1 实验内容

1.1 实验意义

矩阵向量乘法是线性代数中的基础运算，广泛应用于科学计算、机器学习、图形学等领域。本实验通过 CUDA 并行编程技术实现矩阵向量乘法的 GPU 加速，探究 CUDA 在密集计算任务中的性能表现和优化策略，验证 GPU 并行计算相比传统 CPU 串行计算的优势。

1.2 相关工作

矩阵向量乘法运算复杂度为 $O(m \times n)$，其中 m 为矩阵行数，n 为矩阵列数。该运算具有高度的并行性，每行的计算相互独立，非常适合 GPU 的 SIMD 架构。相比 CPU 串行计算，GPU 并行实现能够充分利用大量计算核心，在大规模数据处理中展现显著性能优势。

1.3 算法原理

矩阵向量乘法基本原理：

1. 给定 $m \times n$ 矩阵 A 和 n 维向量 x
2. 计算结果向量 y，其中 $y_i = \sum_{j=0}^{n-1} (A_{ij} \times x_j)$
3. 并行化策略：为每一行分配一个线程，独立计算该行与向量的点积
4. GPU 实现：使用 CUDA kernel 函数，线程索引直接对应矩阵行号

2 实验方法

2.1 并行化设计

线程组织策略：采用一维线程块配置，每个线程负责计算矩阵的一行与向量的乘积。线程块大小设为 512，网格大小根据矩阵行数动态计算：$grid = (rows - 1)/512 + 1$。

内存管理优化：使用 cudaMalloc 分配 GPU 内存，通过 cudaMemcpy 进行主机与设备间的数据传输。计算完成后及时释放 GPU 内存资源，避免内存泄漏。

算法优化：每个线程内部使用循环累加计算点积，避免了复杂的同步操作。线程间完全独立，无需额外的同步机制。

2.2 错误处理与计时

CUDA 错误检查：实现了完整的 CUDA 错误检查机制，包括运行时错误和 kernel 执行错误的检测，并通过宏定义封装了 API 调用，便于错误定位和调试。

```
/// @brief 检查 CUDA 函数运行时错误
/// @param error CUDA 运行时函数
#define CUDA_CHECK(error) _cudaCheck(error, __FILE__, __LINE__)

/// @brief 检查 CUDA kernel 函数运行时错误
#define KERNEL_CHECK() _kernelCheck(__FILE__, __LINE__)

cudaError_t _cudaCheck(cudaError_t error, const char* file, int line) {
  if (error != cudaSuccess) {
    printf("CUDA error:\nerror_code = %d\nfile = %s\nline = %d\n\
           errName = %s\nerrString = %s\n",
           error, file, line,
           cudaGetErrorName(error), cudaGetErrorString(error));
    return error;
  }
  return error;
}

void _kernelCheck(const char* file, int line) {
  cudaError_t error = cudaGetLastError();
  if (error != cudaSuccess)
    printf("CUDA error:\nerror_code = %d\nfile = %s\nline = %d\n\
           errName = %s\nerrString = %s\n",
           error, file, line,
           cudaGetErrorName(error), cudaGetErrorString(error));
}
```

性能测量：使用 CUDA 事件计时器精确测量 kernel 执行时间，同时使用 CPU 时钟测量串行版本执行时间，其中 CUDA kernel 函数的计时通过宏定义和可变参数模板以类函数式编程简化了计时操作。

```
/// @brief CUDA kernel 函数计时器
/// @param kernel_name kernel 函数名
/// @param print 是否打印计时结果，true 打印，false 不打印
#define KERNEL_TIMER(kernel_name, print, grid, block, ...) \
  _kernelTimer((void*)kernel_name, #kernel_name, print, grid, block, \
               ##__VA_ARGS__)

template<typename... Args>
float _kernelTimer(void* kernel_ptr, const char* kernel_name, const bool print,
                   dim3 grid, dim3 block, Args... args) {
  cudaEvent_t start, stop;
  CUDA_CHECK(cudaEventCreate(&start));
  CUDA_CHECK(cudaEventCreate(&stop));

  CUDA_CHECK(cudaEventRecord(start));
  ((void(*)(Args...))kernel_ptr)<<<grid, block>>>(args...);
  KERNEL_CHECK();
  CUDA_CHECK(cudaEventRecord(stop));

  CUDA_CHECK(cudaEventSynchronize(stop));
  float milliseconds = 0;
  CUDA_CHECK(cudaEventElapsedTime(&milliseconds, start, stop));
  CUDA_CHECK(cudaEventDestroy(start));
  CUDA_CHECK(cudaEventDestroy(stop));

  if (print == true)
    printf("Kernel %s execution time: %.6f ms\n", kernel_name, milliseconds);
  return milliseconds;
}
```

3 性能评估

3.1 实验环境

- **硬件配置**
  - CPU: Intel(R) Core(TM) i5-12600KF（10核16线程）
  - GPU: NVIDIA RTX 3060Ti (8GB GDDR6)(Ampere 架构)
    - 计算能力：8.6
    - SM 个数：38
    - CUDA 核心数: 4864
    - 每 SM 最多线程数：1536

        - 每 SM 最多线程块数: 16
    - 内存: DDR5 32GB
- **软件环境**
    - OS: debian12 Bookworm desktop
    - 编译器: NVCC + GCC 12.2.0
    - CUDA 版本: 11.8.89 stable

3.2 性能测试结果

小规模测试（25×10 矩阵）：

- 并行计算时间：0.075616 ms
- 串行计算时间：0.001000 ms
- 结果验证：计算结果完全匹配，但由于分配矩阵行数小于一个线程块的线程数，GPU 并行化开销大于计算收益，出现负加速现象

大规模测试（2500×1000 矩阵）：

- 并行计算时间：0.221792 ms
- 串行计算时间：3.397000 ms
- 结果验证：计算结果完全匹配

4 结论与优化措施

1. **GPU 加速效果显著**：在大规模计算中，CUDA 实现了 15.31 倍的性能提升
2. **规模阈值效应**：存在临界数据规模，低于该规模时 GPU 加速效果不明显
3. **内存传输优化空间**：对于多次计算可考虑数据常驻 GPU 内存
4. **算法正确性验证**：串行和并行版本结果完全一致，验证了实现的正确性

通过本实验深入理解了 CUDA 并行编程模型和 GPU 加速计算的特点。实验结果表明，GPU 在处理大规模密集计算任务时具有显著优势，但需要合理评估并行化开销与计算收益的平衡点。