并行计算实验三报告
——矩阵乘法的并行及优化

PB20111701 叶升宇 **PB20111689 蓝俊玮**

说明

本此实验由 PB20111689 蓝俊玮完成。

1. 实验题目

矩阵乘法的并行及优化

2. 实验环境

个人电脑：

- 操作系统：Windows 10 家庭中文版 22H2
- IDE ：Visual Studio 2022
- OpenMP 库：OpenMP 2.0
- 处理器：AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz

服务器：

- 操作系统：Ubuntu 18.04.6 lts
- 处理器：Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz machine
- GPU: NVIDIA GeForce RTX 2080 Ti

3. 矩阵乘法

矩阵乘法的定义就是两个矩阵对指定行号和列号的元素依次相乘的和作为输出矩阵对应的行号列号的元素。

用数学语言表示 $A=[a]_{m\times n}, B=[b]_{n\times k}$ 就是：

$$C=AB=[c]_{m\times k},\quad c_{ij}=\sum_{k=1}^{n}a_{ik}b_{kj}$$

3.1. OpenMP 并行程序

使用 OpenMP 实现并行编程,采用 `static` 方式的任务调度，将工作量均衡的分配给每个线程，这样可以让每个线程的负载均衡。

```
typedef vector<vector<int> > Matrix;
void matrix_mul_para(Matrix A, Matrix B, Matrix& C, int size) {
    int i, j, k;
#pragma omp parallel shared(A, B, C, size) private(i, j, k) {
    #pragma omp for schedule (static)
        for (i = 0; i < size; i++) {
            for (j = 0; j < size; j++) {
                int sum = 0;
                for (k = 0; k < size; k++) {
                    sum += A[i][k] * B[k][j];
                }
                C[i][j] = sum;
```

```
            }
        }
    }
}
```

3.1.1. 性能测试

说明

实验中测试的矩阵大小为 2048×2048，这主要还是考虑到串行程序的运行时间。因为串行算法的时间复杂度为 $O(n^3)$，因此当数据规模规模大一倍时，运行时间会大 8 倍左右；而规模较小时，程序运行的时间太快，不便于比较高线程时的性能表现。

AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz

由于矩阵乘法的定义十分简单，下面就直接对上述串行矩阵乘法的代码进行性能测试。实验中采取 2048×2048 大小的矩阵用来作为测试样例。程序运行之初先用随机数填充矩阵，然后进行性能测试，程序的运行时间如下：

线程数量	2	4	6	8	10	12	14	16
串行程序	257.306	257.306	257.306	257.306	257.306	257.306	257.306	257.306
并行程序	137.517	49.737	37.973	33.128	31.052	29.925	28.781	28.001
加速比	1.87	5.17	6.78	7.77	8.27	8.67	9.01	9.26

可以看到直接进行并行加速的效果也很好。

3.1.2. 算法优化

在实际编程中，我们通常希望能够使用更加高效的算法来实现矩阵乘法。在算法导论中提到了一些矩阵乘法的算法设计，但是在这里我们不去讨论在非并行算法上的优化。

我们采用的方式是让第二个矩阵 B 转置，然后再按照矩阵乘法的定义来计算。这样做的原因在于，将第二个矩阵进行转置后，其列向量变成了行向量，这使得我们可以将其看做一个行向量的集合，从而利用缓存的局部性来提高计算效率。

矩阵乘法的转置优化方法在并行算法设计中是非常常用的一种方法。在并行计算中，我们通常希望能够最大限度地利用计算机的多个处理单元，从而提高计算效率。而对于矩阵乘法这样的密集型计算任务，我们可以将矩阵分块，然后将每个块分配给不同的处理单元进行计算，从而实现并行计算。但是，由于不同的处理单元之间需要共享数据，因此数据传输的代价可能会成为瓶颈。

为了避免这个问题，我们可以利用转置优化方法，将其中一个矩阵进行转置后再进行计算。这样可以将内存中的数据布局进行重新排列，从而减少数据传输的代价。这样，每个处理单元只需要在内存中访问自己负责的块，可以最大限度地利用缓存的局部性，减少数据传输的代价。因此矩阵乘法的转置优化方法是很有必要的。

```
typedef vector<vector<int> > Matrix;
void matrix_mul_tran(Matrix A, Matrix BT, Matrix& C, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            int sum = 0;
            for (int k = 0; k < size; k++) {
```

```
                sum += A[i][k] * BT[j][k];
            }
            C[i][j] = sum;
        }
    }
}
```

3.1.3. 性能测试

实验中采取 2048×2048 大小的矩阵用来作为测试样例。程序运行的时间如下：

线程数量	2	4	6	8	10	12	14	16
串行程序	257.306	257.306	257.306	257.306	257.306	257.306	257.306	257.306
并行程序	137.517	49.737	37.973	33.128	31.052	29.925	28.781	28.001
加速比	1.87	5.17	6.78	7.77	8.27	8.67	9.01	9.26
串行转置程序	196.676	196.676	196.676	196.676	196.676	196.676	196.676	196.676
并行转置程序	69.890	37.987	29.744	26.125	24.533	23.298	22.651	21.852
加速比	2.814	5.177	6.612	7.528	8.017	8.442	8.683	9.000

对比上述结果，可以看到转置程序可以优化缓存的读取，因此不同的串行程序的访存时间会得到优化。通过合理的转置处理，可以减少缓存不命中与替换发生的次数。而对于并行程序，发现在4和6线程的时候出现了加速比大于线程数量的现象。经过询问老师得知，可能是由于其余没有进行计算的线程在这个算法执行过程中参与了数据通信间的工作，因此发生了上述现象。

从总体看来，并行的效果是很不错的。

3.2. Cuda 并行程序

由于 OpenMP 的实现操作太受限，只能循环前加上 `#pragma`，也少有线程与数据之间的组织，感觉只是很存粹的并行执行，因此我接着使用 Cuda 来探究矩阵乘法。

说明

接下来的实验中测试的矩阵大小为 1024×1024，因为 Cuda 的性能表现很好，因此不使用过大的矩阵进行测试。同时接下来的实验测试环境在服务器上。

Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz machine with an NVIDIA GeForce RTX 2080 Ti

3.2.1. 并行实现

```
__global__ void matrix_mul1(float *out, float *a, float *b, int n)
{
    int tid_x = threadIdx.x;
    int tid_y = threadIdx.y;

    for (int i = tid_y; i < n; i += blockDim.y) {
        for (int j = tid_x; j < n; j += blockDim.x) {
            float sum = 0.0;
            for (int k = 0; k < n; k++) {
```

```
                sum += a[i * n + k] * b[k * n + j];
            }
            out[i * n + j] = sum;
        }
    }
}

__global__ void matrix_mul2(float *out, float *a, float *b, int n)
{
    int tid_x = threadIdx.x;
    int tid_y = threadIdx.y;
    int bid_x = blockIdx.x;
    int bid_y = blockIdx.y;
    int x = bid_x * blockDim.x + tid_x;
    int y = bid_y * blockDim.y + tid_y;
    int stride_x = blockDim.x * gridDim.x;
    int stride_y = blockDim.y * gridDim.y;

    for (int i = y; i < n; i += stride_y) {
        for (int j = x; j < n; j += stride_x) {
            float sum = 0.0;
            for (int k = 0; k < n; k++) {
                sum += a[i * n + k] * b[k * n + j];
            }
            out[i * n + j] = sum;
        }
    }
}
```

这一部分的实现没有什么好说的，实际上就是在 x 和 y 方向上并行的做。重点介绍下面的实现

3.2.2. 共享内存

因为矩阵乘法中有大量的重复访存操作，因为我们可以使用共享内存来存储矩阵，这样可以减少 gpu 中全局内存的访问次数。同时使用共享内存后，缓存不命中和替换次数也会相应地减少。

因为在并行的过程中，实际上我们的程序是通过对矩阵分块进行计算的。分块计算的好处就是内存的排布是连续的，对于缓存也是友好的。因此我们的共享内存也采取分块的形式进行存储。为了减少重复访存次数，我们分别定义了 `temp_a` 和 `temp_b` 共享内存来存储矩阵 a 和 b。由于共享内存是块内共享的，在不同块之间是不共享的，因此我们在存储 a 和 b 的时候需要将其全局的地址存储到块内的地址。

因此在存储之前，首先我们要计算出 a 和 b 的存储位置 `offset_a` 和 `offset_b`。然后将对应的数据复制到共享内存数组中。具体来说，每个线程从矩阵 a 和 b 中读取一个元素，然后将其存储到 temp_a 和 temp_b 数组中。当所有线程都完成了数据加载之后，调用 syncthreads() 函数，等待所有线程都完成了数据加载操作，确保共享内存中的数据已经全部被加载。

接着我们按分块计算的操作，计算每个块中的值，将其加到 `temp_f` 中。当所有线程都完成了计算之后，调用 `__syncthreads()` 函数，等待所有线程都完成了计算操作，确保共享内存中的数据已经全部被使用。(注意这个分块计算十分重要，如果不通过分块计算的话，实际上该算法是不能读取完整的行列信息。它只能读取到 1 个块中的行列信息，即只有 32x32 的信息，是不可能实现完整的矩阵乘法操作)

当然这种实现方式和数据集的大小有很大关系，需要按照数据集的大小来设置 blockSize 和 gridSize ，以及共享内存的大小与读取。

```
__global__ void matrix_mul3(float *out, float *a, float *b, int n)
{
    __shared__ float temp_a[32][32];
    __shared__ float temp_b[32][32];

    int tid_x = threadIdx.x;
    int tid_y = threadIdx.y;
    int bid_x = blockIdx.x;
    int bid_y = blockIdx.y;
    int x = bid_x * blockDim.x + tid_x;
    int y = bid_y * blockDim.y + tid_y;

    float temp_f = 0.0;
    for (int block = 0; block < n / 32; block++) {
        int offset_a = bid_y * blockDim.x * n + block * blockDim.x;
        int offset_b = block * blockDim.x * n + bid_x * blockDim.x;
        temp_a[tid_y][tid_x] = a[offset_a + tid_y * n + tid_x];
        temp_b[tid_y][tid_x] = b[offset_b + tid_y * n + tid_x];

        __syncthreads();

        for (int k = 0; k < 32; k++) {
            temp_f += temp_a[tid_y][k] * temp_b[k][tid_x];
        }

        __syncthreads();
    }

    out[y * n + x] = temp_f;
}
```

3.2.3. 性能测试

实现方法	时间(ms)	加速比
串行程序	5275.674	-
单block多threads	147.601	35.74
多blocks多threads	1.800	2930.93
共享内存	0.995	5302.18

可以看出来，gpu 的浮点计算非常快。程序优化后的性能十分显著。

4. 实验总结

本次实验主要在于探究矩阵乘法的并行以及优化，在本次实验采用了转置的方式优化矩阵乘法，从而达到缓存一致性的目的。通过不同的线程数，我们可以明显的感受到并行程序的高性能表现，也可以明显的感受到优化前后的性能提升。在实验的过程中，可以发现 Cuda 的优化效果要远好于 OpenMP ，是因为 Cuda 可以自己定义数据和处理器之间的对应关系，同时 Cuda 可以开的线程规模要远大于 OpenMP ,同时 GPU 的浮点数处理单元性能要好于 CPU ,因此使用 Cuda 来完成本实验是个更佳的选择。