# CUDA实现卷积

## 1. 实验内容

本实验要求我们实现卷积操作，并且通过CUDA编程来实现并行化。

## 2. 实验目的

1）熟练使用CUDA编程环境

2）加深对于并行编程的相关理解。

3）熟悉使用CUDA并行化卷积过程中的运算来提高计算效率，降低时间成本

3. 实验环境

Operating System: Linux (Ubuntu 18.04)

CUDA Version: 11.2

Driver Version: 460.91.03

Programming Language: CUDA C++

GPU: NVIDIA GeForce RTX 2080 Ti (11019MiB)

## 4. 实验步骤

4.1. GPU信息校验：

int main()

{

    int deviceCount;

    cudaGetDeviceCount(&deviceCount);

    for(int i=0;i<deviceCount;i++)

    {

        cudaDeviceProp devProp;

        cudaGetDeviceProperties(&devProp, i);

        cout << "======================================================" << endl;

        cout << "=Sequential && Parallel Experiments CUDA: Convolution=" << endl;

        cout << "======================================================" << endl;

        cout << "使用GPU device " << i << ": " << devProp.name << endl;

        cout << "设备全局内存总量： " << devProp.totalGlobalMem / 1024 / 1024 << "MB" << endl;

        cout << "SM的数量：" << devProp.multiProcessorCount << endl;

        cout << "每个线程块的共享内存大小：" << devProp.sharedMemPerBlock / 1024.0 << " KB" << endl;

        cout << "每个线程块的最大线程数：" << devProp.maxThreadsPerBlock << endl;

        cout << "设备上一个线程块（Block）种可用的32位寄存器数量： " << devProp.regsPerBlock << endl;

        cout << "每个EM的最大线程数：" << devProp.maxThreadsPerMultiProcessor << endl;

        cout << "每个EM的最大线程束数：" << devProp.maxThreadsPerMultiProcessor / 32 << endl;

        cout << "设备上多处理器的数量： " << devProp.multiProcessorCount << endl;

        cout << "最高频率：" << devProp.clockRate / 1000000.0 << "GHz" << endl;

        cout << "显存位宽：" << devProp.memoryBusWidth << "bit" << endl;

        cout << "显存频率：" << devProp.memoryClockRate / 1000000.0 << "GHz" << endl;

         cout << "======================================================" << endl;

    }

    return 0;

}

4.2. CPU内存与GPU显存数据拷贝校验：

  float\*A, \*Ad, \*B, \*Bd, \*C, \*Cd;

    int n = 1024 \* 1024;

    int size = n \* sizeof(float);

*// CPU端分配内存*

    A = (float\*)malloc(size);

    B = (float\*)malloc(size);

    C = (float\*)malloc(size);

*// 初始化数组*

    for(int i=0;i<n;i++)

    {

        A[i] = 90.0;

        B[i] = 10.0;

    }

*// GPU端分配内存*

    cudaMalloc((void\*\*)&Ad, size);

    cudaMalloc((void\*\*)&Bd, size);

    cudaMalloc((void\*\*)&Cd, size);

*// CPU的数据拷贝到GPU端*

    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);

    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

4.3. 二维卷积在Sequential与Parallel的相关实现：

4.3.1 Sequential Convolution

void conv2d(double \*\**array*, double \*\**kernel*, double \*\**result*, int *array\_size*, int *kernel\_size*)

{

    if (array\_size < kernel\_size)

        return;

    for (int i = 0; i < array\_size - kernel\_size + 1; i++)

        for (int j = 0; j < array\_size - kernel\_size + 1; j++) {

            result[i][j] = 0;

            for (int n = 0; n < kernel\_size; n++)

                for (int m = 0; m < kernel\_size; m++)

                    result[i][j] += array[i + n][j + m] \* kernel[n][m];

        }

}

4.3.2 Parallel Convolution

\_\_global\_\_  void conv2d\_cuda(double \*\**array*, double \*\**kernel*, double \*\**result*, int *array\_size*, int *kernel\_size*)

{

    int idx = threadIdx.x + blockDim.x \* blockIdx.x;

    int idy = threadIdx.y + blockDim.y \* blockIdx.y;

    int result\_size = array\_size - kernel\_size + 1;

    if (idx < result\_size && idy < result\_size) {

        result[idx][idy] = 0;

        for (int i = 0; i < kernel\_size; i++)

            for (int j = 0; j < kernel\_size; j++)

                result[idx][idy] += array[idx + i][idy + j] \* kernel[i][j];

    }

}

## 5. 实验分析

5.1.分别设定不同情况下的卷积核与数值方阵验证卷积结果：

Matrix\_Size=512, Kernel\_Size=3,5,9,12,15

表格

描述已自动生成

Matrix\_Size=1024, Kernel\_Size=3,5,9,12,15

表格

描述已自动生成

Matrix\_Size=2048, Kernel\_Size=3,5,9,12,15

表格

描述已自动生成

Matrix\_Size=8192, Kernel\_Size=3,5,9,12,15

表格

描述已自动生成

5.2可视化结果分析：

Matrix\_Size=512, Kernel\_Size=3,5,9,12,15

Matrix\_Size=1024, Kernel\_Size=3,5,9,12,15

Matrix\_Size=2048, Kernel\_Size=3,5,9,12,15

Matrix\_Size=8192, Kernel\_Size=3,5,9,12,15

根据对处理不同的数值矩阵的大小以及卷积核的大小以及是否通过CUDA实现并行化的设置，运行程序得到程序运行的时间，再将获得的数据进行综合分析，最后得到如上所示的表格和折线图。根据实验结果，我们可以发现并行所用的时间显著小于串行所用时间，并且随着处理数量的增大，两者所用时间的差异越来越大，因此在实际的大规模化生产中，并行化能极大地提升程序运行的效率，节约时间成本。

## 6. 问题与讨论

**6.1为什么要用CUDA加速？**

在科学计算领域所要用到的计算往往不是我们熟知的普通矩阵，而是千维甚至万维的矩阵，而普通的CPU串行计算往往不能满足与科学计算所要求的性能。最好的例子就是深度学习，可以说深度学习起源于感知机但正真发展在于计算能力的提高，也就是显卡计算的兴起。深度学习的计算都是基于矩阵的计算，而普通一个识别手写数字的神经网络都有上千个隐含节点，故CUDA性能优化是一门重要的技术。

**6.2为什么显卡可以加速？**

首先，显卡可以加速最大的原因是其含有上千个CUDA核心，而CPU的核心往往都在各位数，这就决定了显卡可以高度并行的对一些高维度矩阵进行计算。CPU的I/O需要数百上千个周期，串行的计算大部分时间都消耗在I/O上，而显卡则不然，NVIDIA显卡的架构是类似于SIMD架构的SIMT架构，T指的是Threads，也就是单指令多线程架构，上一个线程在进行运算操作时下一个线程就开始I/O操作，类似于指令流水线的形式，当Threads数量足够多时，就可以屏蔽I/O所带来的大量开销，所以本身架构决定了CPU与显卡计算方式的区别。

CPU就好比是一个身强体壮的大汉一下可以挑100公斤粮食，显卡好比是只能挑起10公斤粮食的小矮人，但有20个。现在有2000公斤粮食，I/O相当于需要1分钟时间把粮食装进桶才能挑走（每次只能一个人装桶），运算时间相当于挑一次到目的地需要20分钟。CPU每趟需要21分钟，共20趟，总共需要420分钟。显卡第一趟也需要21分钟，且只能挑10公斤粮食，但第一趟在刚装完桶时，第二个小人就开始装桶了，在第一个小人送完时最后一个小人也装完了桶，接着第一个小人又可以装桶了，所以从共需要2000/10=200个轮回，但每个轮回只需要1分钟，加上最后的小人需要20分钟送完，总计200×1+20=220分钟，比CPU快近一倍的时间。当运算需求越大时，GPU的优势越明显。

6.3 有关CUDA中核函数

Function<<<griddim,blockdim,extern shared memory,GPU stream>>>(param...);

griddim表示调用的block块数  
blockdim表示调用的thread数

kernal函数用\_\_global\_\_修饰符来修饰

其中threadIdx和blockDim是CUDA的内建变量，指示这当前正在调度的线程号和线程的数量。  
每个线程都会调用一次  
后面两个参数分别表示动态定义共享内存大小和可使用的SM处理器数。

## 参考资料：

[1]: <https://docs.nvidia.com/#nvidia-cuda-toolkit>

[2]: https://blog.csdn.net/cyhbrilliant/article/details/79434090