作业 3: GPU 并行计算

龚舒凯 2022202790

1 数据集

作业用到的数据可从此链接下载：`https://pan.baidu.com/s/11e3F7shT2WP_Va-2Bb-KJw?pwd=ejf4`，提取码为 ejf4

2 任务一: 编写 GPU 内核函数实现图像拼接

在“基于卫星图像评估地区发展水平”这一大作业中，如图 1所示，为了得到一个地区**细粒度的发展水平**，我们进行如下操作

1. 将这一地区的所有区块的卫星图像爬取下来，每一个区块的图像大小为 10960×10960。
2. 将每个区块的图像分割为 32×32 个小块，每个小块的大小为 343×343。
3. 对每一个小块的图像进行经济发展水平的打分，得到每一个小块的得分，将得分储存在一个 `csv` 文件中。
4. 对于每个小块，用这一小块的得分值创建一个 343×343 的矩阵。
5. 将所有 32×32 个小块的矩阵拼接起来，得到一个 10960×10960 的矩阵，即为整个区块的发展水平。

图 1: 卫星图像的分割、打分与得分拼接

这一图像拼接的任务非常适合使用 GPU 进行并行计算。**本作业使用 PyCuda 中的 SourceModule 编写 GPU 内核函数，设计如下：**

- 线程：每个线程负责将一个小块的得分填充到对应的 343×343 的矩阵中。
- 线程块：每个线程块负责拼接一张大图像，包含 32×32 个小块。
- 网格：每个网格中含有 n 个线程块，负责拼接出 n 张 10960×10960 的大图像。

这样一来，程序就可以并行的处理 n 张区块图像的拼接，且每张区块图像内部的拼接也是并行的。具体的 GPU 内核函数的实现如下 (只展示内核函数的代码):

```
kernel_code = """
__global__ void score_mapping_kernel(float *scores, float *output, int width, int height,
int num_blocks_per_row, int num_images) {
```

```
        int image_idx = blockIdx.x;  // current image index
        int block_idx = threadIdx.x + threadIdx.y * blockDim.x;  // current thread index in the block

        int row = block_idx / num_blocks_per_row;  // the row index of the chunk
        int col = block_idx % num_blocks_per_row;  // the column index of the chunk

        if (row < num_blocks_per_row && col < num_blocks_per_row) {
                // find the starting position of the score matrix for the current granule
                float score = scores[image_idx * num_blocks_per_row * num_blocks_per_row +
                row * num_blocks_per_row + col];
                int start_row = row * 343;
                int start_col = col * 343;

                // compute the shift of the current image in the output array
                int image_offset = image_idx * width * height;

                // fill the 343x343 region of each chunk
                for (int i = 0; i < 343; i++) {
                        for (int j = 0; j < 343; j++) {
                                int index = image_offset + (start_row + i) * width + (start_col + j);
                                output[index] = score;
                        }
                }
                __syncthreads();
        }
}
"""
mod = SourceModule(kernel_code) # compile the CUDA kernel
score_mapping_kernel = mod.get_function("score_mapping_kernel") # get the kernel function
```

拼接 x 张图像共用时 y。拼接的效果如下所示：

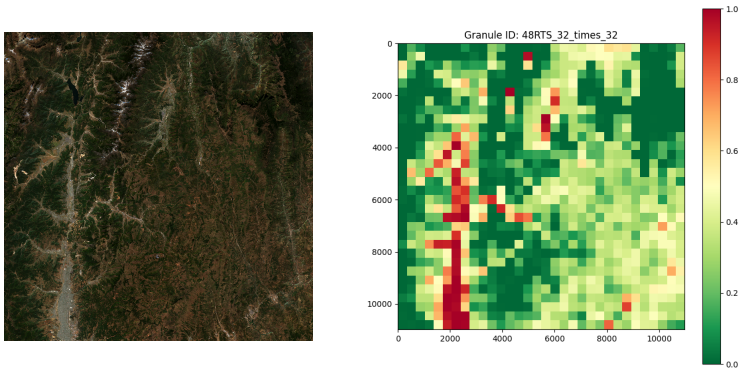

图 2: 左图为 48RTS 区块的原始卫星图像，右图为 48RTS 区块的发展水平热力图

3 任务二: 基于 cuBLAS 库的卫星图像边缘检测

卷积是图像处理中的基本操作之一，对卫星图像进行卷积操作可以提取出图像中的特征，如建筑物边缘、水体纹路等信息。二维图像的卷积特征图通过一个卷积核 (filter) 在输入图像上滑动，计算局部区域与卷积核的加权和得到。给定输入图像矩阵 $\mathbf{I} \in \mathbb{R}^{C\times H\times W}$ 和卷积核 $\mathbf{K} \in \mathbb{R}^{C\times k_h\times k_w}$，输出的卷积图像 $\mathbf{O}$ 的计算公式如下:

$$\mathbf{O}(c,h,w) = \sum_{i=0}^{k_h-1}\sum_{j=0}^{k_w-1}\sum_{k=0}^{C-1}\mathbf{I}(k,h+i,w+j)\mathbf{K}(k,i,j)$$

直接实现卷积操作需要嵌套的循环，计算复杂度高，在大规模图像上效率低下。介于 GPU 的高效矩阵运算能力，可以使用 `im2col` 算法将卷积操作转换为矩阵乘法。具体步骤如下:

1. 对于输入图像矩阵 $\mathbf{I}$，将每个卷积核的滑动到的局部区域拉伸为一列 (不妨设 padding=0, stride=1)，可得到矩阵

   $$\mathbf{I}_{col} \in \mathbb{R}^{(C\times k_h\times k_w)\times(\text{out}_h\times\text{out}_w)}$$

   其中 $\text{out}_h = H - k_h + 1, \text{out}_w = W - k_w + 1$.

2. 将卷积核 $\mathbf{K} \in \mathbb{R}^{C\times k_h\times k_w}$ 展平为一行，得到矩阵

   $$\mathbf{K}_{\text{flat}} \in \mathbb{R}^{1\times(C\times k_h\times k_w)}$$

3. 将展平的卷积核 $\mathbf{K}_{\text{flat}}$ 与拉伸的输入图像矩阵 $\mathbf{I}_{col}$ 相乘，得到输出矩阵

   $$\mathbf{O}_{col} = \mathbf{K}_{\text{flat}}\mathbf{I}_{col} \in \mathbb{R}^{1\times(\text{out}_h\times\text{out}_w)}$$

4. 将输出矩阵重塑为输出图像矩阵

   $$\mathbf{O} = \mathbf{O}.\text{reshape}(C, \text{out}_h, \text{out}_w) \in \mathbb{R}^{C\times\text{out}_h\times\text{out}_w}$$

在本任务下，我们选取一张 10960×10960 **的 TIFF 格式卫星图像，通过使用 Sobel 核对其卷积的方式实现该卫星图像的边缘检测**。Sobel 算子的定义如下:

$$\mathbf{K}_x = \begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix},\quad \mathbf{K}_y = \begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ 1 & 2 & 1\end{bmatrix}$$

$\mathbf{K}_x$ 和 $\mathbf{K}_y$ 分别为 Sobel 算子的水平和垂直方向的卷积核，对输入图像进行卷积操作后，可以得到水平和垂直方向的边缘图像。整体的算法的具体实现如下

1. 读取 TIFF 格式卫星图像，将其转换为 numpy 数组.
2. 使用 `scikit-cuda` 库实现 im2col 算法。对卫星图像的通道 i，将图像矩阵转换为矩阵 $\mathbf{I}_{i,col}$，分别与展平后的 $\mathbf{K}_x$ 和 $\mathbf{K}_y$，得到两个特征图 $\mathbf{G}^x_{i,col}, \mathbf{G}^y_{i,col}$.
3. 计算两个特征图的梯度幅值

   $$\mathbf{G}_{i,col} = \sqrt{\mathbf{G}^x_{i,col}\odot\mathbf{G}^x_{i,col} + \mathbf{G}^y_{i,col}\odot\mathbf{G}^y_{i,col}}$$

4. 对三个通道的梯度幅值图像求平均得到最终的特征图 $\mathbf{O}_{col} = \frac{1}{3}\sum_{i=1}^{3}\mathbf{G}_{i,col}$，得到最终的边缘检测图像。

矩阵乘法通过使用 cuBLAS 库中的 `cublasSgemm` 函数实现。具体的实现代码如下:

```python
def perform_convolution(cublas_handle, kernel_gpu, cols_gpu, m, n, k):
        """
        Conduct convolution operation using cuBLAS.
        input:
           cublas_handle: cublas.cublasHandle_t. The cuBLAS handle.
           kernel_gpu: gpuarray.GPUArray. The kernel (i.e. filter) matrix, shape=(m, k).
           cols_gpu: gpuarray.GPUArray. The column (i.e. the columnized image) matrix, shape=(k, n).
           m: int. The number of rows in kernel matrix.
           n: int. The number of columns in column matrix.
           k: int. The number of columns in kernel matrix.
        output:
           C_gpu: gpuarray.GPUArray. The output matrix, shape=(m, n).
        """
        alpha = 1.0
        beta = 0.0
        C_gpu = gpuarray.zeros((m, n), np.float32)

        status = cublas.cublasSgemm(cublas_handle, cublas._CUBLAS_OP['N'], cublas._CUBLAS_OP['N'],
                                    m, n, k, alpha,
                                    kernel_gpu.gpudata, m,
                                    cols_gpu.gpudata, k,
                                    beta, C_gpu.gpudata, m)

        return C_gpu
```

对于 10960×10960 的超大卫星图像，边缘检测共用时 17.56 秒，效果如图 **??**所示:

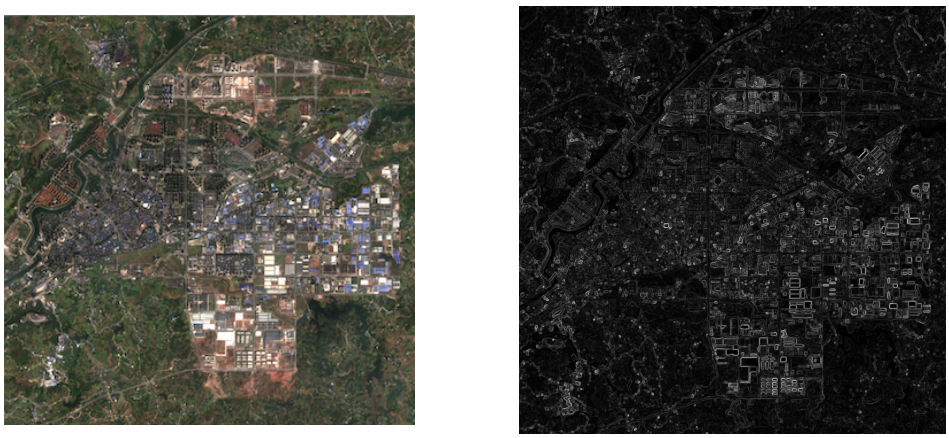

图 3: 使用矩阵乘法实现超大卫星图像的边缘检测：以 48RWT 区块为例

4 任务三: 基于 cuFFt 库的卫星图像边缘检测

在任务二中，我们通过将卷积操作转化为矩阵乘法的方式实现了卫星图像的边缘检测。然而，对于大规模的卫星图像，矩阵乘法的计算复杂度仍然较高，效率不高。在这种情况下，可以使用快速傅立叶变换 (FFT) 来实现卷积操作。

我们首先指出卷积和傅立叶变换之间的关系：不妨设有两时域信号 $f(t)$ 和 $g(t)$，则它们的卷积为

$$(f*g)(t)=\int_{-\infty}^{\infty}f(\tau)g(t-\tau)d\tau$$

傅立叶变换和其逆变换分别定义为

$$F(\omega)=\int_{-\infty}^{\infty}f(t)e^{-j\omega t}dt,\quad f(t)=\frac{1}{2\pi}\int_{-\infty}^{\infty}F(\omega)e^{j\omega t}d\omega$$

注意到 $(f*g)$ 的傅立叶变换为

$$\begin{aligned}\mathcal{F}\{(f*g)(t)\}&=\int_{-\infty}^{\infty}\left[\int_{-\infty}^{\infty}f(\tau)g(t-\tau)d\tau\right]e^{-j\omega t}dt=\int_{-\infty}^{\infty}f(\tau)\left[\int_{-\infty}^{\infty}g(t-\tau)e^{-j\omega t}dt\right]d\tau\\&=\int_{-\infty}^{\infty}f(\tau)e^{-j\omega\tau}d\tau\int_{-\infty}^{\infty}g(t-\tau)e^{-j\omega(t-\tau)}d(t-\tau)\\&=F(\omega)G(\omega)\\\Rightarrow(f*g)(t)&=\mathcal{F}^{-1}\{F(\omega)G(\omega)\}\end{aligned}$$

也就是说，想求时域信号 $f(t)$ 和 $g(t)$ 的卷积，可以先分别对 $f(t)$ 和 $g(t)$ 进行傅立叶变换得到频域信号 $F(\omega)$ 和 $G(\omega)$，然后将两个频域信号的傅立叶变换逐元素相乘，最后对乘积进行逆变换，即可得到卷积的结果。这一结论对于离散卷积 (图像卷积是二维的离散卷积) 同样成立，基于此，我们设计如下算法，利用傅立叶变换实现 Sobel 算子卷积，从而实现卫星图像的边缘检测:

1. 读取 TIFF 格式卫星图像，将其转换为 numpy 数组 $\mathbf{I}\in\mathbb{R}^{C\times M\times M}$。
2. 对于 $\mathbf{I}$ 的每个通道 $\mathbf{I}_c\in\mathbb{R}^{M\times M}$，以及 Sobel 核 $\mathbf{K}_x,\mathbf{K}_y\in\mathbb{R}^{3\times3}$，先对其进行填充 (padding): 令填充后图像和卷积核的大小均为 $(M+3-1)\times(M+3-1)$。即
$$\mathbf{I}_c^{\text{pad}}=\begin{bmatrix}\mathbf{I}_c&\mathbf{0}_{M\times2}\\\mathbf{0}_{2\times M}&\mathbf{0}_{2\times2}\end{bmatrix},\quad\mathbf{K}_x^{\text{pad}}=\begin{bmatrix}\mathbf{K}_x&\mathbf{0}_{3\times(M-1)}\\\mathbf{0}_{(M-1)\times3}&\mathbf{0}_{(M-1)\times(M-1)}\end{bmatrix},\quad\mathbf{K}_y^{\text{pad}}=\begin{bmatrix}\mathbf{K}_y&\mathbf{0}_{3\times(M-1)}\\\mathbf{0}_{(M-1)\times3}&\mathbf{0}_{(M-1)\times(M-1)}\end{bmatrix}$$
3. 对填 $\mathbf{I}_c^{\text{pad}}$，$\mathbf{K}_x^{\text{pad}},\mathbf{K}_y^{\text{pad}}$ 进行二维傅立叶变换，得到频域信号 $\mathbf{I}_c^{\text{FFT}},\mathbf{K}_x^{\text{FFT}},\mathbf{K}_y^{\text{FFT}}$。
4. **先对 Sobel 核的频域信号求共轭 (因为图像卷积对应的是数字信号处理中的自相关)，再与图像的频域信号逐元素相乘，对结果进行逆傅立叶变换，最后取实部得到卷积结果**:
$$\mathbf{G}_c^x=\text{real part}\left\{\mathcal{F}^{-1}\left\{\mathbf{I}_c^{\text{FFT}}\odot\overline{\mathbf{K}_x^{\text{FFT}}}\right\}\right\},\quad\mathbf{G}_c^y=\text{real part}\left\{\mathcal{F}^{-1}\left\{\mathbf{I}_c^{\text{FFT}}\odot\overline{\mathbf{K}_y^{\text{FFT}}}\right\}\right\}$$
5. 计算两个特征图的梯度幅值 $\mathbf{G}_c=\sqrt{\mathbf{G}_c^x\odot\mathbf{G}_c^x+\mathbf{G}_c^y\odot\mathbf{G}_c^y}$。对三个通道的梯度幅值图像求平均得到最终的边缘检测图像 $\mathbf{O}=\frac{1}{3}\sum_{i=1}^{3}\mathbf{G}_c$.

傅立叶变换和逆变换通过使用 scikit-CUDA 中的 fft 和 linalg 库实现。具体的实现代码如下：

```python
def cufft_conv(x, y):
        """
        Convolution operation two padded arrays with the same shape using FFT.
        Input:
                x: np.ndarray. Padded Sobel kernel in this case.
                y: np.ndarray. Padded image matrix in this case.
        Output:
                conv_real: np.ndarray. The convoluted image matrix.
        """

        # FFT requires the input arrays to be complex numbers
        x = x.astype(np.complex64)
        y = y.astype(np.complex64)

        if x.shape != y.shape:
                print("Error: Incompatible shape of padded kernel and padded image.")
                return -1

        # FFT plans
        plan_forward = Plan(shape=x.shape, in_dtype=np.complex64, out_dtype=np.complex64)
        plan_inverse = Plan(shape=x.shape, in_dtype=np.complex64, out_dtype=np.complex64)

        # Copy the padded kernel and padded image to GPU
        x_gpu = gpuarray.to_gpu(x)
        y_gpu = gpuarray.to_gpu(y)
        # Allocate GPU memory for the kernel and image after FFT
        x_fft = gpuarray.empty_like(x_gpu)
        y_fft = gpuarray.empty_like(y_gpu)

        # Forward FFT
        fft.fft(x_gpu, x_fft, plan_forward)
        fft.fft(y_gpu, y_fft, plan_forward)

        x_fft_conj = linalg.conj(x_fft)
        linalg.multiply(x_fft_conj, y_fft, overwrite=True)
        out_gpu = gpuarray.empty_like(y_fft)
        fft.ifft(y_fft, out_gpu, plan_inverse)

        conv_out = out_gpu.get()
        conv_real = np.real(conv_out)

        return conv_real
```

对于 10960×10960 的超大卫星图像，边缘检测共用时 13.48 秒，效果如图 **??**所示: 可以看到与图 **??**的效果是完全相同的。

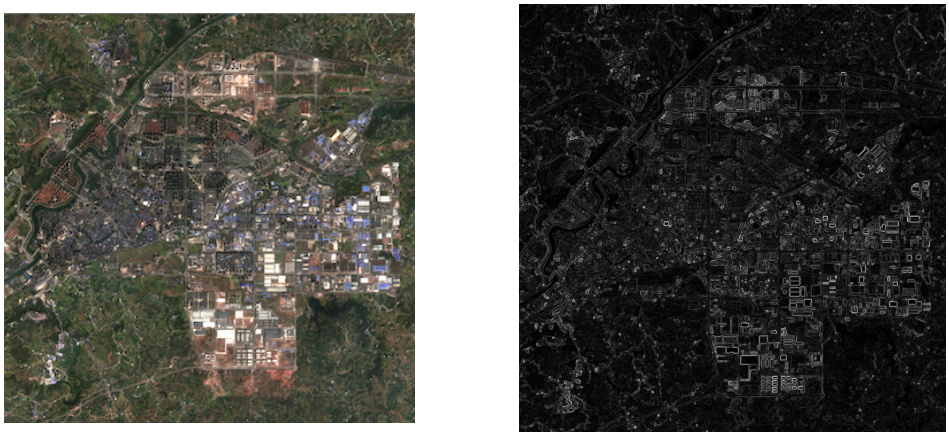

图 4: 使用矩阵乘法实现超大卫星图像的边缘检测：以 48RWT 区块为例

5 任务四: 基于 cuSolver 库对卫星图像进行主成分分析

如果想要知道一张卫星图像中的主要特征 (如城市群、河流、山峦)，可以通过对图像进行主成分分析 (PCA) 来实现。具体而言，我们对图像矩阵 $\mathbf{I} \in \mathbb{R}^{M \times N}$ 进行奇异值分解 (SVD)

$$\mathbf{I} = \mathbf{USV}^T$$

其中 $\mathbf{U} \in \mathbb{R}^{M \times M}$ 和 $\mathbf{V} \in \mathbb{R}^{N \times N}$ 是正交矩阵，$\mathbf{S} \in \mathbb{R}^{M \times N}$ 是对角矩阵，对角元素称为奇异值，衡量了图像中的主要特征。因此，如果想要获得保留前 k 个奇异值的主成分图像，只需要取 $\mathbf{U}$ 的前 k 列，$\mathbf{S}$ 的前 k 个对角元素，$\mathbf{V}$ 的前 k 行，即可得到主成分图像，即

$$\mathbf{I}_k = \mathbf{U}_k \mathbf{S}_k \mathbf{V}_k^T$$

SVD 分解涉及到大规模矩阵的运算，可以使用 GPU 进行显著的加速。因此，在本任务中，我们选取一张 10960×10960 **的 TIFF 格式卫星图像，通过使用 scikit-CUDA 库中的 linalg 作 SVD 分解 (skcuda.linalg 封装了 cuSolver 库的功能，可以调用了 cuSolver 来执行 GPU 上的 SVD 分解)，实现** $k = 1, 5, 10$ **的主成分分析**。效果如图 5所示：原始图像的主要特征为枝干状的农田群，而这一特征也反映在主成分图像中。随着 k 的增大，主成分图像的细节越来越丰富，容纳了如山峦等其他特征的信息。

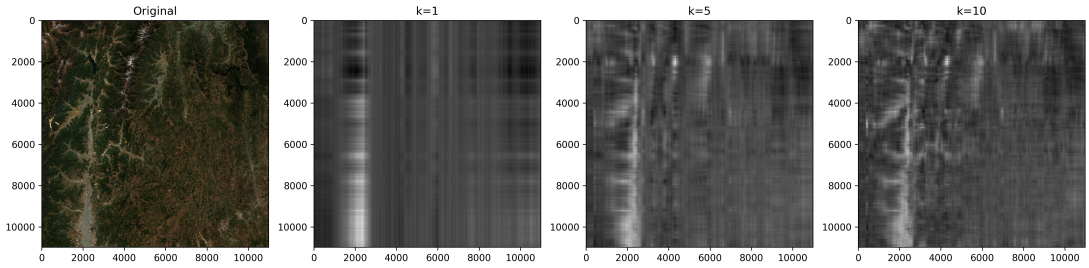

图 5: 对 48RTS 区块的主成分分析

对图像的 SVD 分解共用时 86.19 秒，组装成 $k = 1, 5, 10$ 的主成分图像分别用时 0.26, 0.28, 0.29 秒。

6 任务五: 基于 CUDA MATH API 实现二元函数定积分计算

CUDA Math API 是一组在 CUDA 设备端 (即 GPU) 运行的数学函数库，它提供了一个方便的字符串模版`%(math_function)s`，可以传入数学表达式。**在本任务中，我们通过头文件 `<curand_kernel.h>` 提供的随机数生成器实现了一个基于蒙特卡洛算法的二元函数的定积分计算。**

具体而言，对于一个二元函数 $f(x,y)$，我们希望计算其在区域 $[a,b]\times[c,d]$ 上的定积分。蒙特卡洛算法的基本思想是，我们在区域 $[a,b]\times[c,d]$ 上随机生成 N 个点 (x_i,y_i)，然后计算 $f(x_i,y_i)$ 的平均值，乘以区域 $[a,b]\times[c,d]$ 的面积，即可得到定积分的近似值，即

$$\int_a^b\int_c^d f(x,y)dxdy \approx \frac{(b-a)(d-c)}{N}\sum_{i=1}^{N} f(x_i,y_i)$$

基于此, 我们可以将利用好 GPU 在并行计算方面的优势, 将区域 $[a,b]\times[c,d]$ 划分为 N 个小区域 $[a_i,b_i]\times[c_i,d_i]$, 每个小区域上随机生成一个点 (x_i,y_i)，然后计算 $f(x_i,y_i)$ 的平均值，最后将所有小区域上的平均值相加，乘以小区域的面积，即可得到定积分的近似值。具体而言，**本任务使用 PyCuda 中的 SourceModule 编写 GPU 内核函数，设计如下：**

1. 每个线程负责一个小区间 $[t_{\text{low}_{x_i}},t_{\text{high}_{x_i}}]\times[t_{\text{low}_{y_i}},t_{\text{high}_{y_i}}]$ 上的随机点的生成和函数值的计算。
2. 每个线程块 (这里采取 1D 线程块) 包括 256 个线程，可以处理 $\bigcup_{i=1}^{256}[t_{\text{low}_{x_i}},t_{\text{high}_{x_i}}]\times[t_{\text{low}_{y_i}},t_{\text{high}_{y_i}}]$ 区域上的随机点的生成和函数值的计算。
3. 网格 (这里采取 1D 网格) 中共有 256 个线程块，处理完所有的小区域，得到定积分的近似值。

具体的 GPU 内核函数的实现如下 (只展示内核函数的代码):

```
MonteCarlo2DKernel = """
        #include <curand_kernel.h>

        typedef unsigned long long ULL;
        #define _R(z)   ( 1.0 / (z) )
        #define _P2(z)   ( (z) * (z) )

        // p stands for "precision" (float or double)
        __device__ inline %(p)s f(%(p)s x, %(p)s y){
                %(p)s result;
                %(math_function)s;
                return result;
        }

        extern "C" {
                __global__ void monte_carlo_2d(int iters, %(p)s lo_x, %(p)s hi_x,
                %(p)s lo_y, %(p)s hi_y, %(p)s * ys_out){
                        // Initialize the random number generator
                        curandState_t state;
```

```
                int tid = blockIdx.x * blockDim.x + threadIdx.x;
                int num_threads = blockDim.x * gridDim.x;

                curand_init((ULL)clock64() + tid, 0, 0, &state);

                %(p)s sum = 0.0;

                for(int i = 0; i < iters; i++){
                        // Initialize random number between [lo_x, hi_x]
                        %(p)s rand_x = lo_x + (hi_x - lo_x)
                                        * curand_uniform%(p_curand)s(&state);
                        // Initialize random number between [lo_y, hi_y]
                        %(p)s rand_y = lo_y + (hi_y - lo_y)
                                        * curand_uniform%(p_curand)s(&state);
                        sum += f(rand_x, rand_y);
                }

                ys_out[tid] = sum;
        }
    }
"""
```

本任务选取了三个测试样例，分别是

$$\iint_{[-1,1]\times[-1,1]} x^2+y^2 \mathrm{d}x\mathrm{d}y = \frac{8}{3}$$
$$\iint_{[0,\pi]\times[0,\pi]} \sin x \cos y \mathrm{d}x\mathrm{d}y = 0$$
$$\iint_{[-2,2]\times[-2,2]} e^{-x^2+y^2} \mathrm{d}x\mathrm{d}y \approx 3.1123$$

代码运行效果如图 6所示，可以看到，GPU 计算的结果与理论值非常接近。

```
----------------------------------------
Function: f(x, y) = x*x+y*y
Integration Area: x ∈ [-1.0000, 1.0000], y ∈ [-1.0000, 1.0000]
Monte Carlo Integration: 2.6667
Ground Truth: 2.6667
Error: 0.0000

Computation Time: 0.1072 seconds
----------------------------------------
Function: f(x, y) = sin(x)*cos(y)
Integration Area: x ∈ [0.0000, 3.1416], y ∈ [0.0000, 3.1416]
Monte Carlo Integration: -0.0001
Ground Truth: 0.0000
Error: 0.0001

Computation Time: 0.4622 seconds
----------------------------------------
Function: f(x, y) = exp(-(x*x+y*y))
Integration Area: x ∈ [-2.0000, 2.0000], y ∈ [-2.0000, 2.0000]
Monte Carlo Integration: 3.1122
Ground Truth: 3.1123
Error: 0.0001

Computation Time: 0.2831 seconds _
```

图 6: 蒙特卡洛法计算二元函数定积分

运行配置

作业涉及到的主要工具:

- 任务 1: `pycuda.compiler.SourceModule`
- 任务 2: `opencv-python`, `pycuda.gpuarray`, `skcuda.cublas`
- 任务 3: `pycuda.gpuarray`, `skcuda.fft`, `skcuda.linalg`
- 任务 4: `pycuda.gpuarray`, `skcuda.linalg`
- 任务 5: `pycuda.compiler.SourceModule`, `pycuda.gpuarray`, `pycuda.driver`, `<curand_kernel.h>`

使用的服务器配置:

- GPU: NVIDIA GeForce RTX 4090
- CPU: 10 × Intel(R) Xeon(R) Platinum 8352V CPU @ 2.10GHz