并行计算

GPU 结构 (以 Nvidia GPU 为标准)

1. GPU 架构模型

- **抽象结构 (软件层面，程序员如何看待 GPU)**:
  - **Grid (网格)**：一次 CUDA Kernel 调用的**全部**工作量，可以是一维、二维或三维的。它由多个 Block 组成。
  - **CTA (Block, 块)**：Grid 的基本组成单位，是一组**可以互相协作**的线程集合。一个 Block 里的所有线程，可以访问**同一块共享内存 (Shared Memory)**，并且可以通过 `__syncthreads()` 进行同步。一个 Block 只能被调度到**一个 SM** 上执行。
  - **Warp (线程束)**：是 GPU **调度和执行的基本单位**，通常由 **32个** 线程组成。一个 Block 里的线程会被分成若干个 Warp。同一个 Warp 里的 32 个线程，在同一时刻**必须执行完全相同的指令**。
  - **Thread (线程)**：最基本的执行单元，每个线程有自己的**寄存器 (Register)** 和私有本地内存。
- **硬件结构 (GPU 芯片的物理构成)**:
  - **SM (Streaming Multiprocessor, 流式多处理器)**：GPU 的**核心计算单元**，可以看作是 GPU 上的一个“迷你 CPU 核心簇”。一块 GPU 由多个 SM 组成。一个 Block 会被完整地调度到一个 SM 上。
  - **CUDA Core**：SM 内部最基本的**标量处理单元**，负责执行具体的浮点数和整数运算。一个 SM 包含多个 CUDA Core (比如128个)。
  - **Tensor Core**：专门为深度学习中的**矩阵乘加运算 (MMA)** 设计的**专用硬件单元**。它可以在一个时钟周期内，完成一个小的矩阵乘法（比如 4x4 的 FP16 矩阵乘法），其计算效率远超 CUDA Core。
  - **Warp Scheduler (线程束调度器)**：SM 内部的“交通警察”，负责从分配给该 SM 的所有 Warp 中，挑选出**已经准备就绪**（数据已到达、没有依赖）的 Warp，并把它送上 CUDA Core 或 Tensor Core 去执行。
- **内存层级 (从最慢、最大 到 最快、最小)**:
  1. **Global Memory (全局内存)**：就是我们常说的**显存 (DRAM)**。容量最大（GB级别），但**速度最慢**。所有线程都可以读写，是 CPU 和 GPU 之间数据交换的主要场所。
  2. **L2 Cache (二级缓存)**：位于所有 SM 之间，被**所有 SM 共享**。用来缓存对全局内存的访问，缓解访存压力。
  3. **Shared Memory (共享内存)**：位于**每个 SM 内部**的一块高速缓存（SRAM）。它的速度**远快于**全局内存。**同一个 Block 内**的所有线程可以共享这块内存，用于线程间的快速数据交换。这是 CUDA 编程优化的关键。
  4. **L1 Cache (一级缓存)**：同样位于**每个 SM 内部**，与 Shared Memory 共享一块物理 SRAM。用于缓存单个线程对全局内存的访问。
  5. **Register File (寄存器文件)**：位于**每个 SM 内部**，是**最快**的存储单元。每个线程拥有自己**私有**的寄存器，用于存放临时变量。

2. GPU 和 GPGPU 的区别是什么?

- **GPU (Graphics Processing Unit, 图形处理器)**：是其**原始定义**，指专门为**图形渲染**（比如玩游戏、做动画）设计的硬件。它的核心任务是处理顶点、像素、纹理等图形学相关的并行计算。
- **GPGPU (General-Purpose computing on GPU, 通用计算图形处理器)**：指利用 GPU 强大的并行计算能力，去执行**非图形学**的、**通用的科学与工程计算。**我们现在做的深度学习、物理模拟、密码破解等，都属于 GPGPU 的范畴。
- **区别总结**：GPGPU 本质上就是现代 GPU 的**一种应用模式和能力。**现代的高端 Nvidia GPU (如 A100, H100) 在设计时，就同时考虑了图形和通用计算的需求，集成了 Tensor Core 等专门为 GPGPU 优化的硬件，它们既是 GPU 也是 GPGPU。可以说，GPGPU 是 GPU 功能的**一次伟大扩展。**

3. Ampere 和 Hopper 两代 GPGPU 的显著特点与优化方向?

- **Ampere (安培) 架构 (代表：A100)**：
  - **显著特点**：
    1. **第三代 Tensor Core**：引入了对 **TF32** 和 **BFloat16** 数据类型的支持，在不损失太多精度的情况下，极大提升了计算吞吐量。并支持**稀疏化 (Sparsity)**，能对权重矩阵中的零值进行跳过，实现2倍性能提升。
    2. **MIG (Multi-Instance GPU)**：首次引入的技术，可以将一块 A100 GPU **物理上分割**成多达 7 个独立的、更小的 GPU 实例。这极大地提升了 GPU 在处理多个小任务时的资源利用率。
    3. **更大的 L2 缓存和 HBM2 显存**：提供了极高的访存带宽。
- **Hopper (霍珀) 架构 (代表：H100)**：
  - **显著特点与优化方向**：
    1. **第四代 Tensor Core 与 FP8 支持**：引入了对 **FP8 (8位浮点数)** 格式的硬件支持。相比 FP16，FP8 能将计算吞吐量和内存带宽**再次翻倍**，是大模型推理和训练的重大利器。
    2. **Transformer 引擎 (Transformer Engine)**：这是 Hopper 架构的**杀手锏。**它是一个软硬件结合的系统，能够**动态地、智能地**在 FP16 和 FP8 精度之间进行切换，在保证模型精度的前提下，最大化地利用 FP8 带来的性能优势。
    3. **DPX 指令集**：针对动态规划等特定算法的专用加速指令。
    4. **第二代 MIG**：提供了更安全、更隔离的多实例能力。
    5. **NVLink Switch System**：极大地提升了多 GPU 之间互联的带宽和效率，为超大规模模型训练铺平了道路。

总结：Ampere 的核心是引入了灵活的数据类型和多实例能力，而 Hopper 则是在此基础上，针对 Transformer 和大模型的特点，引入了 FP8 和 Transformer 引擎这一革命性的优化。

并行基础

1. 什么是指令并行，数据并行，任务并行?

- **指令并行 (Instruction-level Parallelism, ILP)**：在**单个处理器核心**内部，通过流水线、乱序执行、多发射等技术，让**多条独立的指令在同一时刻处于执行的不同阶段。**这是现代 CPU 自动实现的并行，程序员通常无感知。

- **数据并行 (Data Parallelism)**：**用多个处理器，对一个巨大的数据集的不同部分，执行完全相同的操作。**这是 GPU 和 SIMD 最擅长的模式。比如，将一张大图片切成 100 块，交给 100 个核心，每个核心都对自己的那一小块执行“锐化”这个相同的操作。
- **任务并行 (Task Parallelism)**：**用多个处理器，同时执行多个不同的、独立的任务。**比如，一个处理器在渲染视频，另一个在压缩文件，第三个在播放音乐。这些任务之间逻辑不同，没有依赖关系。

2. 什么是 SIMD？什么是 SIMT？什么是 SPMD?

- **SIMD (Single Instruction, Multiple Data)**：**单指令，多数据流。**一条指令，可以同时作用于一个向量中的多个数据。这是 CPU 中 **AVX/SSE 指令集**的执行模式。比如 `_mm256_add_ps` 就是一条 SIMD 指令。
- **SIMT (Single Instruction, Multiple Threads)**：**单指令，多线程流。**这是 **Nvidia GPU** 的核心编程模型。一个 Warp (32个线程) 在同一时刻**必须执行相同的指令**，但每个线程可以处理不同的数据。它比 SIMD 更灵活，因为它允许程序员写看起来像是普通线程的代码，而由硬件（Warp Scheduler）来管理这些线程的执行。它允许线程有独立的执行路径（通过掩码实现），尽管这会带来 Warp 分化（Warp Divergence）的性能损失。
- **SPMD (Single Program, Multiple Data)**：**单程序，多数据流。**这是更高层次的编程模型，指多个处理器独立地运行**同一个程序**的副本，但各自处理不同的数据。MPI (Message Passing Interface) 和我们写的 CUDA Kernel 都属于 SPMD 模式。SIMT 可以看作是实现 SPMD 的一种硬件机制。

CUDA 编程

1. 了解 CUDA 的编程范式，如何写一个简单的 CUDA 程序?

- **编程范式**：CUDA 是一种**异构计算 (Heterogeneous Computing)** 编程范式。这意味着程序由两部分组成：
    1. **主机代码 (Host Code)**：运行在 **CPU** 上的标准 C++ 代码，负责处理串行逻辑、内存管理（分配和数据传输）、以及启动 GPU 计算任务。
    2. **设备代码 (Device Code)**：运行在 **GPU** 上的并行计算代码，也称为 **Kernel (核函数)。**这部分代码会被成千上万个线程同时执行。
- **如何写一个简单的 CUDA 程序 (向量加法)**：

```
#include <iostream>

// 设备代码 (Kernel): 在 GPU 上执行
// __global__ 关键字表示这个函数可以从 CPU 调用，在 GPU 上执行
__global__ void vectorAdd(float* A, float* B, float* C, int n) {
    // 计算当前线程的全局唯一索引
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // 确保线程索引没有越界
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}

int main() {
    // --- 主机代码 (在 CPU 上执行) ---
```

```
    int n = 1024;
    size_t size = n * sizeof(float);

    // 1. 在主机上分配内存
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

    // 初始化主机上的数据
    for (int i = 0; i < n; i++) {
        h_A[i] = i;
        h_B[i] = i * 2;
    }

    // 2. 在设备 (GPU) 上分配内存
    float* d_A, * d_B, * d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    // 3. 将数据从主机拷贝到设备 (CPU -> GPU)
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // 4. 定义 Kernel 的执行配置 (启动参数)
    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;

    // 5. 启动 Kernel (在 GPU 上执行并行计算)
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, n);

    // 6. 将结果从设备拷贝回主机 (GPU -> CPU)
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // (7. 验证结果，释放内存...)
    // ...

    return 0;
}
```

2. 一个 CUDA 程序的执行过程是什么?

典型的执行流程如上例所示，可以总结为：

1. **初始化**：CPU 分配主机内存，准备数据。
2. **分配设备内存**：CPU 指示 GPU 分配相应的显存。
3. **数据传输 (H2D)**：CPU 将输入数据从主机内存拷贝到 GPU 显存。
4. **Kernel 启动**：CPU 设置好线程网格（Grid）和线程块（Block）的维度，然后启动 GPU 上的核函数。**这个启动是异步的**，CPU 下达命令后就立刻返回，不会等待 GPU 执行完毕。
5. **并行计算**：GPU 上的成千上万个线程并发执行核函数代码。

6. **同步（可选但常用）**：CPU 在需要使用 GPU 计算结果之前，必须调用一个同步函数（如 `cudaDeviceSynchronize()`），等待 GPU 完成所有工作。
7. **数据传回 (D2H)**：CPU 将计算结果从 GPU 显存拷贝回主机内存。
8. **清理**：CPU 释放主机和设备上的内存。

3. 了解 CUDA 中的 `__global__`, `__device__` 等的用法

这些是函数类型限定符，告诉编译器这个函数在哪里执行、可以从哪里调用：

- `__global__`：定义一个 **Kernel (核函数)**。
  - **执行位置**：GPU。
  - **调用位置**：CPU。
  - **特点**：必须返回 `void` 类型。通过特殊的 `<<<...>>>` 语法调用。
- `__device__`：定义一个 **设备函数**。
  - **执行位置**：GPU。
  - **调用位置**：只能从 `__global__` 或另一个 `__device__` 函数中调用。
  - **特点**：用于在 GPU 代码中封装可复用的功能，就像普通的 C++ 辅助函数一样。
- `__host__`：定义一个 **主机函数**。
  - **执行位置**：CPU。
  - **调用位置**：CPU。
  - **特点**：这是 C++ 函数的**默认类型**，不写也行。`__host__ __device__` 组合则表示这个函数既可以被 CPU 调用，也可以被 GPU 调用。

4. 了解 CUDA 的异步特性 (CUDA Stream，CUDA Graph)

- **CUDA Stream (流)**：
  - **是什么**：一个**任务队列**。你可以把一系列 CUDA 操作（如数据拷贝、Kernel 启动）都放入同一个 Stream 中，这些操作会**按顺序执行**。
  - **异步特性**：你可以创建**多个 Stream**，不同 Stream 中的任务可以**并行执行**！这正是实现"计算与通信重叠"的核心工具。你可以让一个 Stream 负责把下一批数据从 CPU 拷贝到 GPU，同时让另一个 Stream 在 GPU 上处理上一批已经拷贝好的数据，再让第三个 Stream 把更早算完的结果拷贝回 CPU。
- **CUDA Graph (图)**：
  - **是什么**：是对一系列 CUDA 操作的**"预录制"和"固化"**。你可以先在"捕获"模式下，执行一遍你的所有 CUDA 操作，CUDA 会把这个操作序列（包括它们的依赖关系）记录下来，形成一个固定的"图"。
  - **异步特性**：之后，你可以**一次性地、以极低的开销"重放" (Launch)** 整个图。这极大地减少了 CPU 逐个启动 Kernel 的开销，对于那些计算模式固定的循环（比如神经网络的推理），使用 CUDA Graph 能带来巨大的性能提升。

5. 思考如何写一个简单的 Matmul，以及会面临的问题 (Bank Conflict, Warp Divergent)

- **如何写一个简单的 Matmul (思路)**:
  最朴素的 CUDA Matmul Kernel，会让**每个线程负责计算输出矩阵 C 的一个元素** `C[i][j]`。

```
__global__ void matmul_naive(float* A, float* B, float* C, int M, int K, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M && col < N) {
        float sum = 0.0f;
        for (int p = 0; p < K; p++) {
            sum += A[row * K + p] * B[p * N + col];
        }
        C[row * N + col] = sum;
    }
}
```

- **会面临的问题**:
  1. **极高的全局内存访问**：上述实现中，每个线程为了计算一个 `sum`，需要从全局内存中读取 `K` 个 A 的元素和 `K` 个 B 的元素。当线程数量巨大时，这会对显存带宽造成毁灭性的压力。 **(优化的关键是使用 Shared Memory)**
  2. **Bank Conflict (银行冲突)**:
     - **是什么**：Shared Memory 为了提高并行访问速度，被分成了多个（通常是32个）等宽的存储单元，称为 **Bank**。你可以把它们想象成银行的 32 个并排的业务窗口。
     - **冲突发生**：如果**同一个 Warp** 中的多个线程，**同时访问了同一个 Bank**（同一个业务窗口），就会发生冲突。这些访问请求必须**串行化**，一个一个来，导致并行度急剧下降。
     - **在 Matmul 中如何发生**：如果线程在访问 Shared Memory 中的数据时，访问模式（步长）设计不当（比如按列访问），就很容易导致多个线程访问的地址落在同一个 Bank 里。
  3. **Warp Divergence (线程束分化)**:
     - **是什么**：同一个 Warp 的 32 个线程**必须**执行相同的指令。如果代码中出现了 `if-else` 这样的分支，且 Warp 内的线程根据各自的数据，**走了不同的分支**（比如线程0走 `if`，线程1走 `else`），就会发生分化。
     - **硬件如何处理**：硬件会**串行地**执行这两个分支。先执行 `if` 分支（此时 `else` 分支的线程被“屏蔽”和等待），然后再执行 `else` 分支（此时 `if` 分支的线程被屏蔽和等待）。这使得原本的并行计算退化成了串行，性能大幅下降。

如何降低 CUDA 难度

1. 引入 Triton，了解 Triton 想解决什么问题

- **Triton 想解决的问题**：
    1. **降低 GPU 编程门槛**：直接用 CUDA 写高性能算子（比如 FlashAttention）极其困难，需要开发者手动管理内存、处理同步、避免 Bank Conflict 等，还需要深厚的硬件知识。Triton 提供了**类似 Python 的、更高层次的抽象**，让开发者能用更简洁、更直观的代码来编写 GPU Kernel。
    2. **性能可移植性 (Performance Portability)**：专家手写的 CUDA 代码通常是为某一代特定的 GPU 架构（如 Ampere）深度优化的。当新的架构（如 Hopper）出现时，这些代码可能不再是最优的，需要重写。
    3. **自动化优化**：Triton 拥有一个**强大的编译器**，它会把你写的"高级"Triton 代码，自动地、智能地编译成针对特定硬件的高度优化的底层代码（PTX）。它会自动处理分块、内存合并、指令调度等复杂的优化，让不那么懂硬件的开发者也能写出高性能代码。

总结：Triton 的使命，就是让更多的开发者能够**轻松地、快速地写出接近专家手写 CUDA 性能水平的、并且能在不同代 GPU 上都保持高性能**的自定义算子。

Bonus Time!

- **Pytorch 算子下降到底层 CUDA 代码的过程**：
  这是一个非常复杂的"编译"过程，大致可以分为几步：
    1. **Python 前端**：用户调用 PyTorch API (`torch.matmul`)。
    2. **ATen (A Tensor Library)**：PyTorch 的 C++ 核心库。Python 调用被分派到 ATen 里的具体 C++ 函数实现。
    3. **分发 (Dispatch)**：ATen 会根据当前的硬件（是 CPU 还是 CUDA GPU）和数据类型，选择一个最合适的"后端"来实现这个运算。
    4. **CUDA 后端实现**：对于 CUDA 设备，这个调用最终会落到一个具体的 `.cu` 文件里，这个文件里包含了调用 Nvidia **cuBLAS** 库（用于标准 GEMM）、**cuDNN** 库（用于标准卷积）或者 PyTorch 自己手写的、高度优化的 CUDA Kernel 的逻辑。
    5. **CUDA Driver API / Runtime API**：这些 C++ 代码最终通过 CUDA 驱动，向 GPU 发送指令，启动 Kernel，执行计算。
    - **对于 Triton**：现在 PyTorch 2.x 引入了 `torch.compile`，它可以使用 TorchInductor 作为后端。TorchInductor 可以将 PyTorch 的计算图，直接**生成为高效的 Triton 代码**，然后再由 Triton 编译器编译成 PTX，实现了端到端的自动化优化。
- **ThunderKittens 是什么? 有什么优点**
    - **是什么**：ThunderKittens 是一个由 **Tri Dao**（FlashAttention 的作者之一）领导开发的、专注于**稀疏计算 (Sparsity)** 的开源项目/库。它旨在为稀疏矩阵运算提供和稠密运算库（如 cuBLAS）一样易用且高性能的解决方案。
    - **优点**：
        1. **高性能**：它提供了针对现代 GPU 架构高度优化的稀疏计算 CUDA Kernel，其性能远超通用的稀疏计算库。
        2. **易用性**：它提供了简洁的 API，让开发者可以轻松地在自己的项目中使用这些高效的稀疏算子，而无需自己去写复杂的 CUDA 代码。

3. **专注前沿**：它专注于解决现代大模型中越来越重要的“稀疏化”问题，比如 Mixture of Experts (MoE) 模型中的稀疏激活。