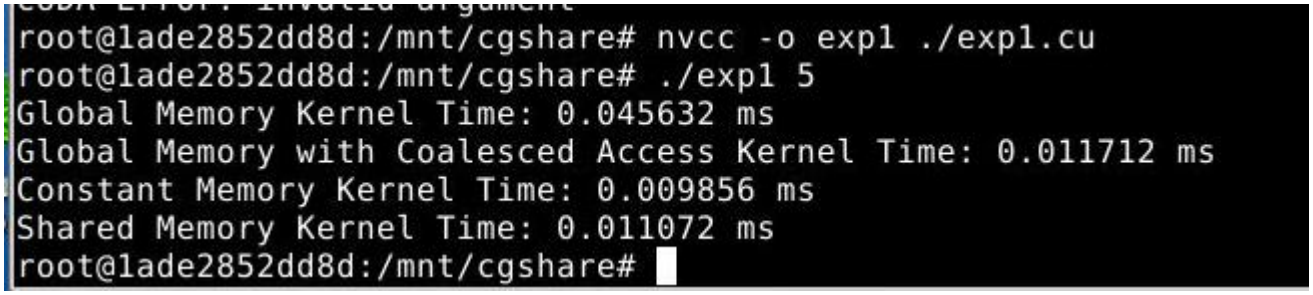
```
    // 计算使用合并访存的核函数执行时间
    checkCudaError(cudaEventRecord(start));
    MatVecMulGlobalMemoryAlign<<<numBlocks, numThreadsPerBlock>>>(d_At, d_B, d_C, nRow, nCol);
    checkCudaError(cudaEventRecord(stop));

    checkCudaError(cudaEventSynchronize(stop));
    checkCudaError(cudaEventElapsedTime(&milliseconds, start, stop));
    std::cout << "Global Memory with Coalesced Access Kernel Time: " << milliseconds << " ms" << std::endl;

    // 将向量 B 复制到常量内存
    checkCudaError(cudaMemcpyToSymbol(d_Bc, B, nCol * sizeof(lf)));

    // 计算使用常量内存的核函数执行时间
    checkCudaError(cudaEventRecord(start));
    MatVecMulConstantMemory<<<numBlocks, numThreadsPerBlock>>>(d_At, d_C, nRow, nCol);
    checkCudaError(cudaEventRecord(stop));

    checkCudaError(cudaEventSynchronize(stop));
    checkCudaError(cudaEventElapsedTime(&milliseconds, start, stop));
    std::cout << "Constant Memory Kernel Time: " << milliseconds << " ms" << std::endl;

    // 计算使用共享内存的核函数执行时间
    size_t sharedMemSize = numThreadsPerBlock * sizeof(lf);
    checkCudaError(cudaEventRecord(start));
    MatVecMulSharedMemory<<<numBlocks, numThreadsPerBlock, sharedMemSize>>>(d_At, d_B, d_C, nRow, nCol);
    checkCudaError(cudaEventRecord(stop));

    checkCudaError(cudaEventSynchronize(stop));
    checkCudaError(cudaEventElapsedTime(&milliseconds, start, stop));
    std::cout << "Shared Memory Kernel Time: " << milliseconds << " ms" << std::endl;

    // 清理资源
    delete[] A;
    delete[] B;
    delete[] C;
    delete[] At;
    cudaFree(d_A);
    cudaFree(d_At);
    cudaFree(d_B);
    cudaFree(d_C);
```

```
    checkCudaError(cudaEventDestroy(start));
    checkCudaError(cudaEventDestroy(stop));

    return 0;
}
```

流程图



五、测试结果



从输出结果可以看出，使用不同内存类型的核函数在执行矩阵-向量乘法时的时间差异：

1. Global Memory Kernel Time: 0.045632 ms
2. Global Memory with Coalesced Access Kernel Time: 0.011712 ms
3. Constant Memory Kernel Time: 0.009856 ms
4. Shared Memory Kernel Time: 0.011072 ms

分析

Global Memory Kernel

全局内存核函数的执行时间最长，这是因为全局内存具有高延迟和低带宽。每次访问全局内存时都需要较长的时间，因此整体性能较差。

Global Memory with Coalesced Access Kernel

使用合并访存的全局内存核函数执行时间大大减少，因为合并访存可以有效地提高内存带宽利用率。合并访存能够将多个小的、不连续的内存访问合并为一个大的、连续的内存访问，减少了内存延迟。

Constant Memory Kernel

常量内存核函数的执行时间最短，因为常量内存是只读的并且被所有线程共享，适用于读取不变的数据。常量内存具有非常高的访问速度，但其大小受限。如果向量 B 能够完全存储在常量内存中，性能会非常好。

Shared Memory Kernel

共享内存核函数的执行时间较短，但比常量内存稍长。共享内存具有低延迟和高带宽，适用于线程间的数据共享和临时存储。通过使用共享内存，可以减少全局内存访问次数，提高性能。然而，正确使用共享内存需要手动管理内存分配和同步，增加了编程复杂度。

总结

1. 全局内存：最慢的内存访问方式，但适用于需要存储大量数据的情况。
2. 合并访存的全局内存：通过优化内存访问模式，可以显著提高全局内存的访问效率。
3. 常量内存：访问速度最快，但受限于内存大小，适用于存储不变的数据。
4. 共享内存：具有较高的访问速度，适用于线程间的数据共享，但需要手动管理。

在实际应用中，选择哪种内存类型和优化策略，取决于具体的应用场景和数据特性。对于矩阵-向量乘法这种计算密集型任务，充分利用合并访存、常量内存和共享内存，可以显著提高性能。