6 GPU 编程

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆⋆ 了解 GPU 的基本特点和 SIMT 执行模型
- ⋆⋆⋆ 掌握 CUDA 编程模型中 `thread`、`block` 与 `grid` 的概念，能阅读简单的 CUDA 程序
- ⋆ 了解 NVIDIA GPU 的中间指令表示：PTX 指令集
- ⋆⋆⋆ 掌握在 PyTorch 中使用 GPU 进行计算的基本方法

6.1 了解 GPU

6.1.1 GPU 体系结构

现代图形处理器（Graphics Processing Unit, GPU）最初设计用于加速图形渲染，但因其高度并行的架构，逐渐成为通用并行计算（General-Purpose GPU, GPGPU）的重要平台。与 CPU 强调低延迟、复杂控制流和单线程性能不同，GPU 采用“吞吐量优先”的设计理念，通过集成数千个轻量级计算核心，同时处理大量相似的计算任务。

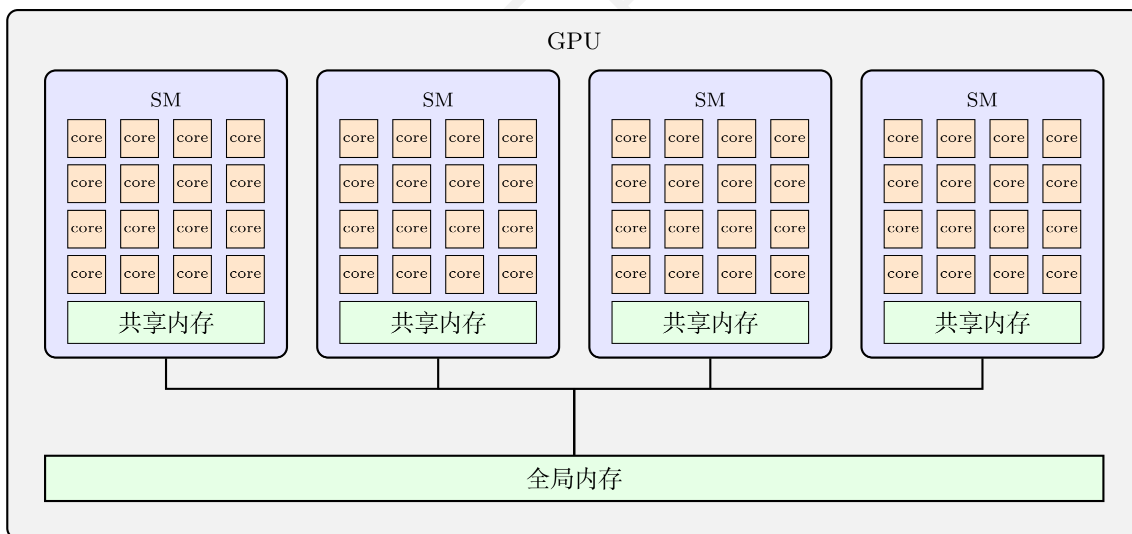

图 6.1: GPU 架构示意图

如图 6.1 所示，NVIDIA GPU 的硬件架构基于流式多处理器（Streaming Multiprocessor, SM）。一个 GPU 通常由多个 SM 组成。每个 SM 内部包含多个 CUDA 核心、共享内存以及线程调度器等硬件资源。CUDA（Compute Unified Device Architecture）是 NVIDIA 提供的 GPU 并行计算平台和编程模型，它允许开发者直接在 GPU 上编写通用计算程序。CUDA 利用 SM 的多核特性，将大量任务分解为多个线程，在硬件上实现大规模数据并行计算。每个 SM 可以同时执行多个线程，提高计算吞吐量。

GPU 的内存层次结构也与 CPU 显著不同，主要包括全局内存和共享内存。全局内存容量大（GB 级），但访问延迟高。共享内存位于 SM 内部，低延迟、高带宽，由线程块内所有线程共享。理解 GPU 的硬件

特性对于编写高效 CUDA 程序至关重要。程序员需合理组织线程结构、优化内存访问模式，并充分利用共享内存与计算资源，才能充分发挥 GPU 的并行计算能力。

6.1.2 SIMT 执行模型

SIMT（Single Instruction, Multiple Threads）是 GPU 的核心执行模型，也是 CUDA 的基本设计理念。在 SIMT 模型下，多个线程共享同一条指令流，但每个线程拥有独立的寄存器和局部状态，可以处理不同的数据。

在 CUDA 编程中，GPU 的并行性并不是通过显式创建线程实现的，而是通过一种分层的抽象执行模型来描述。CUDA 的执行层次结构可以总结为：

$$\textbf{Grid} \rightarrow \textbf{Block} \rightarrow \textbf{Warp} \rightarrow \textbf{Thread}$$

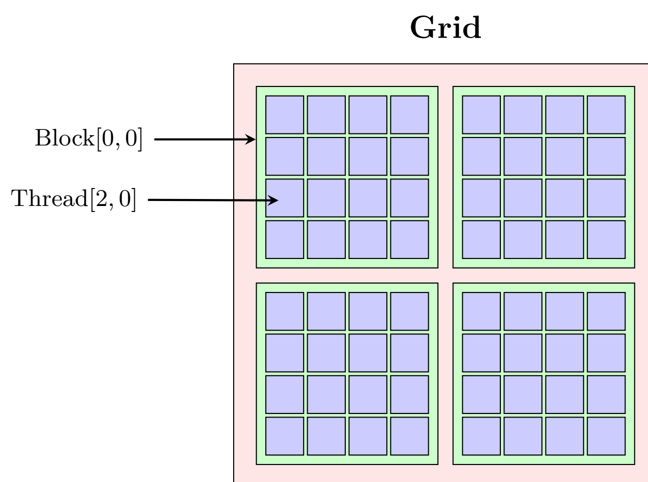

图 6.2: CUDA 层次结构示意图，每个 Grid 含有 2×2 Blocks，每个 Block 含 4×4 Threads

为了直观理解，假设我们在 GPU 上执行一个向量逐元素相乘任务。给定两个长度为 N 的向量 A 和 B，计算向量 C，其中

$$C_i = A_i \cdot B_i$$

- **Grid（网格）**：一次核函数启动所创建的所有 Block 的集合，用于覆盖整个向量计算任务。例如，当向量长度为 $N = 1024$ 时，Grid 需要包含足够多的 Block 来覆盖这 N 个元素。
- **Block（线程块）**：由多个线程组成的执行单元，负责处理向量的一段连续区间。例如，每个 Block 包含 256 个线程，则一个 Block 可以处理 256 个元素。若 $N = 1024$，则需要 $\frac{1024}{256} = 4$ 个 Block。
- **Warp（线程束）**：硬件调度单位，通常包含 32 个线程。一个包含 256 个线程的 Block 会被划分为 $\frac{256}{32} = 8$ 个 Warp。Warp 内的线程执行同一条乘法指令，但各自作用于不同的向量元素。
- **Thread（线程）**：CUDA 的最小执行单元。在逐元素相乘中，通常每个线程负责计算一个元素：$C_i = A_i \cdot B_i$ 由于各元素之间互不依赖，这种计算模式天然适合大规模并行。

CUDA 的 SM 与 Block 的映射并非一一对应。一个 Block 在执行期间被分配到某个 SM 上（Block 只驻留在一个 SM），但一个 SM 可以同时容纳多个 Block，只要硬件资源允许（如寄存器数量、共享内存容量、最大线程数等）。例如，如果一个 SM 支持最多 2048 个并发线程，而每个 Block 有 512 个线程，则该 SM 最多可同时执行 4 个 Block。这种多对一的调度机制可以通过大量线程隐藏内存访问延迟，提高 GPU 利用率。如果 Block 的线程数超过 SM 的资源限制，就无法完全调度整个 Block，需要调整 Block 大小。

6.2 CUDA C++ 编程

CUDA C++ 编程主要分为两步：编写核函数和调用核函数。

- **编写核函数**：核函数是在 GPU 上运行的函数，每个线程都会独立执行它。
- **调用核函数**：程序员从 CPU 端启动核函数，并指定 Grid 和 Block 的布局。调用核函数会在 GPU 上创建相应数量的线程，每个线程执行核函数中的代码。CPU 还负责准备输入数据、将数据从主机内存传输到 GPU 内存，以及在核函数执行完成后将结果传回主机。

6.2.1 编写核函数

代码 6.1 展示了如何使用 CUDA C++ 编写一个简单的核函数，实现向量的逐元素相乘。该核函数使用 `__global__` 关键字定义，表示该函数在 GPU 上执行，并由 CPU 端发起调用。

```
#include <cuda_runtime.h>

#define N 1024  // 向量长度

__global__ void ele_mul(float* a, float* b, float* c) {
    // 计算全局线程索引
    // blockIdx.x: 当前线程所在 Block 的索引
    // blockDim.x: 每个 Block 中线程的数量
    // threadIdx.x: 当前线程在 Block 内的索引
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // 边界检查：确保线程索引不超过向量长度
    if (i < N) {
        c[i] = a[i] * b[i];
    }
}
```

代码 6.1: 使用 CUDA C++ 编写向量逐元素相乘核函数

在该示例中，我们采用一维 Grid 和一维 Block 的组织方式，将向量计算任务映射到一维线程结构上。可以将整个 Grid 理解为对向量索引空间的划分：每个 Block 负责处理向量中的一段连续区间，而 Block 内的线程则分别对应区间中的具体元素。通过这种分层的索引计算方式，GPU 能够将向量逐元素相乘这一数据并行任务自然地映射到大量线程上，实现高效并行计算。每个线程通过表达式 `blockIdx.x * blockDim.x + threadIdx.x` 计算自己负责的全局向量索引。其中，`blockIdx.x` 表示当前 Block 在 Grid 中的编号，`blockDim.x` 表示每个 Block 中的线程数量，`threadIdx.x` 表示线程在 Block 内的局部编号。三者组合形成“块偏移量 + 块内偏移量”的全局索引计算方式。

由于 Grid 和 Block 的尺寸通常按需向上取整，可能会产生多余线程，因此代码中通过条件判断 `i < N` 进行边界检查，确保线程只访问合法的向量元素，从而避免越界访问。

CUDA 为不同层次的并行结构提供了内建索引变量，用于描述 Grid、Block 和 Thread 之间的层级关系。按照自顶向下的结构，可以理解为：

- `gridDim`：表示网格（grid）在三个维度上的尺寸，即整个核函数启动时包含的 block 数量。`gridDim.x`、`gridDim.y`、`gridDim.z` 分别对应 X、Y、Z 方向上的 block 数目。Grid 是一次核函数启动所创建的最大执行单元。
- `blockIdx`：表示当前 block 在整个 grid 中的三维索引，包含 x、y、z 分量，分别表示该 block 在网格 X、Y、Z 方向上的位置。通过 `blockIdx` 可以确定当前 block 负责的数据区域。

- `blockDim`：表示线程块（block）在三个维度上的尺寸，即每个 block 中包含的线程数量。`blockDim.x`、`blockDim.y`、`blockDim.z` 分别对应各维度的线程数量，用于描述 block 内部的线程组织方式。

- `threadIdx`：表示线程在其所属 block 中的三维索引，包含 `x`、`y`、`z` 分量，分别表示线程在 block 内 X、Y、Z 方向上的位置。结合 `blockIdx` 和 `blockDim`，可以计算出线程在整个数据空间中的全局位置。

代码 6.2 展示了一个二维矩阵乘法的例子。每个线程负责计算结果矩阵 C 中的一个元素 $C_{row,col}$。线程首先通过如下公式计算自己对应的矩阵坐标：

$$\texttt{row} = \texttt{blockIdx.y} \times \texttt{blockDim.y} + \texttt{threadIdx.y}$$

$$\texttt{col} = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x}$$

这里，行索引 `row` 和列索引 `col` 是分别独立计算的：`row` 仅依赖纵向的 `blockIdx.y` 与 `threadIdx.y`，而 `col` 仅依赖横向的 `blockIdx.x` 与 `threadIdx.x`。这种方式使每个方向的线程分布可以单独控制，从而灵活地组织二维 Grid 和二维 Block，实现对矩阵每个元素的精确映射。

在核函数内部，线程首先进行边界检查，确保索引不越界，然后通过循环计算对应的点积，并将结果写回全局内存。这种二维索引计算方式体现了 GPU 的二维线程组织结构，同时保证了每个线程独立处理矩阵中的一个元素。

```
#define N 1024   // 矩阵维度: N * N

__global__ void matmul(float* A, float* B, float* C) {
    // 计算当前线程负责的矩阵坐标
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // 边界检查: grid/block 的尺寸通常向上取整，因此可能会产生越界线程
    if (row < N && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < N; k++) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
```

代码 6.2: 使用 CUDA C++ 矩阵乘法核函数

6.2.2 调用核函数

主机端程序调用核函数包括完成以下几个步骤:

1) 在 CPU（Host）端分配并初始化数据;

2) 在 GPU（Device）端分配显存，并将数据从主机拷贝到设备;

3) 配置 Grid 和 Block 的尺寸，启动核函数;

4) 将计算结果从设备拷贝回主机，并释放相关资源。

代码 6.3以向量逐元素相乘为例说明核函数的完整调用流程。

```
int main() {
    size_t size = N * sizeof(float); // 计算向量所需内存

    //----------CPU端内存分配与初始化----------
    float *ha, *hb, *hc;
    ha = (float*)malloc(size); // 为向量 A 分配内存
    hb = (float*)malloc(size); // 为向量 B 分配内存
    hc = (float*)malloc(size); // 为向量 C 分配内存
    for (int i = 0; i < N; i++) {
        ha[i] = 2.0f;   // 初始化向量 A
        hb[i] = 3.0f;   // 初始化向量 B
    }

    //----------GPU端内存分配和数据拷贝----------
    float *da, *db, *dc;
    cudaMalloc(&da, size); // 为向量 A 分配内存
    cudaMalloc(&db, size); // 为向量 B 分配内存
    cudaMalloc(&dc, size); // 为向量 C 分配内存
    cudaMemcpy(da, ha, size, cudaMemcpyHostToDevice); // 将向量A从CPU拷贝到GPU
    cudaMemcpy(db, hb, size, cudaMemcpyHostToDevice); // 将向量B从CPU拷贝到GPU

    //----------配置并启动核函数----------
    int threadsPerBlock = 256; // 每个 Block 含 256 个线程
    int numBlocks = (N + threadsPerBlock - 1) / threadsPerBlock; // 计算所需 Block 数（向上取整）

    // 启动核函数，通过<<<gridDim, blockDim>>>设置核函数参数
    ele_mul<<<numBlocks, threadsPerBlock>>>(da, db, dc);

    //----------结果拷贝回主机，并释放资源----------
    cudaMemcpy(hc, dc, size, cudaMemcpyDeviceToHost);

    cudaFree(da);
    cudaFree(db);
    cudaFree(dc);
    free(ha);
    free(hb);
    free(hc);
    return 0;
}
```

代码 6.3: 调用 CUDA 向量乘法核函数

代码 6.8给出二维矩阵乘法的核函数调用方式。相比一维向量计算，矩阵计算采用二维 Grid 与二维 Block 结构，以便更自然地映射到矩阵的行列坐标。

```
int main() {
    // 省略内存分配与初始化步骤...

    // 每个 block 包含 16 * 16 = 256 个线程
    dim3 threadsPerBlock(16, 16);
    // ↑ dim3 是 CUDA 提供的三维结构 (x, y, z)
    //   这里只传了 (16,16), 等价于 (16,16,1)
    //   表示一个二维线程块 (block)

    // grid 尺寸向上取整，确保覆盖整个矩阵
    dim3 numBlocks(
        (N + threadsPerBlock.x - 1) / threadsPerBlock.x,
        (N + threadsPerBlock.y - 1) / threadsPerBlock.y
    );
    // ↑ numBlocks 也是 dim3 类型，表示 grid 的尺寸 (block 的数量)
    //   这里构造的是二维 grid，用于覆盖 N×N 的矩阵


    //----------启动 GPU 核函数----------
    matmul<<<numBlocks, threadsPerBlock>>>(dA, dB, dC);

    // 省略后续步骤...
}
```

代码 6.4: 调用矩阵乘法核函数

6.2.3 Tile 编程

在 GPU 编程中，访存性能往往成为限制程序性能的主要瓶颈。传统的矩阵乘法实现中，每一次乘加操作都需要从全局内存读取数据，而全局内存具有较高的访问延迟，导致计算单元无法被充分利用。为了解决这一问题，CUDA 中常采用 Tile（分块）优化技术，通过引入共享内存来提高数据复用率，从而显著提升性能。

Tile 编程的核心思想是将大规模矩阵划分为若干个小块，每个线程块（block）负责计算输出矩阵中的一个子块。在计算过程中，线程块中的所有线程协同工作，将输入数据的一部分数据加载到共享内存中。由于共享内存具有远低于全局内存的访问延迟，这些数据可以被线程块内的多个线程重复使用，从而减少对全局内存的访问次数。

代码 6.5展示了一种分块矩阵乘法的核函数实现，每个线程首先根据其在线程块和网格中的索引，确定其负责计算的输出矩阵元素 $C[row][col]$。随后，通过外层循环遍历所有 tile。在每一次迭代中，线程块加载矩阵 A 的一行子块和矩阵 B 的一列子块到共享内存数组 As 和 Bs 中。加载完成后，通过调用 `__syncthreads()` 进行线程同步，确保所有线程都完成数据加载后再进行后续计算。

在每个 tile 内部，线程通过一个长度为 TILE_SIZE 的循环，计算对应子块的部分乘积，即：

$$C[row][col] += \sum_{k=t\cdot TILE_SIZE}^{(t+1)\cdot TILE_SIZE-1} A[row][k]\cdot B[k][col]$$

经过所有 tile 的累加后，即可得到最终的矩阵乘法结果。最后，线程将计算结果写回全局内存。相比朴素实现，Tile 优化通常可以带来数量级的性能提升，是 GPU 矩阵计算中的经典优化方法之一。

```
__global__ void matmul_tiled(float* A, float* B, float* C, int N) {
    // 定义block共享内存，每个block加载 A 和 B 的一个 TILE（子块）
    __shared__ float As[TILE_SIZE][TILE_SIZE];  // A 的子块，TILE_SIZE 为常量
    __shared__ float Bs[TILE_SIZE][TILE_SIZE];  // B 的子块，TILE_SIZE 为常量

    // ----------计算当前线程负责的 C 中元素位置----------
    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;

    float value = 0.0f; // 用于保存C[row][col]的结果

    // ----------遍历所有 tile----------
    for (int t = 0; t < (N + TILE_SIZE - 1) / TILE_SIZE; t++) {
        // ----------加载 A 的一个 tile 到共享内存----------
        if (row < N && t * TILE_SIZE + threadIdx.x < N)
            As[threadIdx.y][threadIdx.x] = A[row * N + t * TILE_SIZE + threadIdx.x];
        else
            As[threadIdx.y][threadIdx.x] = 0.0f;

        // ----------加载 B 的一个 tile 到共享内存----------
        if (col < N && t * TILE_SIZE + threadIdx.y < N)
            Bs[threadIdx.y][threadIdx.x] = B[(t * TILE_SIZE + threadIdx.y) * N + col];
        else
            Bs[threadIdx.y][threadIdx.x] = 0.0f;

        __syncthreads(); // 同步线程，确保整个 tile 已经加载完成

        // ----------计算当前 tile 的部分乘积----------
        for (int k = 0; k < TILE_SIZE; k++) {
            value += As[threadIdx.y][k] * Bs[k][threadIdx.x];
        }

        __syncthreads(); // 再次同步，确保整个 tile 已经计算完成
    }

    // ----------写回结果到全局内存----------
    if (row < N && col < N) {
        C[row * N + col] = value;
    }
}
```

代码 6.5: 矩阵乘法性能实验

6.2.4 GPU 程序的编译过程

CUDA 程序通常使用 NVCC 编译器进行编译。首先，CUDA C/C++ 源代码（.cu 文件）会被编译为 PTX（Parallel Thread Execution）。PTX 是一种面向 NVIDIA GPU 的中间表示语言，类似于汇编语言，但具有一定的可移植性，可以视为 GPU 的“虚拟指令集”。随后，PTX 会进一步被编译为特定 GPU 架构对应的机器码（SASS）。如果目标 GPU 与编译时生成的机器码不兼容，则可以使用保留在二进制中的 PTX，在程序运行时由 GPU 驱动进行 JIT 编译，以生成适配当前 GPU 的机器码。

下面展示了向量逐元素相乘核函数经 NVCC 编译器生成的 PTX 代码。该核函数接收三个浮点数组指针作为参数，在边界检查后执行对应元素的乘法并写回结果。

```
        // ---- 核函数参数：三个指针 a, b, c ----
        ld.param.u64    %rd1, [_Z7ele_mulPfS_S__param_0]; // a 指针
        ld.param.u64    %rd2, [_Z7ele_mulPfS_S__param_1]; // b 指针
        ld.param.u64    %rd3, [_Z7ele_mulPfS_S__param_2]; // c 指针

        // ---- 线程层次索引 ----
        mov.u32         %r2, %ctaid.x;    // blockIdx.x
        mov.u32         %r3, %ntid.x;     // blockDim.x
        mov.u32         %r4, %tid.x;      // threadIdx.x

        // ---- 计算全局线程索引 tid = blockIdx.x * blockDim.x + threadIdx.x ----
        mad.lo.s32      %r1, %r2, %r3, %r4;

        // ---- 边界检查：if (tid > 1023) ----
        setp.gt.s32     %p1, %r1, 1023;
        @%p1 bra        $L__BB0_2;         // 越界线程直接返回

        // ---- 计算 a[tid] 的地址 ----
        cvta.to.global.u64  %rd4, %rd1;   // a 转为全局地址
        mul.wide.s32        %rd5, %r1, 4; // tid * sizeof(float)
        add.s64             %rd6, %rd4, %rd5;

        // ---- 计算 b[tid] 的地址 ----
        cvta.to.global.u64  %rd7, %rd2;
        add.s64             %rd8, %rd7, %rd5;

        // ---- 加载 a[tid], b[tid] ----
        ld.global.f32   %f1, [%rd8];      // b[tid]
        ld.global.f32   %f2, [%rd6];      // a[tid]

        // ---- 向量逐元素相乘 ----
        mul.f32         %f3, %f2, %f1;    // a[tid] * b[tid]

        // ---- 计算 c[tid] 的地址并写回 ----
        cvta.to.global.u64  %rd9, %rd3;
        add.s64             %rd10, %rd9, %rd5;
        st.global.f32       [%rd10], %f3; // c[tid] = result

$L__BB0_2:
        ret;                               // 结束
```

代码 6.6: 代码 6.1向量逐元素相乘核函数对应的 PTX 指令

6.3 实验与应用

6.3.1 实验环境：在 Kaggle 平台调试 GPU 程序

Kaggle[1] 是一个免费的云端数据科学和深度学习平台，提供预装的 PyTorch 等深度学习框架，以及可直接使用的 GPU/TPU 计算资源，非常适合进行 CUDA 程序的实验与调试。

Kaggle 的主要开发环境是 Notebook，其基本组成如下：

- **Notebook**：一个完整的工作环境，用于组织代码、文本以及实验结果。
- **Cell**：Notebook 的基本执行单元，每个 Cell 可以独立运行。根据功能不同，主要分为：
  - **Code Cell**：用于编写和执行代码或命令；
  - **Markdown Cell**：用于编写说明文档。

Kaggle 平台对 GPU 资源的使用存在一定的额度限制。在使用 GPU 之前，需要在 Notebook 设置中手动开启，将 Accelerator 设置为 GPU。Code Cell 中的常用命令如下：

```
# 验证 GPU 是否可用
!nvidia-smi

# Cell 1: 创建 CUDA 代码文件
%%writefile ele_mul.cu
# 此处填写程序源代码

# Cell 2: 编译程序
!nvcc ele_mul.cu -o ele_mul

# Cell 3: 运行程序
!./ele_mul

# Cell 4: 查看PTX代码
!nvcc -ptx ele_mul.cu -o ele_mul.ptx
```

代码 6.7: Code Cell 中编译和运行 CUDA 程序的常用命令

需要注意的是，`%%writefile` 与 `!nvcc` 等命令必须位于不同的 Cell 中，否则可能会导致错误。

AIE31008 人工智能的软件基础

[1]https://www.kaggle.com/

6.3.2 矩阵乘法性能实验

```
void matmul_cpu(float* A, float* B, float* C, int N) { // 普通CPU矩阵乘法 }
__global__ void matmul_naive(float* A, float* B, float* C, int N) { // 普通GPU矩阵乘法 }
template <int TILE_SIZE>
__global__ void matmul_tiled(float* A, float* B, float* C, int N) { // 分块矩阵乘法 }

int main() {
    // 省略若干代码
    // --------1: CPU 矩阵乘法测试--------
    double t1 = cpu_time();
    matmul_cpu(hA, hB, hC, N);
    double t2 = cpu_time();
    printf("CPU time: %f ms\n", t2 - t1);

    // --------2: GPU 矩阵乘法测试--------
    dim3 threads(16, 16);
    dim3 blocks((N + 15) / 16, (N + 15) / 16);
    cudaEventRecord(start);
    matmul_naive<<<blocks, threads>>>(dA, dB, dC, N);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float time_naive;
    cudaEventElapsedTime(&time_naive, start, stop);
    printf("GPU Naive: %f ms\n", time_naive);

     // --------3: GPU 分块矩阵乘法 (多种 TILE_SIZE 配置) 测试--------
#define RUN_TILE(T) { \
        dim3 tpb(T, T); \
        dim3 nb((N + T - 1) / T, (N + T - 1) / T); \
        cudaEventRecord(start); \
        matmul_tiled<T><<<nb, tpb>>>(dA, dB, dC, N); \
        cudaEventRecord(stop); \
        cudaEventSynchronize(stop); \
        float t_ms; \
        cudaEventElapsedTime(&t_ms, start, stop); \
        printf("GPU Tiled (%d): %f ms\n", T, t_ms); \
    }

    RUN_TILE(8);
    RUN_TILE(32);
    // 省略若干代码
}
```

代码 6.8: 矩阵乘法性能实验

实验结果参考如下：

```
CPU time: 3682.179199 ms
GPU Naive: 1.731328 ms
GPU Tiled (8): 1.726048 ms
GPU Tiled (32): 1.040832 ms
```

代码 6.9: 矩阵乘法实验结果

6.3.3 在 PyTorch 中使用 GPU

在 PyTorch 中，GPU 计算主要依赖于 CUDA 来加速张量运算和神经网络训练。通过将模型和数据移动到 GPU，可以显著提高训练速度。

上一章的 LeNet-5 训练代码经过修改以支持 GPU，如代码 6.10 所示。代码训练逻辑与 CPU 版本一致，关键改动包括以下三点：

- **检测 GPU 设备**：使用 `torch.device("cuda" if torch.cuda.is_available() else "cpu")` 自动选择可用的 GPU 或回退到 CPU。
- **模型移动到 GPU**：通过 `model.to(device)` 将模型参数移动到 GPU，以便后续计算在 GPU 上进行。
- **数据移动到 GPU**：在每个训练批次中，将输入数据 `inputs` 和标签 `labels` 移动到 GPU，确保前向和反向传播在 GPU 上执行。

```
# ---------------- 检测 GPU 设备 ----------------
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# ---------------- 实例化 LeNet-5 并移动到GPU ----------------
model = LeNet5().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

num_epochs = 5
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    for batch_idx, (inputs, labels) in enumerate(train_loader):
        # ---------------- 数据移动到GPU ----------------
        inputs = inputs.to(device)
        labels = labels.to(device)

        # 省略训练代码...

print("训练完成！")
torch.save(model.state_dict(), "lenet5_mnist.pth")  # 保存模型参数到文件
```

代码 6.10: 基于 GPU 训练 LeNet-5

人工智能的软件基础

练习

1) 分析矩阵乘法实验中不同分块大小对性能的影响并探索原因。

2) 将上节课的 RNN 代码示例修改为 GPU 版本。

AIE310008 人工智能的软件基础