并行编程实验指导

实验项目名称：并行编程实验指导-CUDA

姓名：江家玮

班级：计科2204班

学号：22281188

自我评价：

在本次并行编程实验中，我通过学习和应用CUDA函数库，深入理解了并行程序的工作原理和实现方法。整个实验过程包括Hello World程序、GPU中的线程、性能评估和矩阵乘法，使我熟悉了CUDA内存分配、数据传输和内核函数执行等基本操作。虽然在实验初期遇到了一些问题，但通过不断学习和调整，最终成功完成了所有实验项目。这次实验不仅提升了我的CUDA编程能力，还让我更好地理解了高性能计算的实际应用，对我今后的学习和研究有很大帮助。

成绩：

北京交通大学计算机科学与技术学院

实验一

```cpp
#include <cstdio>
int const N = 16;
int const blocksize = 16;
__global__
void hello(char* a, int const* b) {
 a[threadIdx.x] += b[threadIdx.x];
}
int main() {
 char a[N] = "Hello \0\0\0\0\0\0";
 int b[N] = {15, 10, 6, 0, -11, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
 char* ad;
 int* bd;
 int const csize = N * sizeof(char);
 int const isize = N * sizeof(int);
 printf("%s", a);
 cudaMalloc(&ad, csize);
 cudaMalloc(&bd, isize);
 cudaMemcpy(&ad, a, csize, cudaMemcpyKind::cudaMemcpyHostToDevice);
 cudaMemcpy(&bd, b, csize, cudaMemcpyKind::cudaMemcpyHostToDevice);
 dim3 dimBlock(blocksize, 1);
 dim3 dimGrid(1, 1);
 hello<<<dimGrid, dimBlock>>>(ad, bd);
 cudaMemcpy(&ad, a, csize, cudaMemcpyKind::cudaMemcpyDeviceToHost);
 cudaFree(ad);
 cudaFree(bd);
 printf("%s\n", a);
 return 0;
}
```

1.1 运行结果

```
[bjtu1@login02 22281188]$ sbatch submit.sh ./helloworldCUDA
Submitted batch job 36178
[bjtu1@login02 22281188]$ nvcc -o helloworldCUDA helloworldCUDA.cu
[bjtu1@login02 22281188]$ tail -f job.out
Hello World!
```

1.2 代码分析

这段CUDA代码通过在GPU上并行运行一个内核函数 `hello`，将两个数组的对应元素相加，最后将结果从GPU拷贝回主机，并打印结果字符串。具体步骤如下：

1. 定义了两个常量 `N` 和 `blocksize`，均为16。
2. 内核函数 `hello` 在GPU上执行，将字符数组 `a` 和整数数组 `b` 的对应元素相加，并将结果存储在字符数组 `a` 中。
3. `main` 函数初始化了字符数组 `a` 和整数数组 `b`，并打印初始的字符串 `a`。
4. 在GPU上分配字符数组和整数数组的内存，并将主机上的数据拷贝到设备内存。
5. 启动CUDA内核，执行并行计算。
6. 将计算结果从设备内存拷贝回主机，并释放设备内存。
7. 打印结果字符串 `a`。

1.3 实验结果分析

根据代码中的操作，初始的字符串`a`为"Hello "，整数数组`b`包含一些特定的值。执行内核函数后，字符数组`a`中的每个字符被对应的整数数组`b`中的值进行加法操作。具体如下：

- 'H' + 15 = 'W' (ASCII: 72 + 15 = 87)
- 'e' + 10 = 'o' (ASCII: 101 + 10 = 111)
- 'l' + 6 = 'r' (ASCII: 108 + 6 = 114)
- 'l' + 0 = 'l' (ASCII: 108 + 0 = 108)
- 'o' - 11 = 'd' (ASCII: 111 - 11 = 100)
- ' ' + 1 = '!' (ASCII: 32 + 1 = 33)

其他字符保持不变。因此，最终结果字符串为“Hello World!”，这与图中的输出相符。

1.4 实验中出现的问题

这个实验主要是熟悉一些命令，刚开始输入`tail-f job.out`的时候，输出的结果非常不稳定，有时候只输出`hello`或者是`hello hello`之后我通过输入`squeue-u bitul`查看了正在运行的进程，之后我发现可能是上面的进程没有结束，和下面的进程一起输出了，因此我在实验后面增加了`scancel <进程号>`后就没有再出现类似情况出现。

1.5 实验体会

通过这个实验，我熟悉了基本命令行输入的语句：

`sbatch submit.sh ./helloworldCUDA`

`nvcc -o helloworldCUDA helloworldCUDA.cu`

`tail -f job.out`

`scancel <进程号>`

实验二

```cpp
//实验二增加saxpy函数后代码
#include <cstdio>
#include <cstdlib>

__global__
void saxpy(int n, float a, float const* x, float* y) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride) {
        y[i] = a * x[i] + y[i];
    }
}

int main() {
    int const N = 1 << 20;
    size_t bytes = N * sizeof(float);
    float* x = (float*) malloc(bytes);
    float* y = (float*) malloc(bytes);
    float* d_x;
    float* d_y;
    cudaMalloc(&d_x, bytes);
    cudaMalloc(&d_y, bytes);

    for (int i = 0; i < N; ++i) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    cudaMemcpy(d_x, x, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, bytes, cudaMemcpyHostToDevice);

    saxpy<<<(N + 255) / 256, 256>>>(N, 2.0, d_x, d_y);

    cudaMemcpy(y, d_y, bytes, cudaMemcpyDeviceToHost);

    float maxError = 0.0f;
    for (int i = 0; i < N; ++i) {
        float error = abs(y[i] - 4.0f);
        if (error > maxError) {
            maxError = error;
        }
    }

    printf("Max error: %f\n", maxError);

    cudaFree(d_x);
    cudaFree(d_y);

    free(x);
    free(y);

    return 0;
```

```
53 }
```

2.1 运行结果

```
[bjtu1@login02 22281188]$ sbatch submit.sh ./saxpy
Submitted batch job 36133
[bjtu1@login02 22281188]$ nvcc -o saxpy saxpy.cu
[bjtu1@login02 22281188]$ tail -f job.out
Max error: 0.000000
```

2.2 代码分析

1. **CUDA 内核函数** `saxpy`:
   - `int index = blockIdx.x * blockDim.x + threadIdx.x;` 计算每个线程的全局索引。
   - `int stride = blockDim.x * gridDim.x;` 计算线程步长，使得每个线程可以处理多个数组元素。
   - `for (int i = index; i < n; i += stride) { y[i] = a * x[i] + y[i]; }` 遍历数组，执行`saxpy`操作，即对每个元素进行标量乘积和加法。
2. **主函数** `main`:
   - `float* x = (float*) malloc(bytes);` 和 `float* y = (float*) malloc(bytes);` 分配主机内存。
   - `cudaMalloc(&d_x, bytes);` 和 `cudaMalloc(&d_y, bytes);` 在设备上分配内存。
   - `cudaMemcpy(d_x, x, bytes, cudaMemcpyHostToDevice);` 和 `cudaMemcpy(d_y, y, bytes, cudaMemcpyHostToDevice);` 将数组从主机拷贝到设备。
   - `saxpy<<<(N + 255) / 256, 256>>>(N, 2.0, d_x, d_y);` 启动CUDA内核，配置的线程块和线程数确保所有元素都能被处理。
   - `cudaMemcpy(y, d_y, bytes, cudaMemcpyDeviceToHost);` 将计算结果从设备拷贝回主机。
   - 遍历数组 `y`，计算并输出最大误差：`float error = abs(y[i] - 4.0f); if (error > maxError) { maxError = error; }`
   - `cudaFree(d_x);` 和 `cudaFree(d_y);` 释放设备内存。
   - `free(x);` 和 `free(y);` 释放主机内存。

2.2.1 代码修改

1. **实现saxpy内核函数**:
   - 原始代码中，内核函数`void saxpy(int n, float a, float const* x, float* y)`是空的，没有实现具体的向量加法操作。
   - 修改后的代码中实现了saxpy操作，遍历数组`x`和`y`，将`y[i]`更新为`a * x[i] + y[i]`。
2. **并行计算的合理分配**:
   - 我修改后的代码通过计算索引和步长，使得每个线程可以处理多个元素，充分利用GPU的并行计算能力。

2.3 实验结果分析

- **原始结果**: `Max error: 2.000000`
  - 原始代码未对数组`y`进行修改，所有元素仍然保持初始值2.0f，因此计算出的误差为2.0。
- **修改后结果**: `Max error: 0.000000`
  - 修改后的代码正确实现了saxpy操作，修改后的代码实现了`saxpy`内核函数，对每个元素执行`y[i] = 2 * x[i] + y[i]`，即`y[i] = 2 * 1.0f + 2.0f = 4.0f`。由于所有元素都被

正确更新，因此最大误差为0。

2.4 实验中出现的问题

刚开始输出的是`Max error: 2.000000`,而要求是改为`Max error: 0.000000`,而后根据老师文档上的提示和仔细阅读代码，发现这是因为所有元素都保持初始值，未进行实际计算。最后通过修改`saxpy`函数中`y[i] = 2 * x[i] + y[i]`中实现输出要求

2.5 实验体会

我通过实现和优化SAXPY函数，加深了对CUDA并行计算的理解。代码中涉及的每个步骤，包括内存分配、数据传输和内核函数执行，都使我进一步掌握了CUDA编程的基本流程和要点。

实验三

```cpp
#include <cstdio>
#include <cstdlib>
#include <cuda_runtime.h>

size_t const N = 1048576;
size_t const bytes = N * sizeof(int);

int main() {
    int* h_a = (int*)malloc(bytes);
    int* d_a;
    float duration;
    cudaEvent_t start, end;
    cudaEventCreate(&start);
    cudaEventCreate(&end);

    cudaMalloc(&d_a, bytes);
    memset(h_a, 0, bytes);

    cudaEventRecord(start);

    cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(h_a, d_a, bytes, cudaMemcpyDeviceToHost);

    cudaEventRecord(end);
    cudaEventSynchronize(end);

    cudaEventElapsedTime(&duration, start, end);

    cudaFree(d_a);
    free(h_a);

    printf("Takes %f ms\n", duration);

    return 0;
}
```

3.1 运行结果

```
[bjtu1@login02 22281188]$ sbatch submit.sh ./memoryTransferOverhead
Submitted batch job 36241
[bjtu1@login02 22281188]$ nvcc -o memoryTransferOverhead memoryTransferOverhead.cu
[bjtu1@login02 22281188]$ tail -f job.out
Takes 2.174016 ms
```

3.2 代码分析

创建和记录CUDA事件:

- `cudaEventCreate(&start);`: 创建开始事件。
- `cudaEventCreate(&end);`: 创建结束事件。
- `cudaEventRecord(start);`: 记录开始事件。
- `cudaEventRecord(end);`: 记录结束事件。
- `cudaEventSynchronize(end);`: 等待结束事件完成。

设备内存分配和数据传输:

- `cudaMalloc(&d_a, bytes);`: 在设备上分配内存。
- `memset(h_a, 0, bytes);`: 将主机内存设置为0。
- `cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);`: 将数据从主机拷贝到设备。
- `cudaMemcpy(h_a, d_a, bytes, cudaMemcpyDeviceToHost);`: 将数据从设备拷贝回主机。

计算和输出时间:

- `cudaEventElapsedTime(&duration, start, end);`: 计算开始和结束事件之间的时间差。
- `printf("Takes %f ms\n", duration);`: 输出持续时间。

释放内存:

- `cudaFree(d_a);`: 释放设备内存。
- `free(h_a);`: 释放主机内存。

3.2.1 代码修改

添加CUDA事件来测量时间:

- 原始代码中没有测量CUDA操作的时间。
- 修改后的代码添加了CUDA事件来记录和计算操作的持续时间。

创建和销毁CUDA事件:

- 修改后的代码中使用`cudaEventCreate`创建开始和结束事件，并在程序结束时销毁它们。

记录和同步事件:

- 修改后的代码中使用`cudaEventRecord`来记录开始和结束事件，并使用`cudaEventSynchronize`确保结束事件已完成。

计算并输出时间:

- 修改后的代码中使用`cudaEventElapsedTime`来计算两个事件之间的时间差，并输出结果。

3.3 实验结果分析

原始结果: `Takes 0.000000ms`

- 原始代码没有测量时间的操作，因此持续时间始终为0。

修改后结果: `Takes 2.174016ms`

- 修改后的代码通过记录开始和结束事件，准确测量了数据从主机到设备，再从设备回到主机的时间。
- 这个时间反映了内存分配、数据传输等操作的总时间，更能体现CUDA操作的真实性能。

3.4实验体会

我通过添加CUDA事件来测量内存操作的持续时间，让我学习到了CUDA编程中的性能优化方法。通过时间测量，可以更直观地看到不同操作的性能的好坏。

实验四

```cpp
#include <cstdio>
#include <cstdlib>

size_t const BLOCK_SIZE = 16;

__global__
void mat_mult(float const* A, float const* B, float* C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y; // 计算行索引
    int col = blockIdx.x * blockDim.x + threadIdx.x; // 计算列索引

    float sum = 0.0f;
    for (int n = 0; n < N; ++n) {
        sum += A[row * N + n] * B[n * N + col];
    }
    C[row * N + col] = sum;
}

void mat_mult_cpu(float const* A, float const* B, float* C, int N) {
    for (int row = 0; row < N; ++row) {
        for (int col = 0; col < N; ++col) {
            float sum = 0.0f;
            for (int n = 0; n < N; ++n) {
                sum += A[row * N + n] * B[n * N + col];
            }
            C[row * N + col] = sum;
        }
    }
}

int main() {
    int K = 100;
    int N = K * BLOCK_SIZE;
    float* hA = new float[N * N];
    float* hB = new float[N * N];
    float* hC = new float[N * N];

    for (int j = 0; j < N; ++j) {
        for (int i = 0; i < N; ++i) {
            hA[j * N + i] = 2.f * (float)(j + i);
            hB[j * N + i] = 1.f * (float)(j - i);
        }
    }

    size_t size = N * N * sizeof(float);
    float *dA, *dB, *dC;
    cudaMalloc(&dA, size);
    cudaMalloc(&dB, size);
    cudaMalloc(&dC, size);

    dim3 threadBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid(K, K);

```

```
    cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice);

    mat_mult<<<grid, threadBlock>>>(dA, dB, dC, N);

    if (cudaPeekAtLastError() != cudaError::cudaSuccess) {
        fprintf(stderr, "CUDA error detected: \"%s\"\n",
                cudaGetErrorString(cudaGetLastError()));
        return EXIT_FAILURE;
    }

    float* C = new float[N * N];
    cudaMemcpy(C, dC, size, cudaMemcpyDeviceToHost);

    mat_mult_cpu(hA, hB, hC, N);

    for (int row = 0; row < N; ++row) {
        for (int col = 0; col < N; ++col) {
            if (C[row * N + col] != hC[row * N + col]) {
                fprintf(stderr, "Validation failed at row=%d, col=%d.\n",
row, col);
                delete[] hA;
                delete[] hB;
                delete[] hC;
                delete[] C;
                cudaFree(dA);
                cudaFree(dB);
                cudaFree(dC);
                return EXIT_FAILURE;
            }
        }
    }

    printf("OK!");

    delete[] hA;
    delete[] hB;
    delete[] hC;
    delete[] C;

    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);

    return EXIT_SUCCESS;
}
```

4.1 运行结果

```
[bjtu1@login02 22281188]$ sbatch submit.sh ./matrixMultiplication
Submitted batch job 36326
[bjtu1@login02 22281188]$ nvcc -o matrixMultiplication matrixMultiplication.cu
[bjtu1@login02 22281188]$ tail -f job.out
OK!
```

4.2 代码分析

4.2.1代码修改

CUDA内核函数 `mat_mult` 中的行列索引计算:

- 原始代码中的 `int row = 0;` 和 `int col = 0;` 未实现行和列的正确索引计算。
- 修改后的代码计算行索引为 `int row = blockIdx.y * blockDim.y + threadIdx.y;`，列索引为 `int col = blockIdx.x * blockDim.x + threadIdx.x;`。

4.3 实验结果分析

原始结果: `Validation failed`

- 原始代码未正确计算行和列索引，所有线程都操作同一位置，导致CUDA内核函数执行的结果不正确，最终导致验证失败。

修改后结果: `OK!`

- 修改后的代码通过`blockIdx`和`threadIdx`计算每个线程对应的矩阵位置，从而正确实现并行矩阵乘法，计算了行和列索引，并正确执行了CUDA内核函数，实现了正确的矩阵乘法操作。验证通过，输出`OK!`。

4.4 实验中出现问题

通过阅读原始代码发现未正确计算行和列索引，因此我通过修改`blockIdx`和`threadIdx`计算每个线程对应的矩阵位置，从而满足输出`OK!`的实验要求。

4.5 实验体会

我通过实现并优化矩阵乘法，进一步理解了CUDA的并行计算模型以及块和线程的配置方式。通过合理的块和线程分配，可以显著提高矩阵乘法的计算效率。