Parallel-Programming Task11

刘森元, 21307289

中山大学计算机学院

Codes on https://github.com/Myocardial-infarction-Jerry/Parallel-Programming/tree/main/Task11.

Project built by CMake.

```
> cd Task11
> ./gen.sh
> build/bin/Convolution res/image.jpg res/kernel1.dat
> build/bin/Convolution res/image.jpg res/kernel1.dat res/kernel2.dat res/kernel3.dat
> build/bin/ConvolutionCuDNN res/image.jpg res/kernel1.dat res/kernel2.dat res/kernel3.dat

# Generating new kernel
> build/bin/KernelGenerate <height> <width> <channels> # New kernels will be in res/
```

1 Environment

11th Gen Intel(R) Core(TM) i7-11700KF @ 3.60GHz

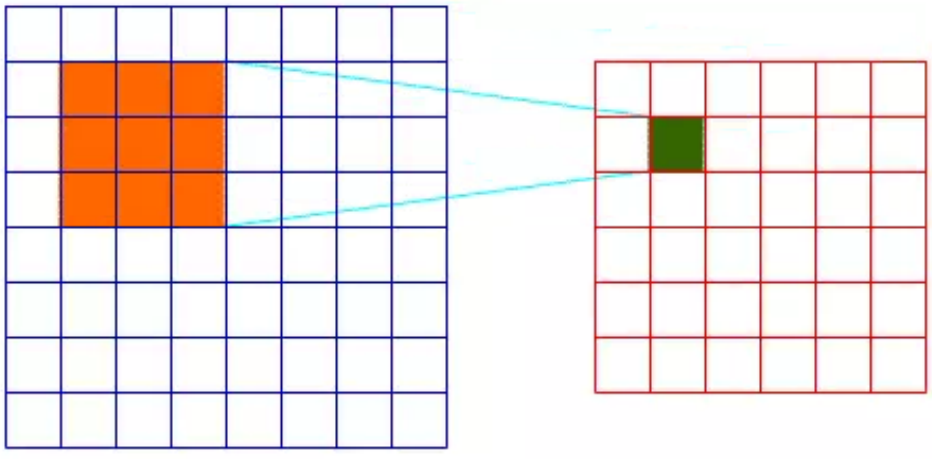
NVIDIA GeForce RTX 3080 Ti O12G

Windows Subsystem for Linux @ Ubuntu 22.04 LTS

2 Tasks

2.1 Task 1

> **Note**
>
> 在信号处理、图像处理和其他工程/科学领域，卷积是一种使用广泛的技术。在深度学习领域，卷积神经网络(CNN)这种模型架构就得名于这种技术。在本实验中，我们将在GPU上实现卷积操作，注意这里的卷积是指神经网络中的卷积操作，与信号处理领域中的卷积操作不同，它不需要对Filter进行翻转，不考虑bias。



> 任务一通过CUDA实现直接卷积（滑窗法），输入从256*256*增加至*4096*4096或者输入从32*32*增加至*512*512.
>
> 输入：Input matrix size和Kernel size, 例如 32 和 3
>
> 问题描述：用直接卷积的方式对Input进行卷积，这里只需要实现2D, height*width*，通道*channel(depth)*设置为*3*，*Kernel (Filter)*大小设置为*3*3，kernel channel(depth)设置为3，步幅(stride)分别设置为1，2，3，可能需要通过填充(padding)配合步幅(stride)完成CNN操作。注：实验的卷积操作不需要考虑bias(b)，bias设置为0.
>
> 输出：输出卷积结果以及计算时间

2.2 Task 2

> **Note**
>
> 任务二使用im2col方法结合上次实验实现的GEMM实现卷积操作。输入从256*256增加至4096*4096或者输入从32*32增加至512*512，具体实现的过程可以参考下面的图片和参考资料。
>
> 输入：Input matrix size和Kernel size, 例如 32 和 3
>
> 问题描述：用直接卷积的方式对Input进行卷积，这里只需要实现2D, height*width，通道channel(depth)设置为3，Kernel (Filter)大小设置为3*3，kernel channel(depth)设置为3，步幅(stride)分别设置为1，2，3，可能需要通过填充(padding)配合步幅(stride)完成CNN操作。注：实验的卷积操作不需要考虑bias(b)，bias设置为0.
>
> 输出：卷积结果和时间。

2.3 Task 3

> **Note**
>
> NVIDIA cuDNN是用于深度神经网络的GPU加速库。它强调性能、易用性和低内存开销。
>
> 使用cuDNN提供的卷积方法进行卷积操作，记录其相应Input的卷积时间，与自己实现的卷积操作进行比较。如果性能不如cuDNN，用文字描述可能的改进方法。

3 Theory

`img2col` 是一个常见的图像处理技术，特别是在实现卷积神经网络（CNN）时。这个方法将涉及到的图像卷积操作转化为矩阵乘法，以便更有效地使用现代计算架构（特别是 GPU）进行大规模并行处理。

3.1 图像卷积

图像卷积是一种用特定的卷积核（或滤波器）对图像进行滤波的操作，以提取某些特征，如边缘、纹理等。在 CNN 中，卷积核的参数是学习得来的，用于从图像数据中自动学习和提取有用的特征。

3.2 img2col 方法

`img2col` 方法将涉及卷积操作的图像区域重新排列成列。这使得多个卷积核可以通过一个矩阵乘法操作应用到图像上。具体来说，对于每个卷积窗口，`img2col` 将其展开为一列向量。这样，所有的窗口展开后形成的大矩阵可以与一个包含所有卷积核展开向量的权重矩阵相乘，从而一次完成所有的卷积操作。

3.3 并行处理与 GPU 加速

在 GPU 上使用 `img2col` 加速卷积的过程是高度并行的。GPU 通过其大量的小处理单元（cores）同时处理多个数据，使得图像卷积这种本质上可以并行的操作得到显著的速度提升。`img2col` 通过将图像块重新排列成独立的列，使得每个线程可以独立地计算输出特征图的一个元素。

3.4 内存管理

在 CUDA 编程中，管理内存传输（如从主内存到 GPU 内存）是性能优化的关键部分。有效地利用内存带宽和减少内存传输是优化 GPU 程序的重要方面。

3.5 性能优化

在实际的 GPU 实现中，还需要考虑诸如内存访问模式、线程分配策略和使用共享内存等因素，这些都可以显著影响程序的执行速度。

4 Code

> **Caution**
>
> 源代码详见 img2col.cu

```cpp
// CUDA kernel for img2col operation
__global__ void img2col(float *input, int inputHeight, int inputWidth, int inputChannels, float *output, int outputHeight, int outputWidth, int outputChannels, float *kernels, int kernelHeight, int kernelWidth, int stride) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int k = blockIdx.z;

    if (i < outputHeight && j < outputWidth) {
        for (int l = 0; l < kernelHeight; ++l)
            for (int m = 0; m < kernelWidth; ++m)
                for (int n = 0; n < inputChannels; ++n)
                    output[(i * outputWidth + j) * outputChannels + k] += input[(i * stride + l) * inputWidth * inputChannels + (j * stride + m) * inputChannels + n] * kernels[k * kernelHeight * kernelWidth * inputChannels + l * kernelWidth * inputChannels + m * inputChannels + n];
    }
}
```

5 Result

以 640x640 图像 image.jpg 为例

```
~/Parallel-Programming/Task11 main
build/bin/Convolution res/image.jpg res/kernel1.dat res/kernel2.dat res/kernel3.dat
Loaded image: res/image.jpg (640x640x3)
Loaded kernel: res/kernel1.dat (3x3x3)
Loaded kernel: res/kernel2.dat (3x3x3)
Loaded kernel: res/kernel3.dat (3x3x3)
[Img2Col] Input size:  642x642x3
[Img2Col] Output size: 640x640x3
[Img2Col] Kernel size: 3x3
[Img2Col] Stride: 1
[Img2Col] Memory allocating...
[Img2Col] Time: 0.889478ms
```

```
~/Parallel-Programming/Task11 main
build/bin/ConvolutionCuDNN res/image.jpg res/kernel1.dat res/kernel2.dat res/kernel3.dat
Loaded image: res/image.jpg (640x640x3)
Loaded kernel: res/kernel1.dat (3x3x3)
Loaded kernel: res/kernel2.dat (3x3x3)
Loaded kernel: res/kernel3.dat (3x3x3)
[cuDNN] Running...
[cuDNN] Time: 0.00637ms
```

5.1 原因分析

1. **硬件利用效率**: cuDNN 专门针对 NVIDIA GPU 的硬件架构进行了优化，包括对多种型号和配置的 GPU 的特定优化。您的代码可能没有充分利用 GPU 的所有能力，如流多处理器、共享内存和内存带宽。

2. **算法优化**: cuDNN 实现了多种卷积算法（如快速傅里叶变换（FFT）、Winograd算法等），并根据具体的输入大小、滤波器大小和硬件特性动态选择最优算法。相比之下，基于 `img2col` 的传统方法可能在算法效率上不足。

3. **内存访问模式**: 在 GPU 上，不合理的内存访问模式会导致严重的性能下降。如果内存访问不是连续的或存在大量的全局内存访问，这可能会限制整体性能。

4. **多线程和块配置**: 线程块的大小和形状对性能有重大影响。不合适的线程和块配置可能导致 GPU 计算资源的浪费。

5.2 改进方法

1. **使用更高效的内存访问策略**:
   - 利用共享内存减少全局内存的访问次数。共享内存是 GPU 上的一种快速缓存，可以显著减少对全局内存的访问延迟。
   - 优化内存访问模式，尽量使线程块内的内存访问连续和对齐。

2. **优化线程块的配置**:
   - 实验不同的线程块大小和形状，找到最适合您卷积操作的配置。
   - 确保 GPU 的多处理器能够尽可能地被均匀和充分地利用。

3. **尝试不同的卷积算法**:
   - 除了传统的直接卷积，可以考虑实现基于 FFT 或 Winograd 算法的卷积方法。这些方法可以在特定条件下提供更好的性能。

4. **使用动态算法选择**:
   - 类似 cuDNN 的方法，根据输入大小和滤波器配置动态选择最优的卷积算法。

5. **减少内核启动开销**:
   - 尽量减少 GPU 内核调用的次数，合并可并行的操作到一个内核中，以减少启动和同步的开销。

6. **利用 cuDNN 提供的API进行优化**:
   - 如果可能的话，可以直接在您的应用中集成 cuDNN 库，或者参考 cuDNN 的某些实现策略来优化您的代码。