并行程序设计与算法
作业 2：矩阵与向量乘法

20337025 崔璨明
计算机科学与技术（人工智能与大数据）

2023 年 7 月 4 日

目录

1 实验题目

- 矩阵与向量乘法
  - 给定一个输入矩阵 A 及向量 b，输出其向量 $c = A \cdot b$
- 回答以下问题
  1. 介绍程序整体逻辑，包含的函数，每个函数完成的内容
  2. 讨论矩阵大小及线程组织对性能的影响，可考虑但不限于以下因素
     - 一维 vs 二维
     - 线程块大小对性能的影响
     - 每个线程计算的元素数量对性能的影响
     - 存储方式对性能的影响
     - 以上配置在处理不同大小/稀疏程度的矩阵时，性能可能的差异
  3. 优化

2 程序逻辑 & 代码介绍

首先编写了一个简单的矩阵和向量相乘的 CUDA 程序。程序包括中三个函数：

- matrixVectorMultiplication() 函数: 该函数是主机上的函数，用于管理 GPU 计算过程。首先，在设备上分配内存，分别为输入矩阵 A、向量 b 和输出向量 c。然后，将输入数据从主机内存复制到设备内存中。接下来，计算启动参数，包括线程块大小和网格大小。调用核函数 matrixVectorMul 在 GPU 上执行矩阵与向量乘法。最后，将结果从设备内存复制回主机内存，并释放设备内存
- matrixVectorMul() 函数：该函数是 CUDA 核函数，用于在 GPU 上执行矩阵与向量乘法。核函数使用并行计算的方式，每个线程负责计算输出向量中的一个元素。每个线程根据线程索引 tid 计算输出向量中对应位置的值，使用矩阵 A 的一行与向量 b 进行点积计算。计算结果存储在输出向量 c 的对应位置。
- main() 函数：负责读入数据，构建输入矩阵 A 和向量 b，然后创建输出向量 c。调用 matrixVectorMultiplication() 函数进行矩阵与向量乘法计算。最后，输出结果或将结果写入到输出文件。

代码如下：

```
1  // CUDA核函数，用于在GPU上执行矩阵向量乘法
2  __global__ void matrixVectorMul(float* A, float* b, float* c, int rows,
       int cols) {
3      // 输入参数：
4      //   A: 输入矩阵A的指针
5      //   b: 输入向量b的指针
6      //   c: 输出向量c的指针
7      //   rows: 矩阵A的行数
```

```
8       //    cols: 矩阵A的列数
9       // 计算当前线程的唯一标识符
10      int tid = blockIdx.x * blockDim.x + threadIdx.x;
11      if (tid < rows) {
12          // 当前线程负责计算向量c的一个元素
13          float sum = 0.0f;
14          for (int j = 0; j < cols; j++) {
15              // 累加矩阵A的一行与向量b的对应元素的乘积
16              sum += A[tid * cols + j] * b[j];
17          }
18          // 将结果保存到向量c中
19          c[tid] = sum;
20      }
21  }
22  void matrixVectorMultiplication(std::vector<float>& A, std::vector<
        float>& b, std::vector<float>& c, int rows, int cols) {
23      // 执行矩阵向量乘法的函数
24      // 输入参数:
25      //   A: 输入矩阵A
26      //   b: 输入向量b
27      //   c: 输出向量c
28      //   rows: 矩阵A的行数
29      //   cols: 矩阵A的列数
30      // 在GPU上分配设备内存
31      float *d_A, *d_b, *d_c;
32      cudaMalloc((void**)&d_A, rows * cols * sizeof(float));
33      cudaMalloc((void**)&d_b, cols * sizeof(float));
34      cudaMalloc((void**)&d_c, rows * sizeof(float));
35      // 将数据从主机内存复制到设备内存
36      cudaMemcpy(d_A, A.data(), rows * cols * sizeof(float),
            cudaMemcpyHostToDevice);
37      cudaMemcpy(d_b, b.data(), cols * sizeof(float),
            cudaMemcpyHostToDevice);
38      // 启动CUDA核函数来执行矩阵向量乘法
39      int blockSize = 256;
40      int gridSize = (rows + blockSize - 1) / blockSize;
41      matrixVectorMul<<<gridSize, blockSize>>>(d_A, d_b, d_c, rows, cols)
            ;
42      // 将结果从设备内存复制到主机内存
43      cudaMemcpy(c.data(), d_c, rows * sizeof(float),
            cudaMemcpyDeviceToHost);
44
```

```
45      // 释放设备内存
46      cudaFree(d_A);
47      cudaFree(d_b);
48      cudaFree(d_c);
49  }
50  // 主函数
51  int main() {
52      // 定义输入和输出文件路径
53      string read_dir = "data/test1.in";
54      string save_dir = "output/o1.out";
55
56      std::vector<float> A;
57      std::vector<std::vector<float>> array_2d;
58      std::vector<float> b;
59      // 读取数据或生成数据到向量A、array_2d和b
60      //read(A, array_2d, b, read_dir);
61      generate_data(A, array_2d, b, 64);
62      // 获取矩阵A的行数和列数
63      int rows = array_2d.size();
64      int cols = array_2d[0].size();
65      // 创建输出向量c
66      std::vector<float> c(rows);
67      // 计时开始
68      clock_t start, finish;
69      start = clock();
70      // 执行矩阵向量乘法
71      matrixVectorMultiplication(A, b, c, rows, cols);
72      // 计时结束
73      finish = clock();
74      // 打印结果
75      std::cout << "Result: ";
76      for (int i = 0; i < rows; i++) {
77          //std::cout << c[i] << " ";
78          printf("%.5f ", c[i]);
79      }
80      std::cout << std::endl;
81      // 打印执行时间
82      std::cout << "Using time: " << 1000 * double(finish - start) /
            CLOCKS_PER_SEC << " ms" << std::endl;
83      return 0;
84  }
```

程序中包括了头文件 read_data.h，该头文件用于生成实验数据、读取二进制文件并写入到矩

阵中、将运算结果写出到二进制文件，包括以下几个函数：

- read(array,vec,path)：读取路径 path 的二进制文件，并将矩阵存储到 array 数组中，将向量存储到 vec 数组中。
- write(result,save_dir)：将数组 result 写入到路径 save_dir 的二进制文件。
- generate_data(array,vec,size)：生成一个 size*size 的随机浮点数组 array 和大小为 size 向量 vec，用于进行对比实验。

程序整体逻辑：

在主函数中创建输入矩阵 A 和向量 b ，并初始化数据。输出向量 c 用于存储计算结果。随后调用主机函数 matrixVectorMultiplication，将输入数据和输出向量作为参数传递给该函数。

在 matrixVectorMultiplication 函数中，分配 GPU 设备上的内存，用于存储输入矩阵 A、向量 b 和输出向量 c。然后将输入数据从主机内存复制到 GPU 设备上的内存中。接着计算启动参数，包括线程块大小和网格大小，以确保在 GPU 上进行并行计算。接下来调用核函数 matrixVectorMul 在 GPU 上执行矩阵与向量乘法。

最后将计算结果从 GPU 设备上的内存复制回主机内存中的输出向量 c 并释放 GPU 设备上的内存。返回至主函数中，输出结果或进行保存。

3 实验部分

3.1 实验环境

在一个服务器上进行实验，该服务器一些配置如下：

- CPU 为 Intel(R) Xeon(R) CPU E5-2630
- CPU 核心数：20 个物理核，40 个逻辑核
- GPU：NVIDIA TITAN Xp
- 编译器：gcc 5.5.0，nvcc 10.1
- 系统版本：Ubuntu 16.04

3.2 程序正确性

为了验证程序的正确性，我将计算得到的结果输出到二进制文件中，然后编写了一个测试程序，用于与助教提供的输出进行对比，若每个元素的误差都在 10^{-5} 以内，则程序正确性得到了验证，测试程序如下：

```
#include "read_data.h"
int SIZE=2048;
void test(string file1,string file2){
    std::vector<float> res1;
    std::vector<float> res2;
    read_res(res1,file1,SIZE);
    read_res(res2,file2,SIZE);
```

```
    bool flag=false;
    for(int i=0;i<SIZE;i++){
        if(fabs(res1[i]-res2[i])>1e-5){
            printf("%.5f %.5f\n",res1[i],res2[i]);
            flag=true;
        }
    }
    if(flag==false)
    printf("The result is right.\n");
}
int main(){
    string file1="data/test1.out";
    string file2="output/res1.out";
    test(file1,file2);
    return 0;
}
```

对每个输入输出样例我都进行了测试，测试结果都是正确的，输出文件可以直接运行程序得到。

3.3 一维 vs 二维

上面的程序使用的是一维的线程块，为了探究一维线程块和二维线程块对性能的影响，在原程序上稍加修改，使用二维的线程块：

```
__global__ void matrixVectorMul(float* A, float* b, float* c, int rows,
    int cols) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if (row < rows && col < cols) {
        float sum = 0.0f;
        for (int j = 0; j < cols; j++) {
            sum += A[row * cols + j] * b[j];
        }
        c[row] = sum;
    }
}
void matrixVectorMultiplication(float* A, std::vector<float>& b, std::
   vector<float>& c, int rows, int cols){
    ...
    // Define block and grid dimensions
    dim3 blockSize(BLOCK_SIZE_X, BLOCK_SIZE_Y);
    dim3 gridSize((cols + blockSize.x - 1) / blockSize.x, (rows +
       blockSize.y - 1) / blockSize.y);
    ...
}
```

设置 blockSize 为 256，随机生成不同规模的矩阵和向量，分别运行两个程序进行计算，记录每次运行的时间（多次运行取平均值），得到结果如表 1 所示：

矩阵规模与向量大小	一维 block 用时 (ms)	二维 block 用时 (ms)
(128*128),128	238.987	220.015
(256*256),256	242.616	236.124
(512*512),512	249.510	243.073
(1024*1024),1024	234.007	264.892
(2048*2048),2048	263.675	308.721
(4096*4096),4096	277.846	644.994

表 1: 一维 block vs 二维 block

从实验结果分析，在矩阵乘向量的计算中，一维线程块相对于二维线程块较快。这可能是因为一维线程块可以更好地利用连续内存访问模式。在矩阵乘向量的计算中，矩阵通常是以行主序存储的，即相邻元素在内存中是连续的。当一个线程块中的线程按顺序读取矩阵的行并乘以向量的元素时，连续内存访问将提供更高的内存带宽和更好的缓存利用率。

3.4 线程块大小对性能的影响

以一维的线程块的程序进行测试，输入矩阵和向量为 test1.in 中的数据，分别采用不同的线程块大小，每个线程块大小运行程序十次，记录每次计算所花费的时间，取平均值，得到的结果如表 2 所示：

线程块大小	用时 (ms)
64	235.014
128	245.338
256	244.819
512	249.906
1024	250.452

表 2: 不同线程块大小

可见，虽然在理论上，较大的线程块大小可以更好地隐藏内存访问延迟，但过大的线程块可能导致资源竞争和调度延迟。因此，需要进行实验和调优来确定最佳的线程块大小。在这次实验中，线程块大小增大时速度反而有所降低。

3.5 每个线程计算的元素数量对性能的影响

要改变每个线程计算的元素数量，可以调整线程块和线程的数量，以及计算每个线程处理的元素索引范围。以下代码确定了线程块和线程的数量：

```
1 int blockSize = 256;
2 int gridSize = (rows + blockSize - 1) / blockSize;
```

核函数中，可以修改计算每个线程处理的元素索引范围的逻辑。例如，可以使用以下方式来计算每个线程的起始索引和结束索引：

```
int tid = blockIdx.x * blockDim.x + threadIdx.x;
int startIdx = tid * elementsPerThread;
int endIdx = min((tid + 1) * elementsPerThread, rows);
```

这里，elementsPerThread 是每个线程处理的元素数量。通过将 startIdx 和 endIdx 用于循环或计算中，可以确保每个线程处理指定数量的元素。

修改 elementsPerThread，进行多次实验，得到结果如表 3 所示：

元素数量	1	8	32	64	128	256
用时 (ms)	235.522	224.267	224.58	216.789	227.432	247.13

表 3: 每个线程处理元素数量的影响

可以看到，当每个线程处理 64 个元素时速度较快，并不是越大越好。因此每个线程计算的元素数量应适当，避免计算资源浪费和负载不均衡。如果每个线程计算的元素数量太少，则可能无法充分利用 GPU 的并行性；如果太多，则可能导致负载不均衡和线程间同步的开销。

3.6 存储方式对性能的影响

在原程序中，矩阵 A 和向量 b 都是存储在全局内存中，为了探究存储方式对性能的影响，我分别使用常量内存和纹理内存来存储矩阵，并比较不同规模矩阵下各种存储方式的计算时间，得到的结果如表 4 所示。

存储方式/矩阵规模	128	256	512	1024	2048	4096
全局内存	185.907	187.088	186.811	180.324	181.73	202.141
常量内存	150.747	176.216	173.642	166.529	176.033	超出限制
纹理内存	180.605	156.919	162.597	171.055	161.029	195.985

表 4: 存储方式对性能的影响

由此可见，使用常量内存和纹理内存的高速缓存机制可以加速数据访问，使程序的计算时间更短。

4 优化

4.1 使用常量内存

在原程序的基础上，采用常量内存来进行优化。我们知道，在运算的过程中，矩阵 A 和向量 b 是不变的，因此可以用常量内存储存向量 b，具体实现如下。首先声明常量内存数组：

```
__constant__ float d_b[2048];
```

调用核函数之前，先将主机内存中的向量 b 拷贝到常量内存中即可：

```
cudaMemcpyToSymbol(d_b, b.data(), cols * sizeof(float));
```

线程的核函数如下：

```
__global__ void matrixVectorMul(float* A,float* c, int rows, int cols)
    {
```

```
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < rows) {
        float sum = 0.0f;
        for (int j = 0; j < cols; j++) {
            sum += A[tid * cols + j] * d_b[j];
        }
        c[tid] = sum;
    }
}
```

使用了常量内存后，程序的性能有所提升，具体实验结果见 3.6 节，但当矩阵的规模过大时（如 4096），常量内存貌似放不下整个数组 b，程序的运行结果全为 0。

4.2 使用纹理内存

首先定义纹理内存对象：在核函数之前，声明纹理内存对象并指定数据类型，具体代码如下：

```
texture<float, 1, cudaReadModeElementType> texA;
texture<float, 1, cudaReadModeElementType> texB;
```

绑定纹理内存对象：在数据从主机到设备的拷贝之前，使用 cudaBindTexture 函数将纹理内存对象绑定到相应的设备指针上，在 cudaMemcpy 之前添加代码：

```
cudaBindTexture(NULL, texA, d_A, rows * cols * sizeof(float));
cudaBindTexture(NULL, texB, d_b, cols * sizeof(float));
```

核函数中直接使用纹理内存对象进行计算：

```
__global__ void matrixVectorMul(float* c, int rows, int cols) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < rows) {
        float sum = 0.0f;
        for (int j = 0; j < cols; j++) {
            sum += tex1Dfetch(texA, tid * cols + j) * tex1Dfetch(texB,
                j);
        }
        c[tid] = sum;
    }
}
```

最后还要解绑：

```
cudaUnbindTexture(texA);
cudaUnbindTexture(texB);
```

同样，使用了纹理内存后，程序的性能有所提升，具体实验结果见 3.6 节。

4.3 使用 cuBLAS 进行优化

cuBLAS 是 CUDA 基本线性代数子程序库（CUDA Basic Linear Algebra Subroutine library），可以使用 cuBLAS 库用于进行矩阵运算，具体实现如下：

首先创建 cuBLAS 句柄：在 matrixVectorMultiplication 函数中添加以下代码：

```
cublasHandle_t handle;
cublasCreate(&handle);
```

然后可以调用 cuBLAS 函数进行矩阵向量乘法：

```
float alpha = 1.0f;
float beta = 0.0f;
cublasSgemv(handle, CUBLAS_OP_N, cols, rows, &alpha, d_A, cols, d_b, 1,
    &beta, d_c, 1);
```

使用完成后销毁 cuBLAS 句柄：

```
cublasDestroy(handle);
```

完整代码如下：

```
//v4版本，用了cublas进行优化
void matrixVectorMultiplication(std::vector<float>& A, std::vector<
    float>& b, std::vector<float>& c, int rows, int cols) {
    // Device memory allocation
    float *d_A, *d_b, *d_c;
    cudaMalloc((void**)&d_A, rows * cols * sizeof(float));
    cudaMalloc((void**)&d_b, cols * sizeof(float));
    cudaMalloc((void**)&d_c, rows * sizeof(float));
    // Copy data from host to device
    cudaMemcpy(d_A, A.data(), rows * cols * sizeof(float),
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b.data(), cols * sizeof(float),
        cudaMemcpyHostToDevice);
    // cuBLAS initialization
    cublasHandle_t handle;
    cublasCreate(&handle);
    // Matrix-vector multiplication using cuBLAS
    float alpha = 1.0f;
    float beta = 0.0f;
    cublasSgemv(handle, CUBLAS_OP_T, cols, rows, &alpha, d_A, cols, d_b
        , 1, &beta, d_c, 1);
    // Copy result from device to host
    cudaMemcpy(c.data(), d_c, rows * sizeof(float),
        cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A);
```

```
22      cudaFree(d_b);
23      cudaFree(d_c);
24      // Destroy cuBLAS handle
25      cublasDestroy(handle);
26  }
27
28  int main() {
29      ...
30      matrixVectorMultiplication(A, b, c, rows, cols);
31      ...
32  }
```

用于调用了 cuBLAS 库，因此编译时需要动态链接，编译命令为：

nvcc matrix_vector_mul_v4.cu -o matrix_vector_mul_v4 -lcublas -std=c++11

对于这三种优化，分别使用不同规模的矩阵进行实验，每一次实验进行多次取平均值，绘制图像如下：

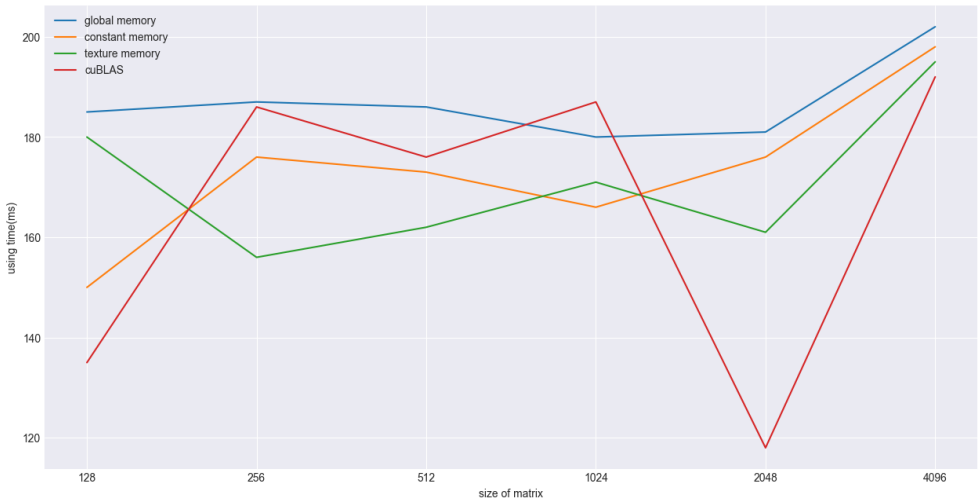

图 1: 不同优化的性能对比

5 实验总结

这次实验我实现了矩阵和向量相乘的 cuda 程序。并通过实验对影响性能的各项因素进行了分析，如选择使用一维线程块还是二维线程块会对性能产生影响，取决于矩阵的大小和计算的规模。而且线程块的大小会影响并行计算的效率。选择合适的线程块大小可以充分利用计算资源并减少调度开销。除此之外每个线程计算的元素数量：合理分配每个线程计算的元素数量可以避免线程空闲和负载不均衡的问题，提高并行计算的效率。当然选择适合计算模式的存储方式可以提高计算的效率，如使用常量内存来代替全局内存，可以更好地利用缓存的局部性。

此外，我还进行了一些优化，如合理选择矩阵的存储方式，充分利用缓存的局部性；使用更高效的算法，调用 cuBLAS 库。

通过这次实验，我不仅了解了矩阵与向量乘法的实现，还学到了如何通过优化策略和合理选择配置来提高程序的性能。这对我今后在并行计算和优化领域的学习和工作都有很大的帮助。