CS205 C/C++ Programming -- Project 5 GPU Acceleration with CUDA

Name: 陈长信(Chen Changxin)

SID: 12210731

Part1--问题分析

本项目旨在探究GPU在矩阵计算中的加速效果，本项目使用NVIDIA GPU，服务器各参数如下所示：

- CPU: Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz, 24 Cores
- Memory: 128GB
- GPU: NVIDIA GeForce RTX 2080 Ti x 4
- OS: Ubuntu 22.04.4
- GCC: 11.4.0
- make: 4.3
- cmake: 3.22.1

项目具体需求：

1. 利用GPU计算包含标量的矩阵运算 `B = a A + b`，默认A,B有相同的矩阵大小。
2. 利用CPU中的OpenBLAS以及GPU中的cuBLAS库，比较矩阵乘法在使用NVIDIA中cuBLAS库加速后的表现情况，其中OpenBLAS使用cblas_sgemm()函数，而cuBLAS使用cublasSgemm()函数。
3. 探究了GPU中矩阵乘法各个步骤的时间消耗并尝试进行了一些优化。

Part2--代码实现以及结构

基础代码结构

1. 定义矩阵类，其中data和data_device指针分别代表指向CPU以及GPU的矩阵内存空间。

```
typedef struct
{
    size_t rows;
    size_t cols;
    float * data; // CPU memory
    float * data_device; // GPU memory
} Matrix;
```

2. 定义create_Matrix,freeMatrix,setMatrix函数，**本函数不作为本项目主体，故使用shiqiYu课件里的example，代码来源于**

   https://github.com/ShiqiYu/CPP/blob/main/week08/examples/cuda/matadd.cu。

   这些函数仅作为矩阵的初始化、释放内存空间、分配基础值使用。

```
Matrix * createMatrix(size_t r, size_t c)
{
    size_t len = r * c;
    if(len == 0)
    {
        fprintf(stderr, "Invalid size. The input should be > 0.\n");
        return NULL;
    }
    Matrix * p  = (Matrix *) malloc(sizeof(Matrix));
    if (p == NULL)
    {
        fprintf(stderr, "Allocate host memory failed.\n");
        goto ERR_TAG;
    }
    p->rows = r;
    p->cols = c;
    p->data = (float*)malloc(sizeof(float)*len);
    if(p->data == NULL)
    {
        fprintf(stderr, "Allocate host memory failed.\n");
        goto ERR_TAG;
    }
    //给GPU CUDA分配内存空间，思路同CPU
    if (cudaMalloc (&p->data_device, sizeof(float) * len) != cudaSuccess)
    {
        fprintf(stderr, "Allocate device memory failed.\n");
        goto ERR_TAG;
    }
    return p;
ERR_TAG:
    if(p && p->data) free(p->data);
    if(p) free(p);
    return NULL;
}
void freeMatrix(Matrix ** pp)
{
    if(pp == NULL) return;
    Matrix * p = *pp;
    if(p != NULL)
    {
        if(p->data) free(p->data);
        if(p->data_device) cudaFree(p->data_device);
    }
    *pp = NULL;
}
```

```
// A simple function to set all elements to the same value
bool setMatrix(Matrix * pMat, float val)
{
    if(pMat == NULL)
    {
        fprintf(stderr, "NULL pointer.\n");
        return false;
    }
    size_t len = pMat->rows * pMat->cols;
    for(size_t i = 0; i < len; i++)
        pMat->data[i] = val;

    return true;
}
```

矩阵计算函数

含标量的矩阵运算

在shiqiYu的example中matadd.cu的基础上，添加了 `B = a A + b` 矩阵运算，其中A,B为相同大小的矩阵，a,b以标量的形式存在。

1. 在CPU中实现

```
bool scaleAddCPU(const Matrix * pMatA, Matrix * pMatB, float a, float b)
{
    if (pMatA == NULL || pMatB == NULL)
    {
        fprintf(stderr, "Null pointer.\n");
        return false;
    }
    if (pMatA->rows != pMatB->rows || pMatA->cols != pMatB->cols)
    {
        fprintf(stderr, "The matrices are not in the same size.\n");
        return false;
    }

    size_t len = pMatA->rows * pMatA->cols;
    for (size_t i = 0; i < len; i++)
        pMatB->data[i] = a * pMatA->data[i] + b;

    return true;
}
```

2. 在GPU中实现

首先定义 `__global__ void scaleAddKernel` 函数，与普通矩阵乘法不同的是，这里添加了标量来定义矩阵的前参数。

```
__global__ void scaleAddKernel(const float *input, float *output, float a, float b,
size_t len)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i < len)
        output[i] = a * input[i] + b;
}
```

定义GPU中的 `scaleAddGPU` 函数

```
bool scaleAddGPU(const Matrix * pMatA, Matrix * pMatB, float a, float b)
{
    if (pMatA == NULL || pMatB == NULL)
    {
        fprintf(stderr, "Null pointer.\n");
        return false;
    }
    if (pMatA->rows != pMatB->rows || pMatA->cols != pMatB->cols)
    {
        fprintf(stderr, "The matrices are not in the same size.\n");
        return false;
    }

    size_t len = pMatA->rows * pMatA->cols;

    //进行memory的copy，相当于将数据载入到GPU中
    cudaMemcpy(pMatA->data_device, pMatA->data, sizeof(float) * len,
cudaMemcpyHostToDevice);
    scaleAddKernel<<<(len + 255) / 256, 256>>>(pMatA->data_device, pMatB-
>data_device, a, b, len);

    cudaError_t ecode;
    if ((ecode = cudaGetLastError()) != cudaSuccess)
    {
        fprintf(stderr, "CUDA Error: %s\n", cudaGetErrorString(ecode));
        return false;
    }
    //计算完之后从GPU拿回来
    cudaMemcpy(pMatB->data, pMatB->data_device, sizeof(float) * len,
cudaMemcpyDeviceToHost);

    return true;
}
```

3. 在主函数中进行测试，并且进行时间测试。

这里矩阵的大小设为 `size = 4096` ，标量默认为1和2。

使用 `nvcc matadd.cu` 命令行进行编译。

```
Matrix * pMatA = createMatrix(4096, 4096);
Matrix * pMatB = createMatrix(4096, 4096);

setMatrix(pMatA, 1.1f);

float a = 2.0f;
float b = 1.0f;

TIME_START
scaleAddCPU(pMatA, pMatB, a, b);
TIME_END(scaleAddCPU)
printf("  Result = [%.1f, ..., %.1f]\n", pMatB->data[0], pMatB->data[pMatB-
>rows*pMatB->cols-1]);

TIME_START
scaleAddGPU(pMatA, pMatB, a, b);
TIME_END(scaleAddGPU)
printf("  Result = [%.1f, ..., %.1f]\n", pMatB->data[0], pMatB->data[pMatB-
>rows*pMatB->cols-1]);

freeMatrix(&pMatA);
freeMatrix(&pMatB);
```

运行结果如下图所示

```
$ nvcc matadd.cu
$ ./a.out
You have 4 cuda devices.
You are using device 2.
scaleAddCPU Time = 59.414000 ms.
  Result = [3.2, ..., 3.2]
scaleAddGPU Time = 23.438000 ms.
  Result = [3.2, ..., 3.2]
```

可见，在运行加法时候GPU中的运行速度较快。

矩阵乘法速度的比较

1. 使用CPU中的OpenBLAS进行矩阵乘法，使用cblas_sgemm进行快速矩阵乘法计算。

```
void matmulCPU(const Matrix * A, const Matrix * B, Matrix * C)
{
    const float alpha = 1.0f;
    const float beta = 0.0f;

    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                A->rows, B->cols, A->cols,
                alpha, A->data, A->cols,
                B->data, B->cols,
                beta, C->data, C->cols);
}
```

2. 使用GPU中的cuBLAS进行矩阵乘法，使用cublasSgemm()进行计算。

```
void matmulGPU(const Matrix * A, const Matrix * B, Matrix * C)
{
    cublasHandle_t handle;
    cublasCreate(&handle);

    const float alpha = 1.0f;
    const float beta = 0.0f;

    // Copy data to GPU
    cudaMemcpy(A->data_device, A->data, A->rows * A->cols * sizeof(float),
cudaMemcpyHostToDevice);
    cudaMemcpy(B->data_device, B->data, B->rows * B->cols * sizeof(float),
cudaMemcpyHostToDevice);

    // Perform matrix multiplication: C = alpha * A * B + beta * C
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
                A->rows, B->cols, A->cols,
                &alpha,
                A->data_device, A->rows,
                B->data_device, B->rows,
                &beta,
                C->data_device, C->rows);

    cudaDeviceSynchronize(); // Ensure the kernel has completed

    // Copy result back to CPU
    cudaMemcpy(C->data, C->data_device, C->rows * C->cols * sizeof(float),
cudaMemcpyDeviceToHost);

    cublasDestroy(handle);
}
```

3. 同样的，在主函数中进行矩阵构造，并进行时间比较。

同样适用 `size = 4096` 进行时间测试，这里忽略具体的值所带来的运算速度差异，专注于探究硬件资源带来的计算差异。

```
struct timeval t_start, t_end;
double elapsedTime = 0;

size_t size = 4096;
Matrix * A = createMatrix(size, size);
Matrix * B = createMatrix(size, size);
Matrix * C = createMatrix(size, size);

//设置为相同的值，这里A都设为1，B都设为2
setMatrix(A, 1.0f);
setMatrix(B, 2.0f);

// Measure CPU matrix multiplication time
TIME_START
matmulCPU(A, B, C);
TIME_END(matmulCPU)
printf("  Result (CPU) = [%.1f, ..., %.1f]\n", C->data[0], C->data[C->rows * C->cols
- 1]);

// Measure GPU matrix multiplication time
TIME_START
matmulGPU(A, B, C);
TIME_END(matmulGPU)
printf("  Result (GPU) = [%.1f, ..., %.1f]\n", C->data[0], C->data[C->rows * C->cols
- 1]);

freeMatrix(&A);
freeMatrix(&B);
freeMatrix(&C);
```

使用命令行执行

```
nvcc -o matmul matmul.cu -lcublas -lcuda -lopenblas
```

结果如下所示

```
$ nvcc -o matmul matmul.cu -lcublas -lcuda -lopenblas
$ ./matmul
You have 4 cuda devices.
You are using device 0.
matmulCPU Time = 99.475000 ms.
  Result (CPU) = [8192.0, ..., 8192.0]
matmulGPU Time = 101.354000 ms.
  Result (GPU) = [8192.0, ..., 8192.0]
```

我们发现，结果并不如我们所期待，CPU反而快于GPU，这是为什么呢?

接着我测试了 `size = 2048`

```
$ nvcc -o matmul matmul.cu -lcublas -lcuda -lopenblas
$ ./matmul
You have 4 cuda devices.
You are using device 0.
matmulCPU Time = 37.841000 ms.
  Result (CPU) = [4096.0, ..., 4096.0]
matmulGPU Time = 55.512000 ms.
  Result (GPU) = [4096.0, ..., 4096.0]
```

同样发现CPU执行更快。

我又（没忍住）测试了 `size = 8192`

```
$ nvcc -o matmul matmul.cu -lcublas -lcuda -lopenblas
$ ./matmul
You have 4 cuda devices.
You are using device 0.
matmulCPU Time = 477.180000 ms.
  Result (CPU) = [16384.0, ..., 16384.0]
matmulGPU Time = 296.484000 ms.
  Result (GPU) = [16384.0, ..., 16384.0]
```

这时候GPU时间更快了。基于上述现象，我进行了更多的探究。

Part3--其他探究以及运算优化

GPU矩阵乘法时间实际分布

我认为上述现象是由于GPU的Mul函数中包含了一些copy数据，传输数据的操作，导致在矩阵size相对较小的时候表现甚至不如CPU，于是我对GPU矩阵乘法函数中所有可能产生时间消耗的部分进行了分部时间测试。

```
struct timeval t_start, t_end;
double elapsedTime = 0;

TIME_START
cublasHandle_t handle;
cublasCreate(&handle);
TIME_END(createhandle)

// Copy data to GPU
TIME_START
```

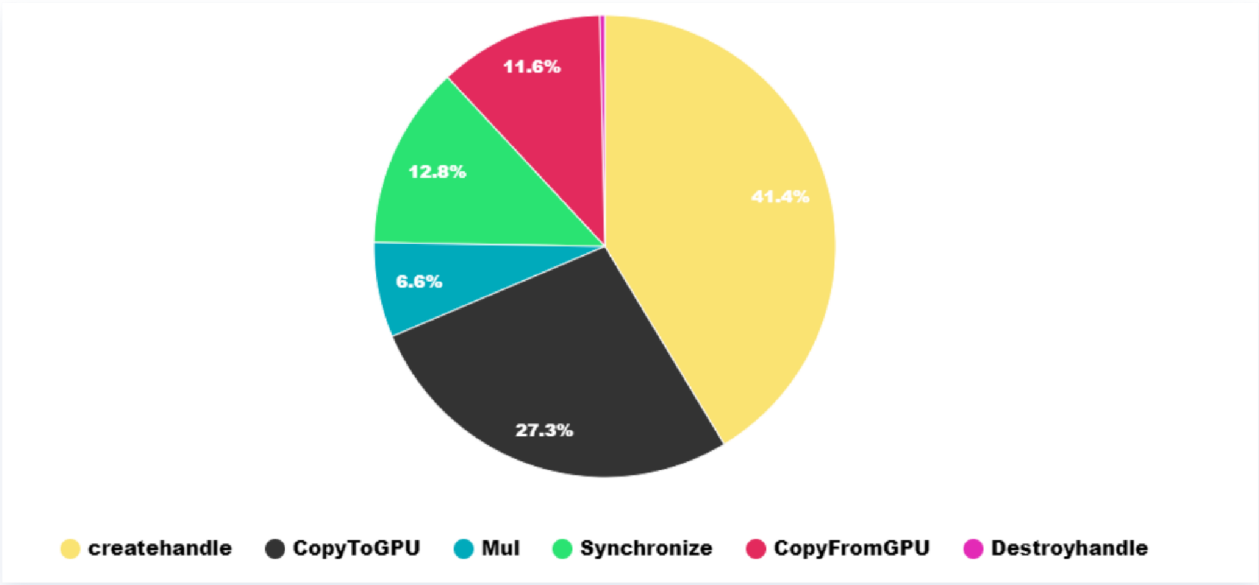
```
cudaMemcpy(A->data_device, A->data, A->rows * A->cols * sizeof(float),
cudaMemcpyHostToDevice);
cudaMemcpy(B->data_device, B->data, B->rows * B->cols * sizeof(float),
cudaMemcpyHostToDevice);
TIME_END(CopytoGPU)
// Start measuring time after data is copied

// Perform matrix multiplication: C = alpha * A * B + beta * C
TIME_START
cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
            A->rows, B->cols, A->cols,
            &alpha,
            A->data_device, A->rows,
            B->data_device, B->rows,
            &beta,
            C->data_device, C->rows);
TIME_END(RealMultiplication)

TIME_START
cudaDeviceSynchronize(); // Ensure the kernel has completed
TIME_END(Synchronize)

// Copy result back to CPU
TIME_START
cudaMemcpy(C->data, C->data_device, C->rows * C->cols * sizeof(float),
cudaMemcpyDeviceToHost);
TIME_END(CopyfromGPU)

TIME_START
cublasDestroy(handle);
TIME_END(Destroyhandle)
```

运行结果如下

```
$ nvcc -o matmul matmul.cu -lcublas -lcuda -lopenblas
$ ./matmul
You have 4 cuda devices.
You are using device 0.
matmulCPU Time = 101.192000 ms.
  Result (CPU) = [8192.0, ..., 8192.0]
createhandle Time = 39.220000 ms.
CopytoGPU Time = 25.868000 ms.
RealMultiplication Time = 6.295000 ms.
Synchronize Time = 12.149000 ms.
CopyfromGPU Time = 10.979000 ms.
Destroyhandle Time = 0.352000 ms.
matmulGPU Time = 95.217000 ms.
  Result (GPU) = [8192.0, ..., 8192.0]
```

果然不出我们所料，实际上真正运行矩阵乘法时间是非常短的，只有6.29ms，真正长的时间花费在了createhandle,Copy等操作上了。

更直观的，我做出了饼图，如下图所示：



其中createhandle的时间明显是最长的，那么什么是createhandle呢?

`handle` 是 cuBLAS 库的句柄，它的作用是管理 cuBLAS 库的内部状态。主要目的是为了确保不同的 cuBLAS 函数调用可以共享相同的上下文，并提高效率。 `handle` 保存了 cuBLAS 库的状态信息，包括配置、调用模式等。

其中我认为创建 `handle` 的时间较长可能有以下几个原因：

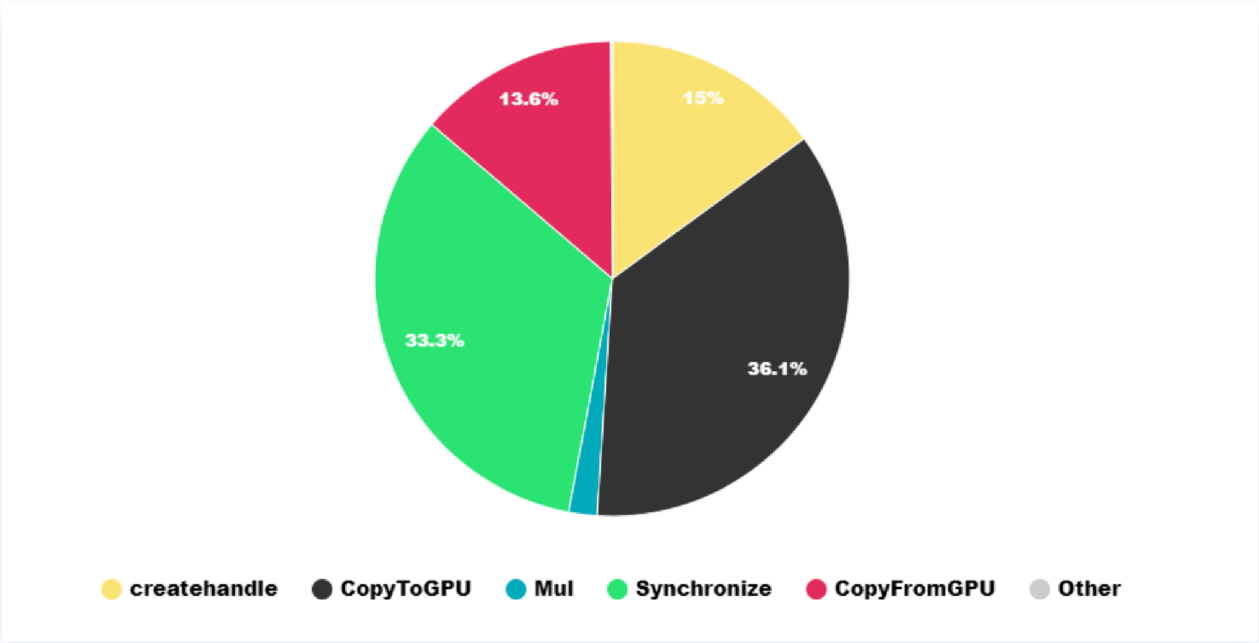
1. **初始化过程**：创建 `handle` 时，cuBLAS 库可能需要进行一些初始化操作，包括分配内存、初始化状态等。
2. **环境配置**：创建 `handle` 时，cuBLAS 库可能需要根据当前环境进行一些配置，比如选择合适的算法、优化选项等。

所以在创建handle时候实际上会花费一些时间，但是既然如此，使用GPU就要创建句柄，也是消耗时间，为什么还要使用GPU呢?

我再次（**没忍住**）对 `size = 8192` 进行了测试，得到下列结果

```
$ nvcc -o matmul matmul.cu -lcublas -lcuda -lopenblas
$ ./matmul
You have 4 cuda devices.
You are using device 0.
matmulCPU Time = 471.803000 ms.
  Result (CPU) = [16384.0, ..., 16384.0]
createhandle Time = 42.684000 ms.
CopytoGPU Time = 102.880000 ms.
RealMultiplication Time = 5.468000 ms.
Synchronize Time = 95.003000 ms.
CopyfromGPU Time = 38.901000 ms.
Destroyhandle Time = 0.351000 ms.
matmulGPU Time = 285.568000 ms.
  Result (GPU) = [16384.0, ..., 16384.0]
```

我发现createhandle的time并没有显著上升，每一次进行矩阵乘法又只用创建一次，所以在进行更大矩阵计算上GPU依旧有着显著的优势，但是取而代之的是拷贝以及同步的操作，他们的时间以及占比都显著上升。



实际上在不断提高size之后，一直占据主要时间占比的都是Copy操作以及同步操作。真正的矩阵计算占据很少的运算时间，Copy操作是数据的拷贝，随着数据量的增大当然需要更多的时间，那么什么是Synchronize()操作呢，这个操作是必要的吗?

`cudaDeviceSynchronize()` 是一个同步函数，它的作用是确保在当前设备上所有先前的 CUDA 函数调用都已经执行完成。这意味着在调用 `cudaDeviceSynchronize()` 之前的所有 CUDA 函数都已经执行完毕，而且设备已经完成了所有的任务。这个函数通常用于检查 CUDA 函数是否成功执行，并等待它们完成，以确保后续代码能够安全地使用设备上的结果。所以说这个操作我认为是必要的。

当然，在计算量增大之后，需要同步的数据就更多了，所以自然执行时间就更长了，所以我认为还能对GPU矩阵乘法进行一下运算优化。

GPU计算优化

既然是在Copy操作和同步操作花费时间较多，又由于同步操作较难提升，毕竟涉及到较多的内部机制以及环境配置，这里主要从Copy操作入手

考虑使用异步内存操作来替代，这样运行数据传输和计算重叠，提高效率。

```
cudaMemcpyAsync(A->data_device, A->data, A->rows * A->cols * sizeof(float),
cudaMemcpyHostToDevice);
```

不出所料，虽然说在 `size = 4096` 的环境下提高不大，但是随着size的增大，在Copy操作上的时间果然有所缩短。

```
//同步(size = 16384)
CopytoGPU Time = 416.074000 ms.

//异步(size = 16384)
CopytoGPU Time = 397.871000 ms.
```

Part4--项目总结

1. 本项目使用OpenBLAS以及cuBLAS矩阵计算库，主要专注于在硬件上带来的矩阵计算速度的不同，而非算法本身。
2. 本项目通过对GPU的应用，比较了矩阵乘法在GPU以及CPU上的表现情况，加深了对于多核，CUDA等高性能计算的理解。

Part5--补充说明

1. 本项目部分代码框架来自ChatGPT以及shiqiYu的example，核心代码均为本人撰写。