Weiming HPC Training Camp
×
LCPU AI Infra Seminars

北京大学未名超算队
北京大学学生 Linux 俱乐部

活动简介 OVERVIEW

北京大学
未名超算队
学生 Linux 俱乐部

HPC × Infra

4个Topics

这个暑假教你大模型如何

训更快、推更省、跑更好

学习和实践 AI Infra！

活动简介 OVERVIEW

- 提供算力、
- 内容设计、
- 实践练习、
- 评测排行、
- 交流讨论、
- 业界交流、
- 企业直通、
- ……

活动简介 OVERVIEW

- 提供算力、
- 内容设计、
- 实践练习、
- 评测排行、
- 交流讨论、
- 业界交流、
- 企业直通、
- ……

专题	时间	Keywords
Kernel & ML Compiler	3 周	CUDA, GPU Programming, DSL, Layout, Pipeline Ordering, Complier Stack, SoL Kernel Exlpained, Hardware
Interconnect & Communication	1 周	Scale-Up, Scale-Out, Interconnect, Network Fabric, P2P, Collective Communication, EP/MoE, Communication Overlap, Distributed Kernel, Memory-Storage Co-design
LLM Serving & Inference	2 周	KV Cache, SGLang/vLLM, 框架与算子，并行策略，PD分离，投机解码
Distributed Reinforcement Learning Systems	1 周	RL算法、框架、Rollout、训推一致性，RL 投机解码，分布式 RL 系统、Agentic RL

暑期部分内容

活动简介 OVERVIEW

- 提供算力、
- 内容设计、
- 实践练习、
- 评测排行、
- 交流讨论、
- 业界交流、
- 企业直通、
- ……

```
"""
First let's implement the trivial extension of conv1d to
multiple output channels, based on the
above F=1 version.
"""



@tilelang.jit(
    pass_configs={
    tilelang.PassConfigKey.TL_DISABLE_WARP_SPECIALIZED: True,
    tilelang.PassConfigKey.TL_DISABLE_TMA_LOWER: True,
    },
)
def tl_conv1d_multi_outchannel(X, K, BLOCK_N: int, BLOCK_L:
int):
    N, L, KL, F = T.const("N, L, KL, F")
    dtype = T.float16
    accum_dtype = T.float32
    X: T.Tensor((N, L), dtype)
    K: T.Tensor((KL, F), dtype)
    O = T.empty((N, L, F), dtype)


    # TODO: Implement this function


    return O
```

（推理部分会有 vLLM 框架优化实践）

活动简介 OVERVIEW

- 提供算力、
- 内容设计、
- 实践练习、
- 评测排行、
- 交流讨论、
- 业界交流、
- 企业直通、
- ……

活动简介 OVERVIEW

- 提供算力、
- 内容设计、
- 实践练习、
- 评测排行、
- 交流讨论、
- 业界交流、
- 企业直通、
- ……

活动赞助

Tencent 腾讯

宽德投资 WIZARD QUANT

合作伙伴

LLM

关于我们 ABOUT US

北京大学学生 Linux 俱乐部

北京大学学生 Linux 俱乐部 (Linux Club of Peking University, 简称 **LCPU**) 成立于 2003 年，是北京大学最活跃的学术科创类社团之一。我们致力于学习和研究 Linux 操作系统及其他开源软硬件技术，并通过多个高质量项目和活动推动开源精神的发展。

LCPU是一个年轻、多元、充满活力的团队！欢迎大家积极报名骨干团队，一起为社团发展做出贡献。

北京大学 未名超算队

北京大学未名超算队组建于2016年，积淀了崇尚技术、团结协作、互助包容的文化氛围。大家齐心协力，取得了令人瞩目的成就：不仅斩获 ASC 2023、2024、**2026** 冠军，还荣膺 ASC 2025、SC23 SCC 和 SC24 SCC 亚军、 SC23 SCC Highest Linpack， 并在 ISC 2026 SCC 线上赛中**首战夺魁**，展现了北京大学在高性能计算领域的卓越实力。

欢迎更多热爱超算的同学加入我们！ （本科新生优先）

0. From HPC To AI Infra
并行计算与并行编程

Chen Jiajun
2026. 07. 25

Weiming Supercomputing Team
Linux Club of Peking University

Linux俱乐部项目

什么是计算 COMPUTING

计算是一种将输入值按照特定规则转换为输出的过程

- 加、减、乘、除
- 乘方、开根
- 指数、对数
- 比较
- ……
- 矩阵乘法

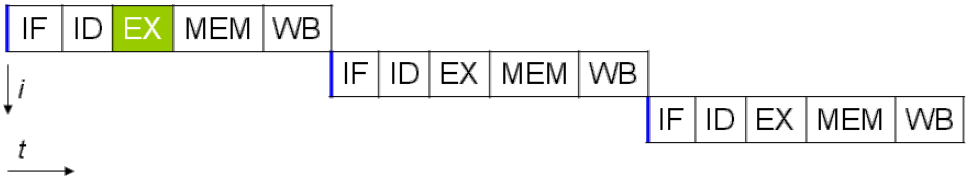
输入输出：数据与数据搬运

特定规则：计算单元

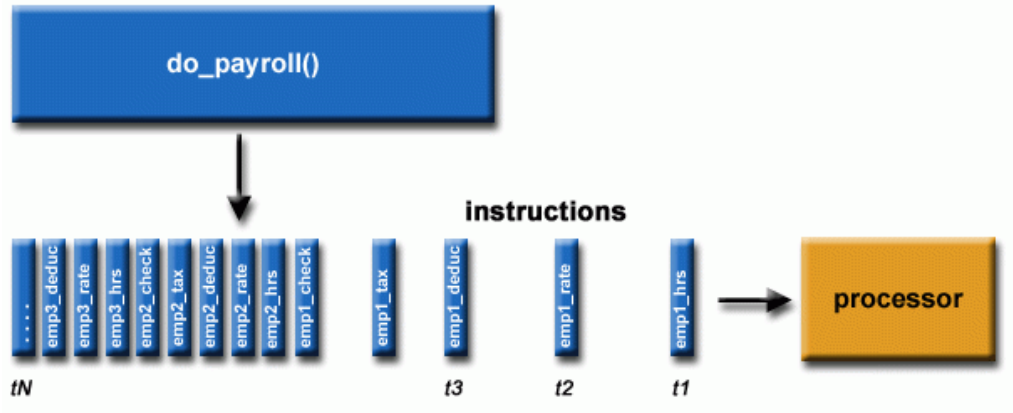
如何计算 COMPUTING

用计算机进行计算，用程序描述计算

- 计算机程序是由处理器执行的一连串指令流
- 时钟周期是处理器执行指令的基本单位



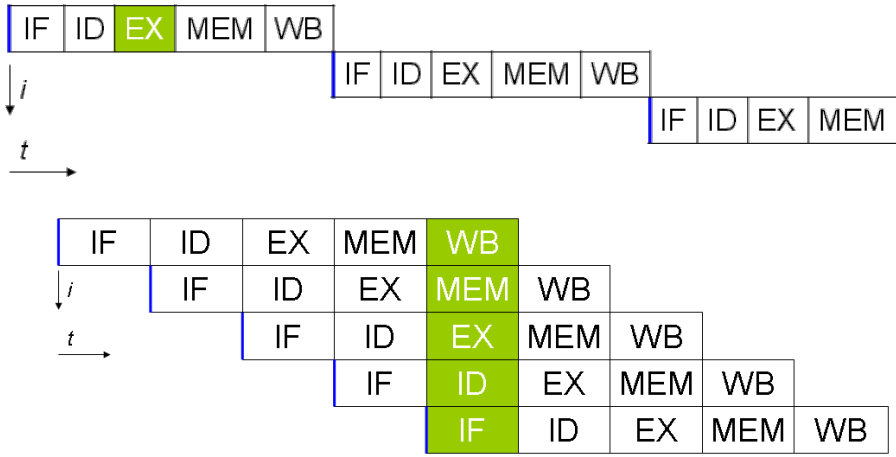
取指、译码、执行、访存、写回
IPC性能（Instructions Per Cycle）

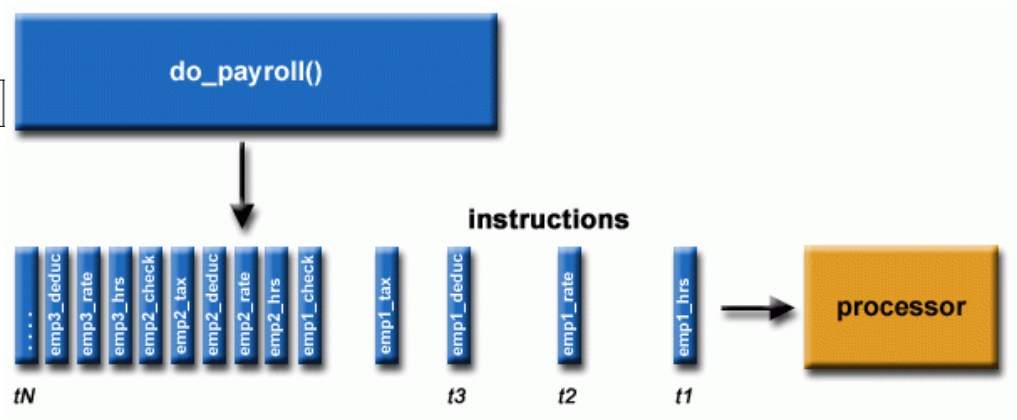


如何计算 COMPUTING

用计算机进行计算，用程序描述计算

- 计算机程序是由处理器执行的一连串指令流
- 时钟周期是处理器执行指令的基本单位





取指、译码、执行、访存、写回

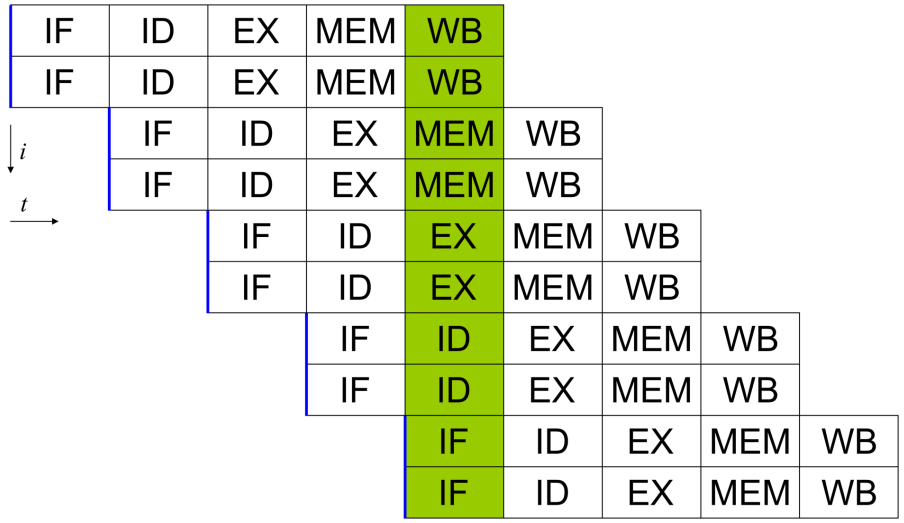
IPC性能（Instructions Per Cycle）

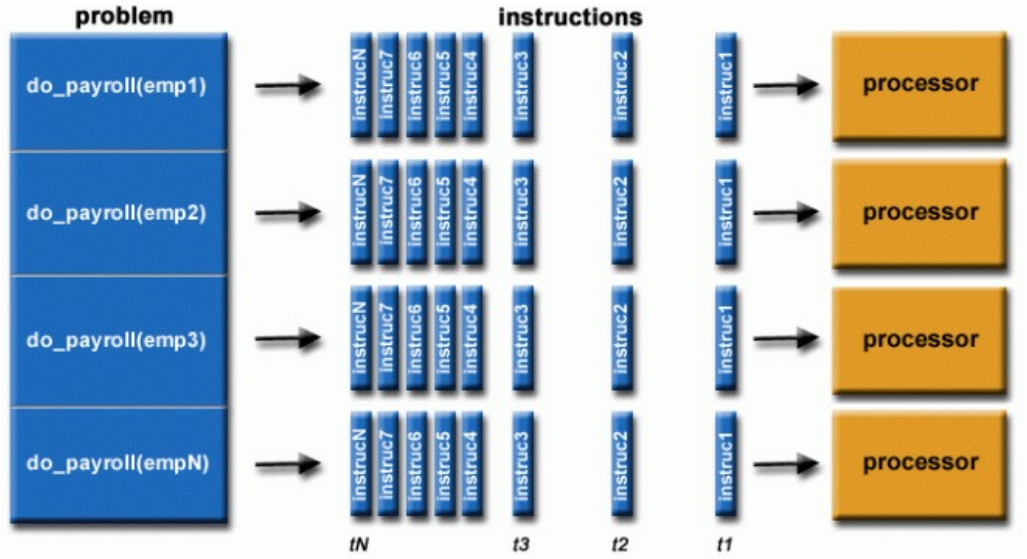
并行计算 PARALLEL COMPUTING

用计算机进行计算，用程序描述计算

- 计算机程序是由处理器执行的一连串指令流
- 时钟周期是处理器执行指令的基本单位

不止一个计算单元!





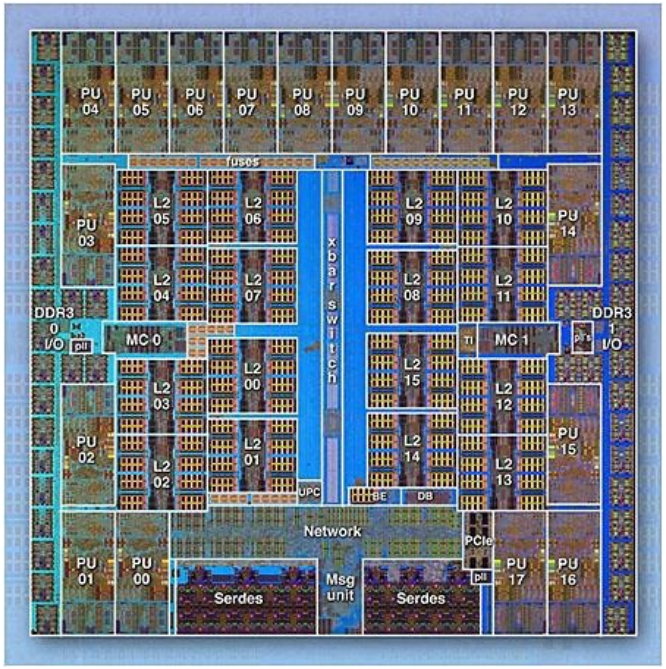
IPC > 1， 超标量性能

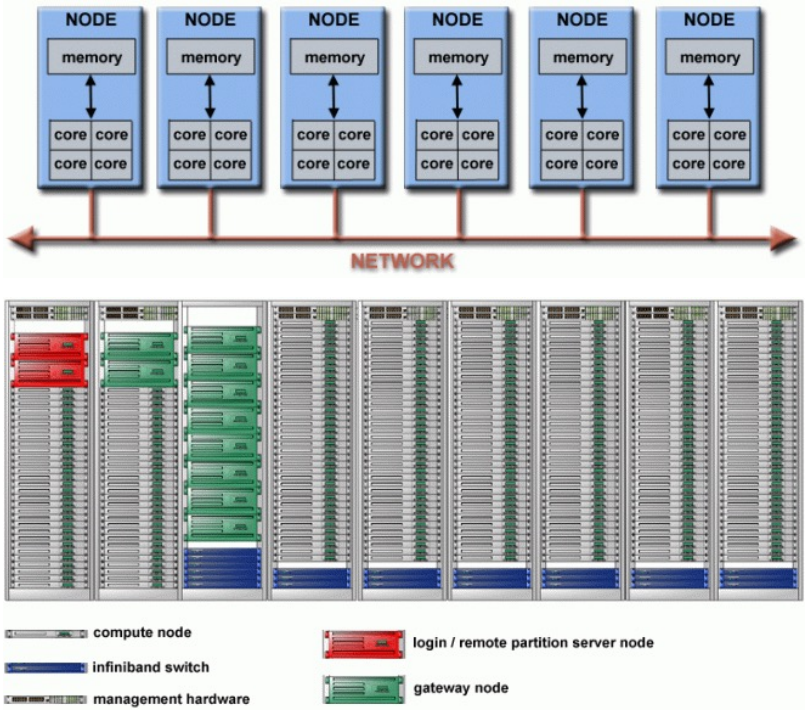
并行计算 PARALLEL COMPUTING

用计算机进行计算，用程序描述计算

- 计算机程序是由处理器执行的一连串指令流
- 时钟周期是处理器执行指令的基本单位

不止一个计算单元!





并行计算 PARALLEL COMPUTING

用计算机进行计算，用程序描述计算

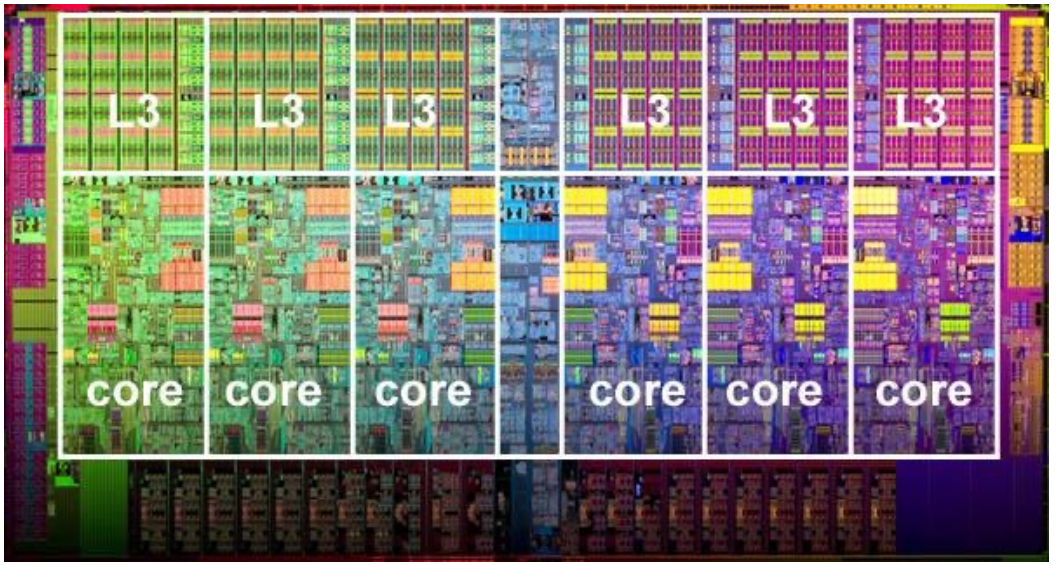
- 计算机程序是由处理器执行的一连串指令流
- 时钟周期是处理器执行指令的基本单位

不止一个计算单元!

衡量 “性能”

- 延迟 Latency
  - -> 单个计算单元更强 每次算得更快
  - -> 频率
- 吞吐 Throughput
  - -> 多个计算单元的计算量充分累加
  - -> 核心数

并行化 与 并行度!



从串行到并行 PARALLELIZATION

什么样的计算可并行?

- 数据依赖关系：Bernstein's Conditions
- 竞争、互斥与锁
- 通信与同步

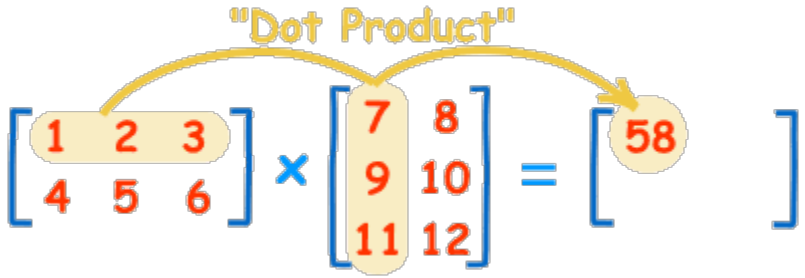
```
1: function Dep(a, b)
2: c := a * b
3: d := 3 * c
4: end function
```

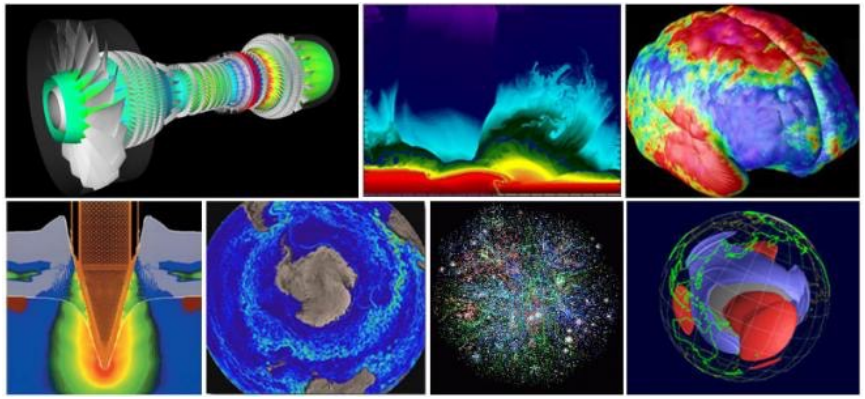
```
1: function NoDep(a, b)
2: c := a * b
3: d := 3 * b
4: e := a + b
5: end function
```

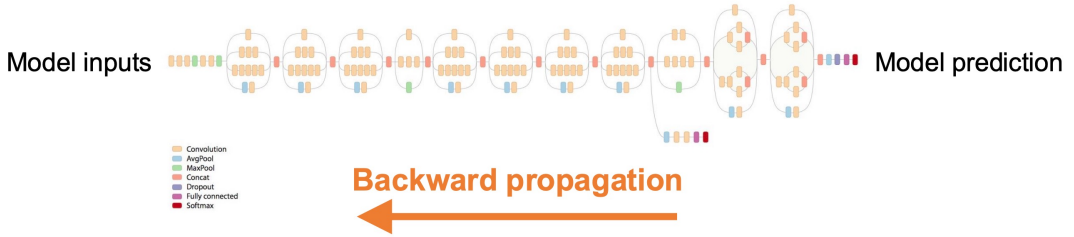
并行机会 PARALLELIZATION OPPORTUNITY

什么样的计算可并行?

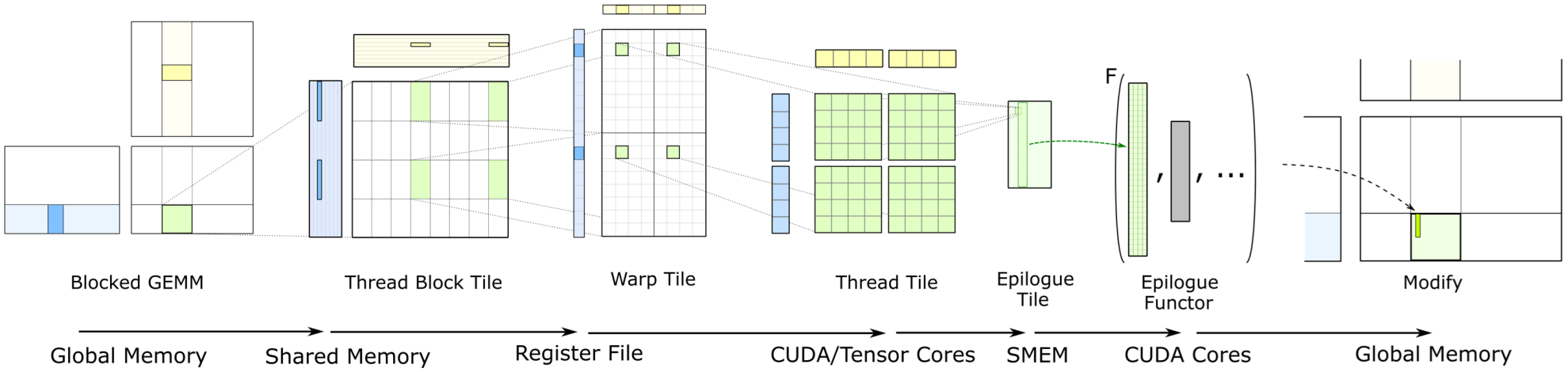
- 科学计算：气候模拟、物理模拟、生物模拟 .....
- 矩阵计算、逐元素计算：深度学习与人工智能







并行机会 PARALLELIZATION OPPORTUNITY



- GemmGlobalIteratorAb
- Transformer
- GemmSharedStoreTileAb

Global Load Stream

- GemmSharedLoadTile{A,B}

Shared Load Stream

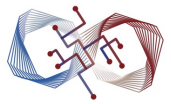
- fma, dp4a
- WMMA

Matrix Multiply

- Transformer
- GemmSharedStoreTileD
- GemmSharedLoadTileD
- Functor
- GemmGlobalIteratorC
- GemmGlobalIteratorD

Epilogue

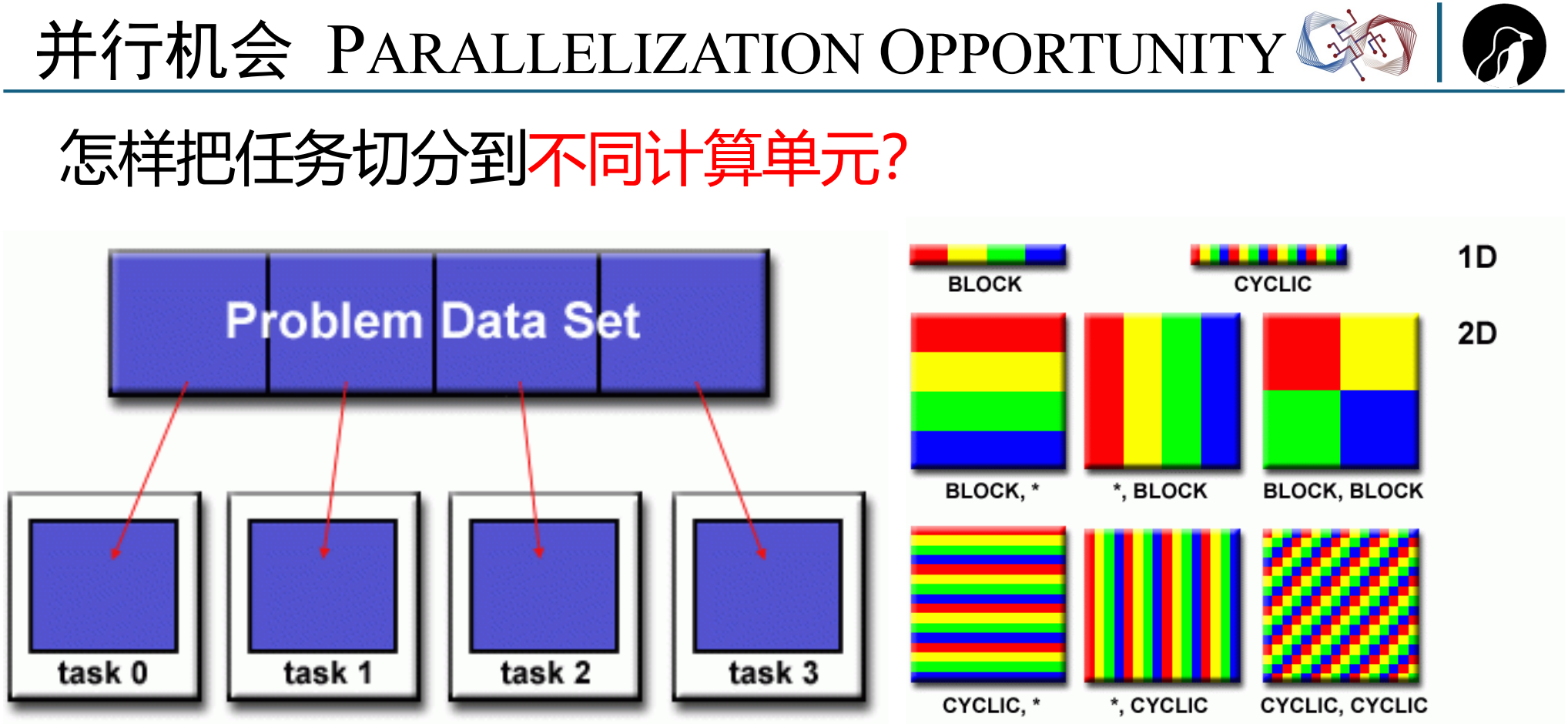
并行机会 PARALLELIZATION OPPORTUNITY



怎样把任务切分到不同计算单元?

充分并行化，打满并行度!

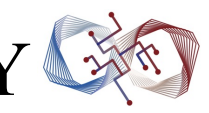
并行机会 PARALLELIZATION OPPORTUNITY



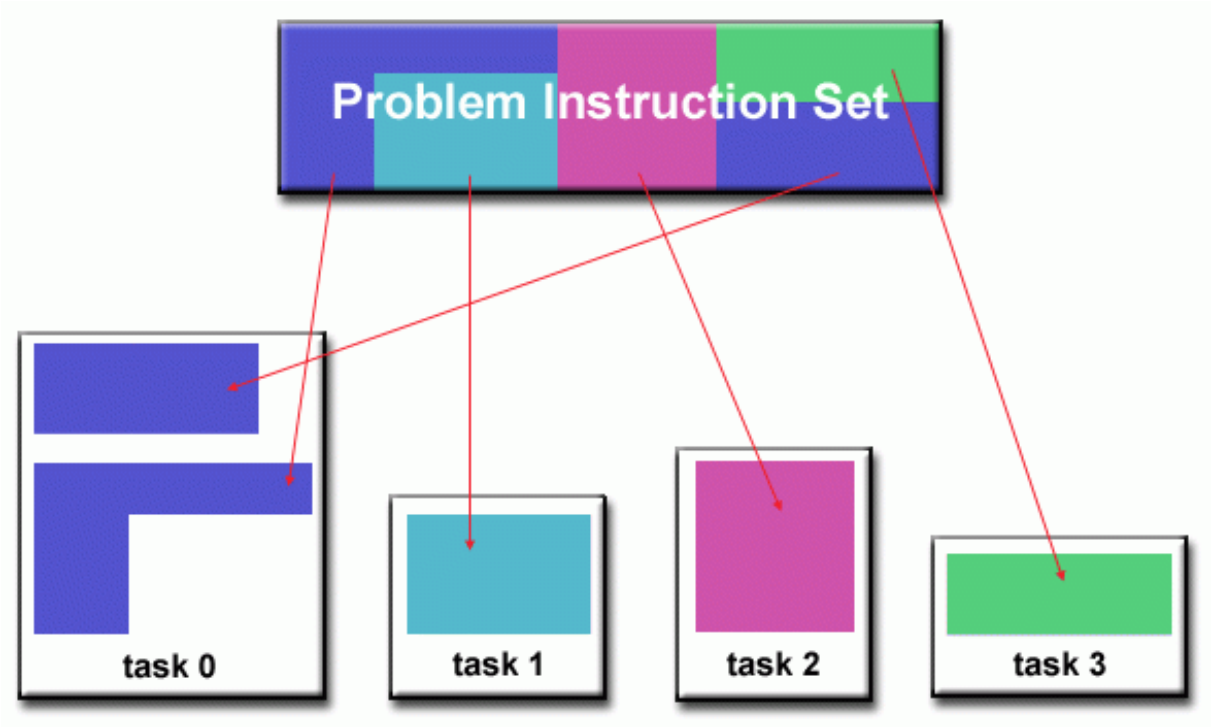
怎样把任务切分到不同计算单元?

Domain Decomposition

并行机会 PARALLELIZATION OPPORTUNITY

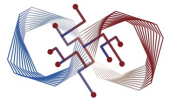


怎样把任务切分到不同计算单元?



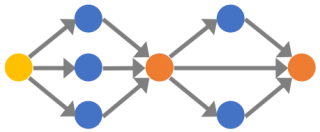
Functional Decomposition

并行机会 PARALLELIZATION OPPORTUNITY



怎样把任务切分到不同计算单元?

Data Parallelism



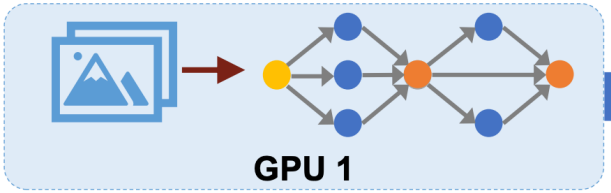
ML Model

Dataset

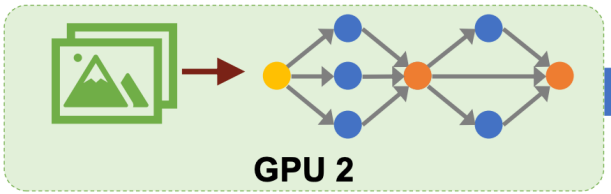
Data Parallelism

$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

1. Partition dataset into batches

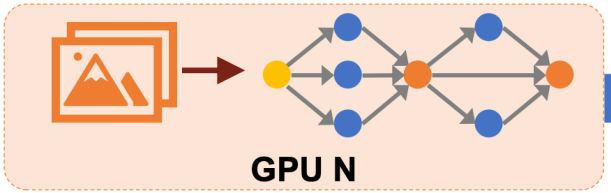


GPU 1



GPU 2

...



GPU N

2. Forward/backward of each batch on a GPU

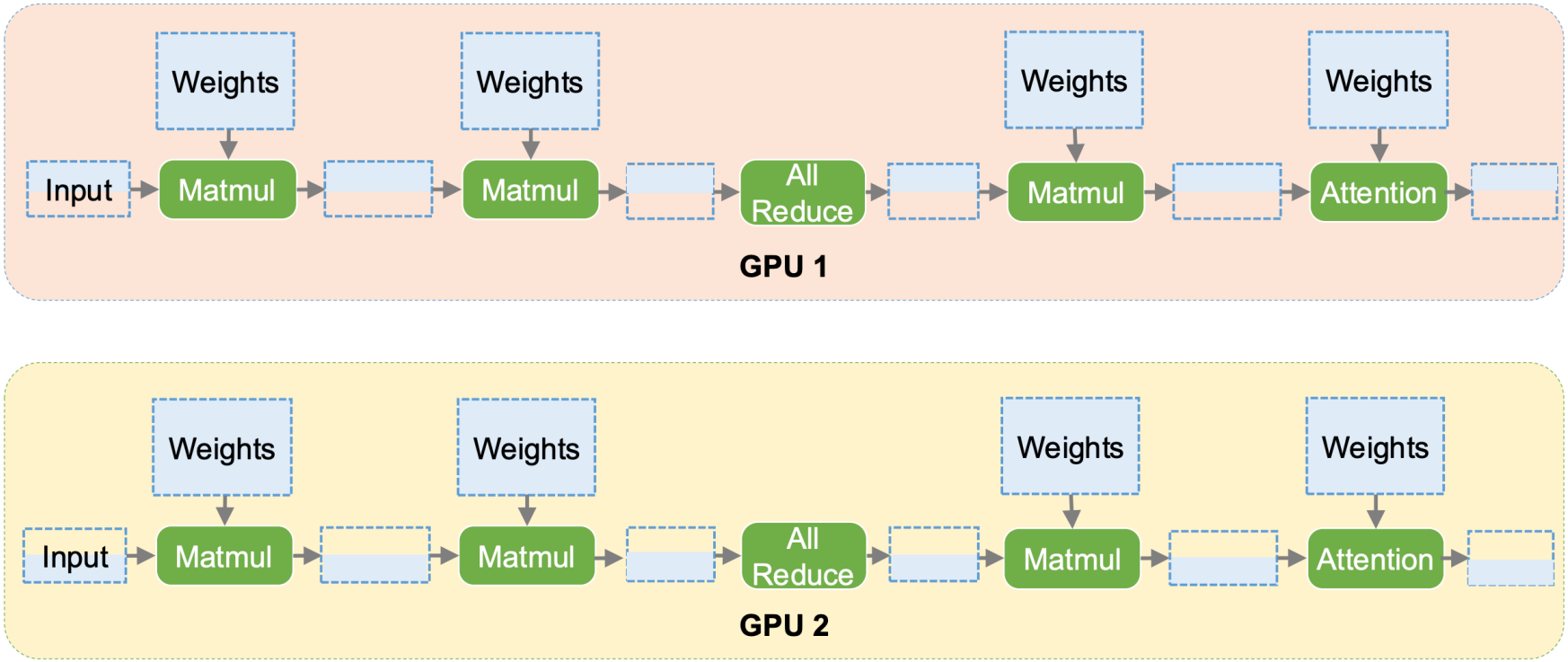
Gradients Aggregation

3. Aggregate gradients across GPUs

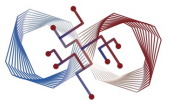
并行机会 PARALLELIZATION OPPORTUNITY

怎样把任务切分到不同计算单元?

Data Parallelism for Transformer



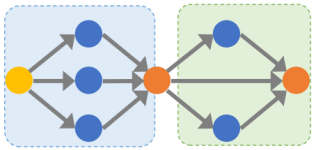
并行机会 PARALLELIZATION OPPORTUNITY



怎样把任务切分到不同计算单元?

Model Parallelism

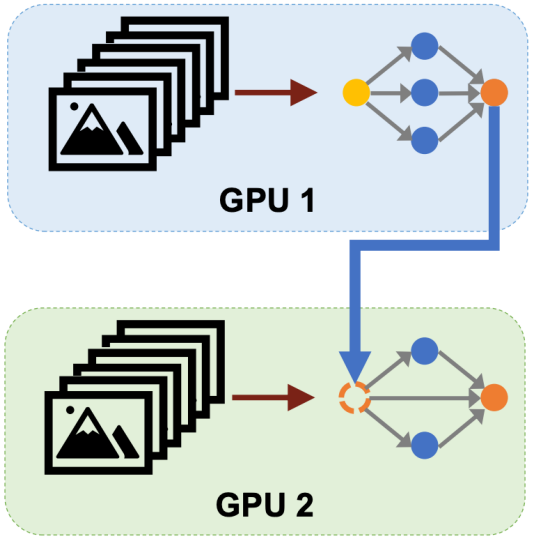
- Split a model into multiple subgraphs and assign them to different devices



ML Model

Dataset

Model Parallelism



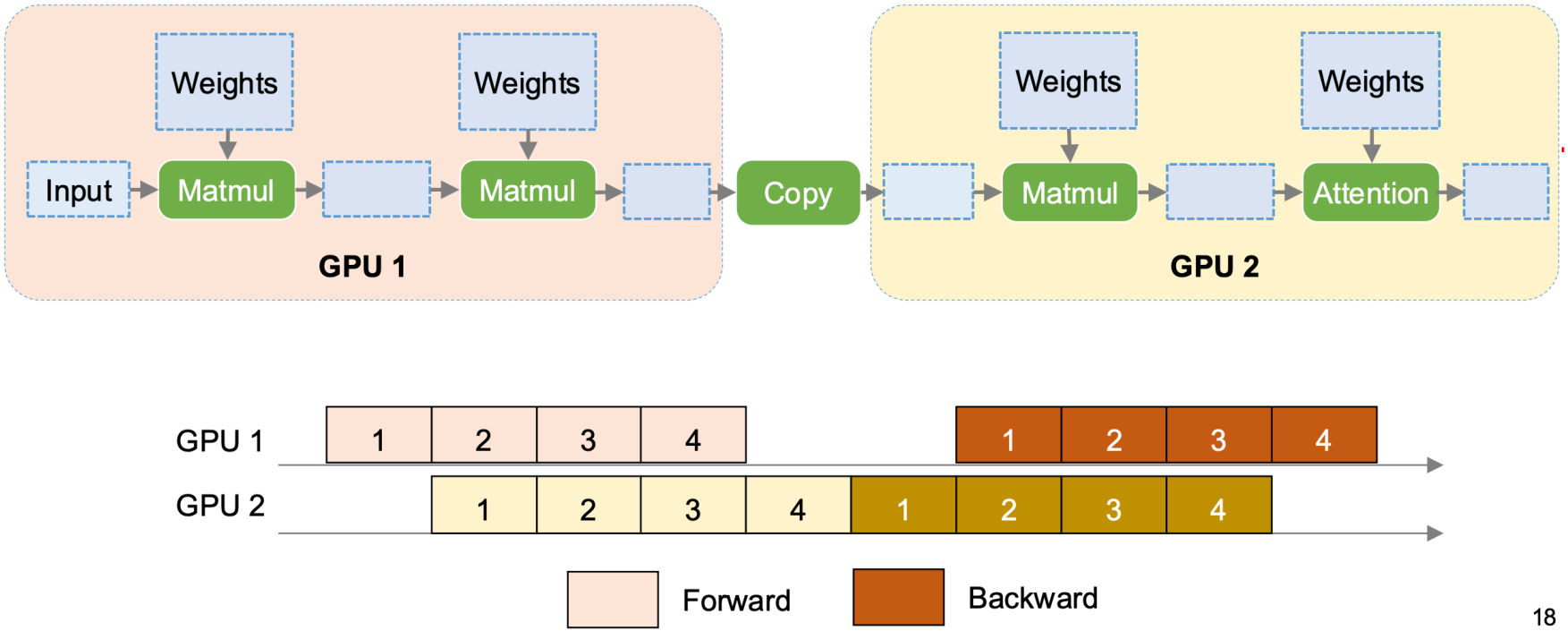
Transfer intermediate results between devices

$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

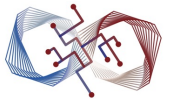
并行机会 PARALLELIZATION OPPORTUNITY

怎样把任务切分到不同计算单元?

Pipeline Model Parallelism for Transformer

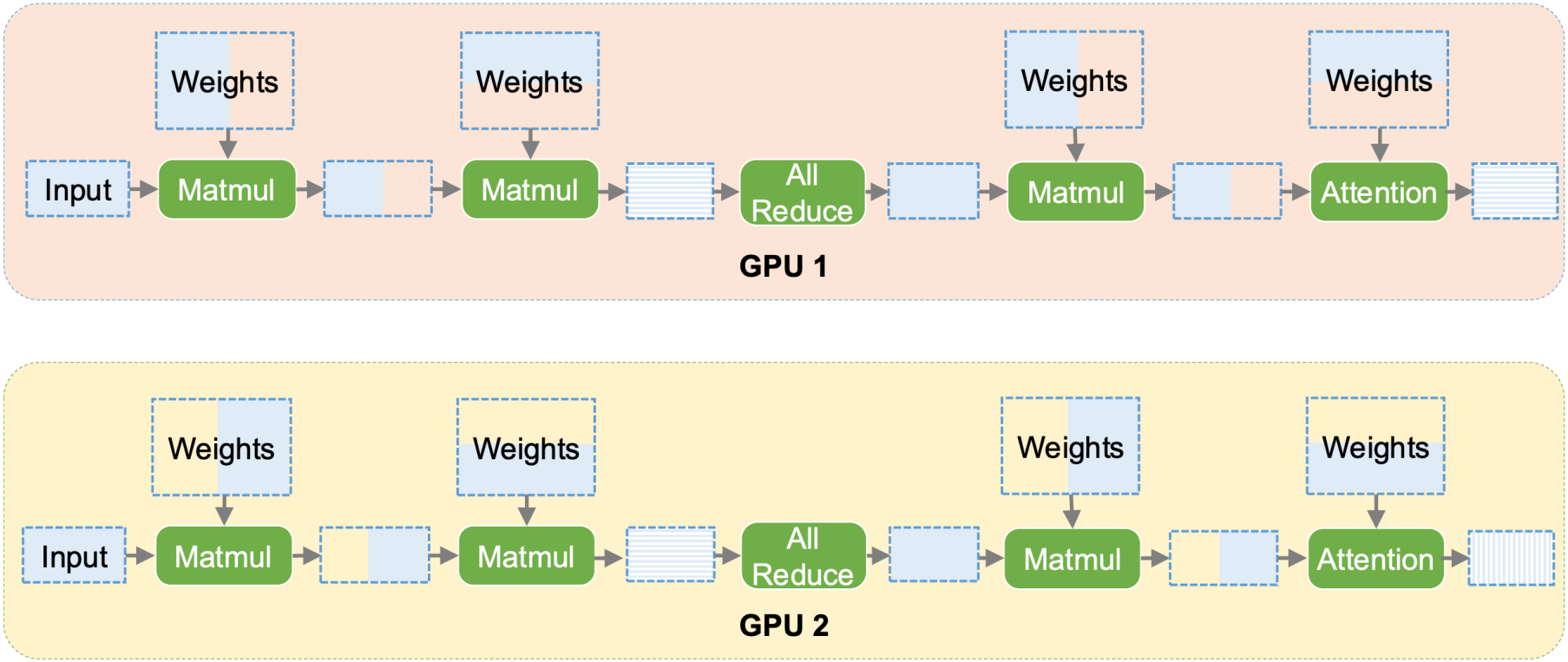


并行机会 PARALLELIZATION OPPORTUNITY



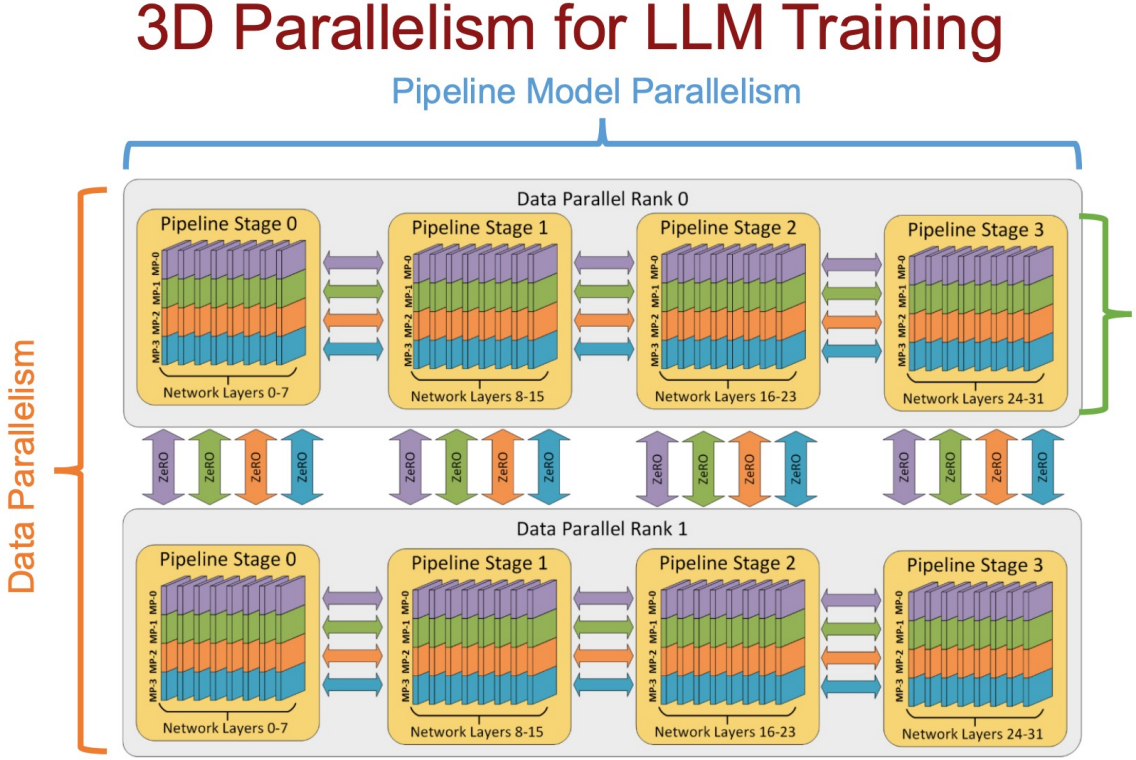
怎样把任务切分到不同计算单元?

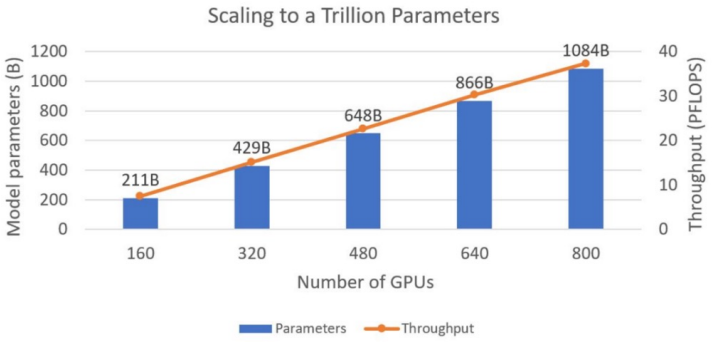
Tensor Model Parallelism for Transformer



并行机会 PARALLELIZATION OPPORTUNITY

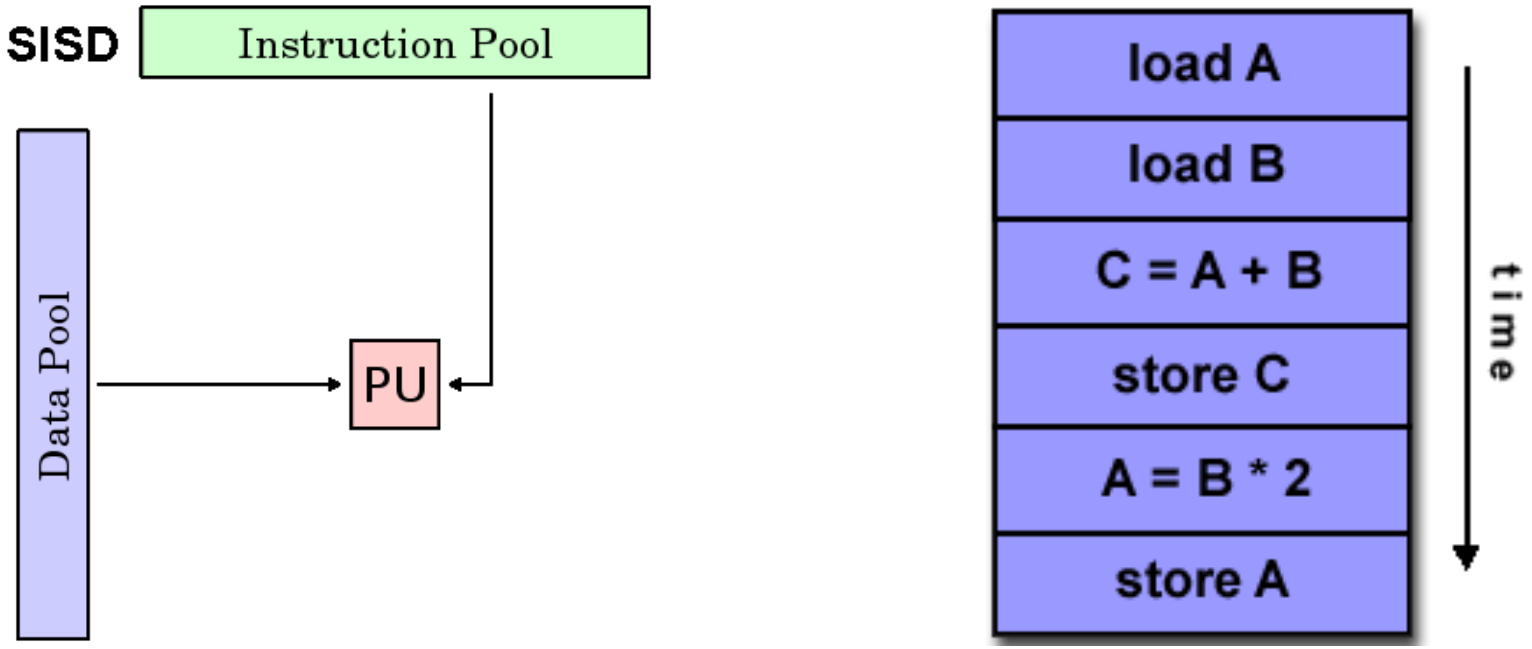
怎样把任务切分到不同计算单元?





并行计算机 PARALLEL COMPUTER

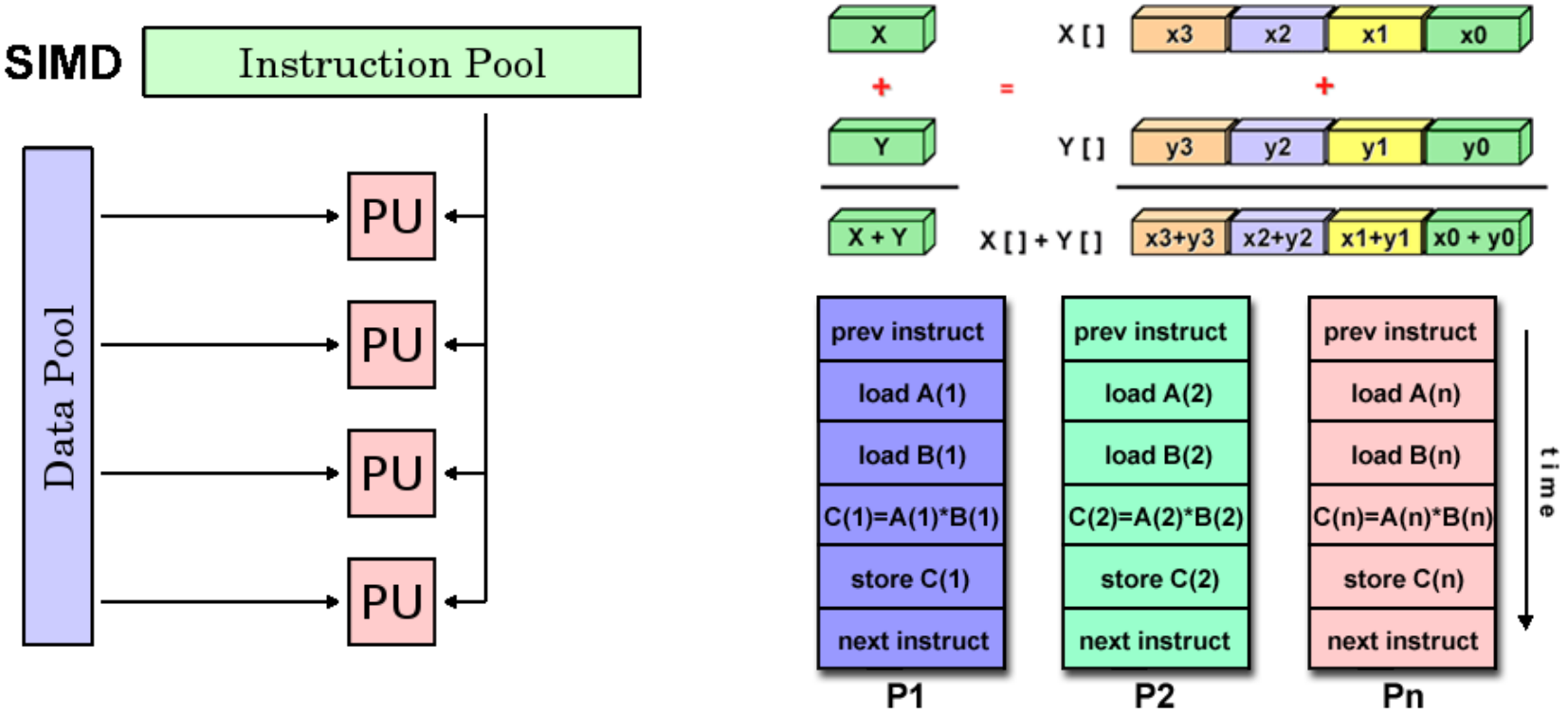
如何做并行计算？指令与数据 (by Flynn)



Single Instruction Single Data

并行计算机 PARALLEL COMPUTER

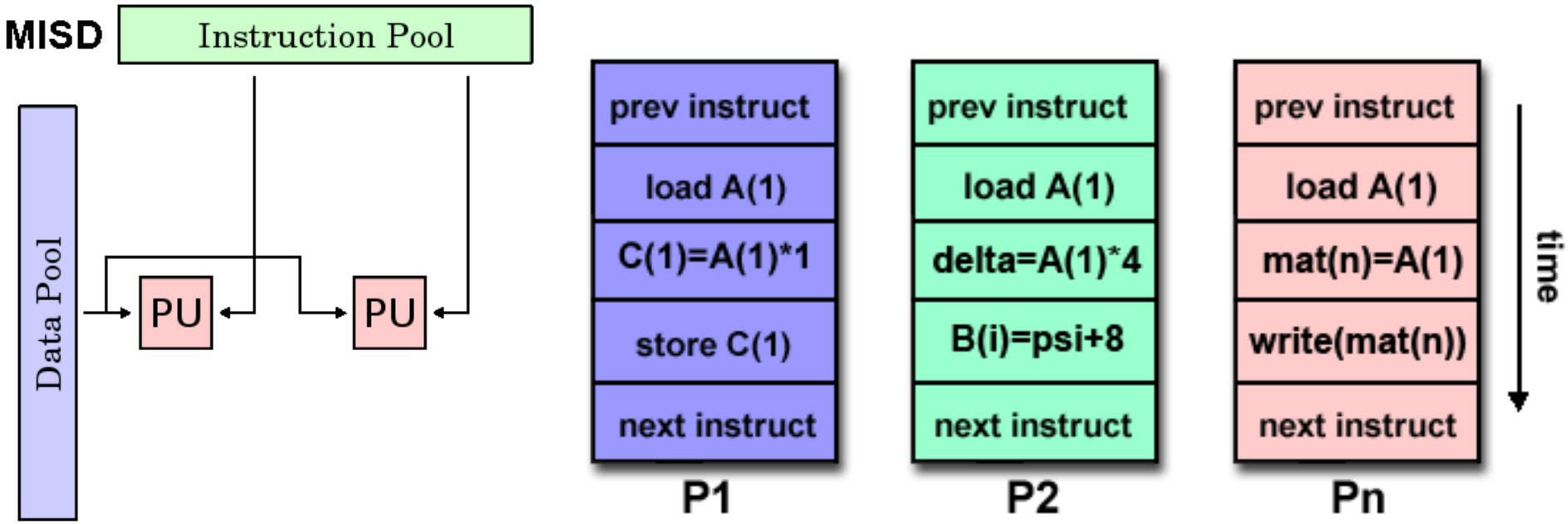
如何做并行计算？指令与数据 (by Flynn)



Single Instruction Multiple Data: 昇腾 NPU

并行计算机 PARALLEL COMPUTER

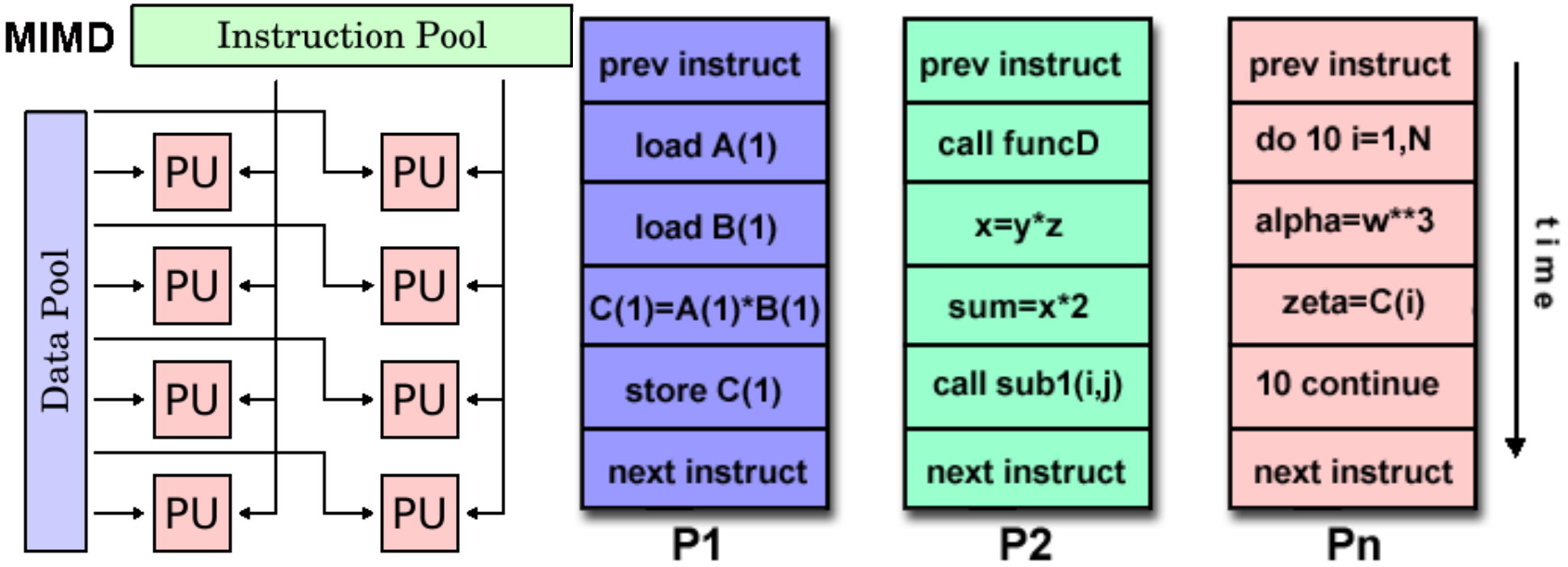
如何做并行计算？指令与数据 (by Flynn)



Multiple Instruction Single Data

并行计算机 PARALLEL COMPUTER

如何做并行计算？指令与数据 (by Flynn)

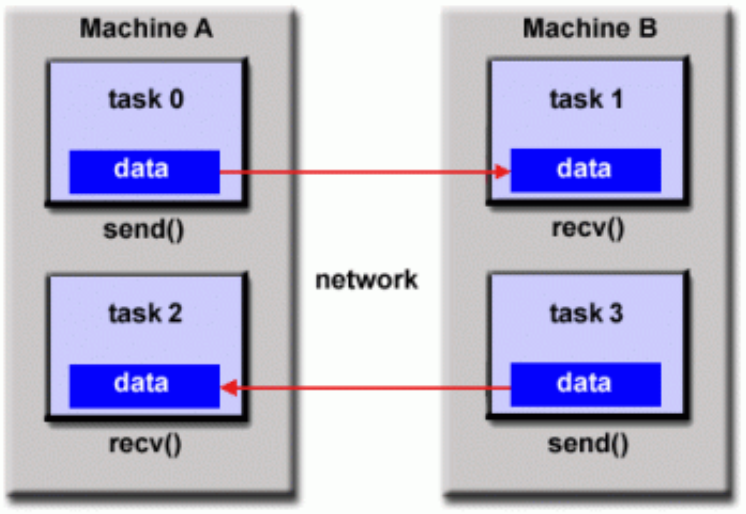


Multiple Instruction Multiple Data: 现代超算、CPU、(GPU)

并行编程 PARALLEL PROGRAMMING

如何做并行计算？并行编程模型 (by Programmers)

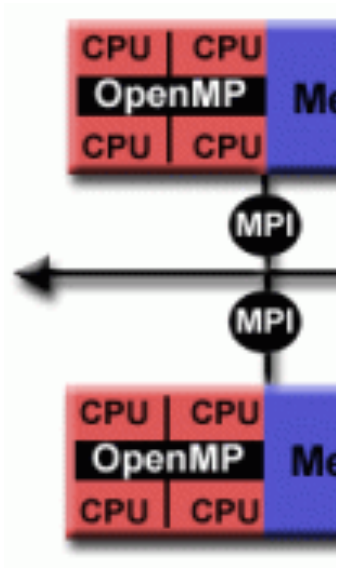
- Shared Memory (without threads)
- Threads
- Distributed Memory / Message Passing
- Data Parallel / Partitioned Global Address Space
- Hybrid
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

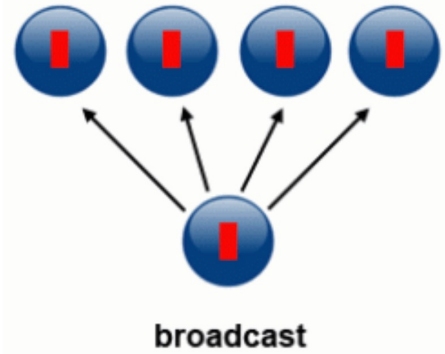


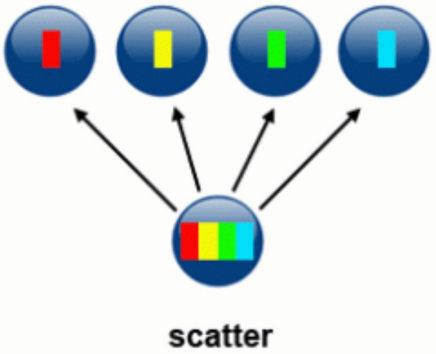
用计算机进行计算，用程序描述计算

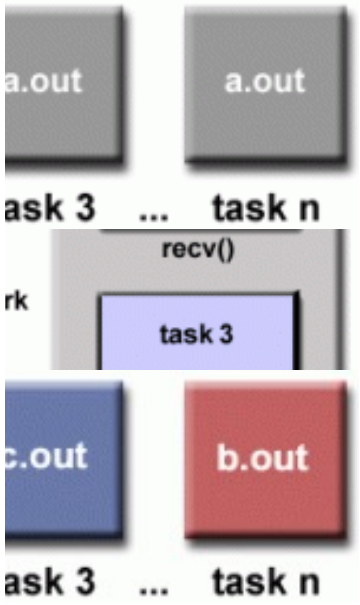
并行编程 PARALLEL PROGRAMMING

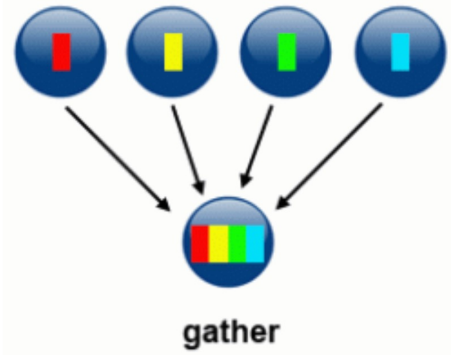
如何做并行计算？并行编程模型 (by Programmers)

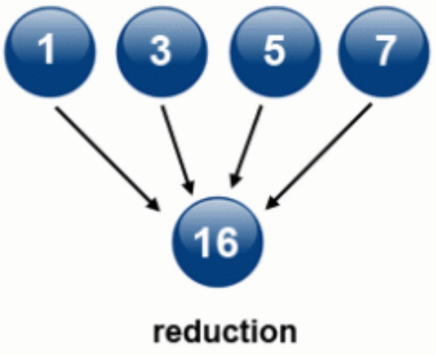






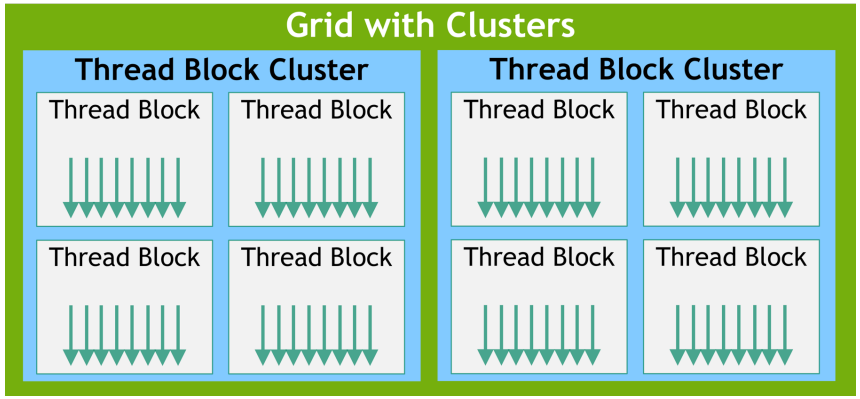




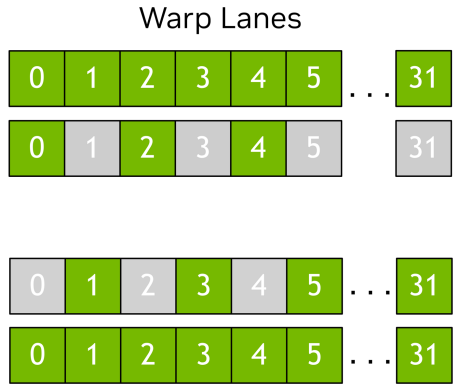


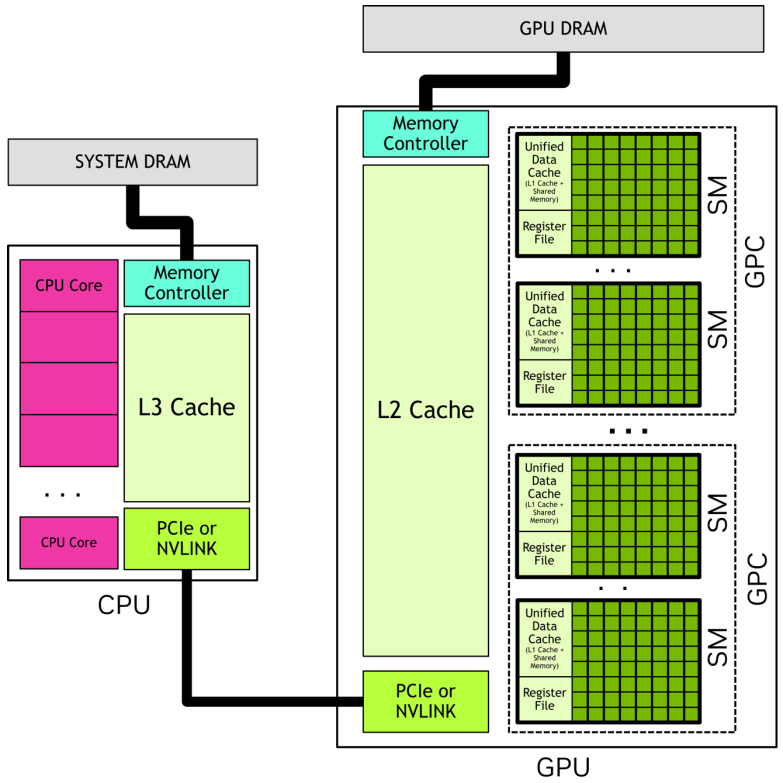
并行编程 PARALLEL PROGRAMMING

如何做并行计算？"SIMT" 模型 & CUDA (by NVIDIA)




```
if(threadIdx.x%2 == 0)
   {a = r(t); }
else
   {a = q(t); }
y = f(a);
```

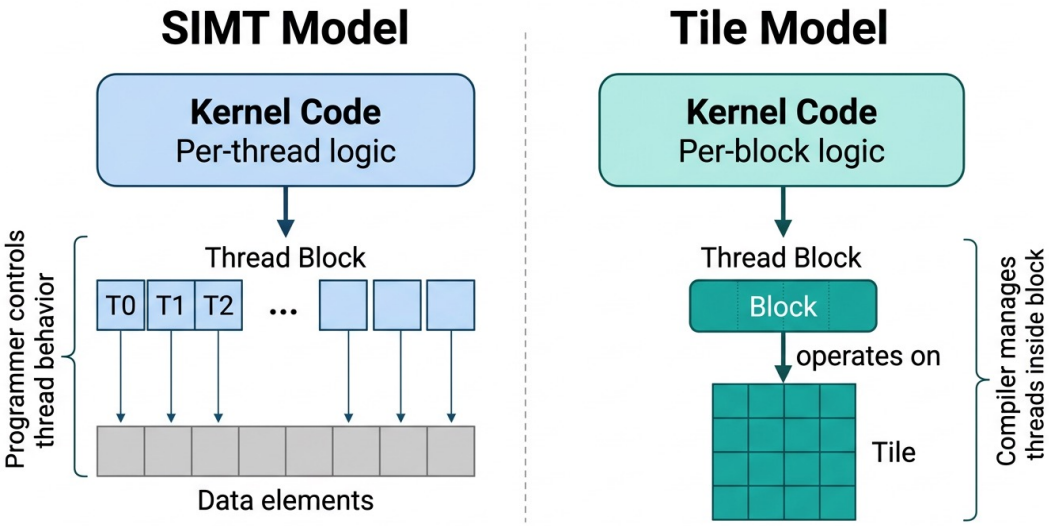




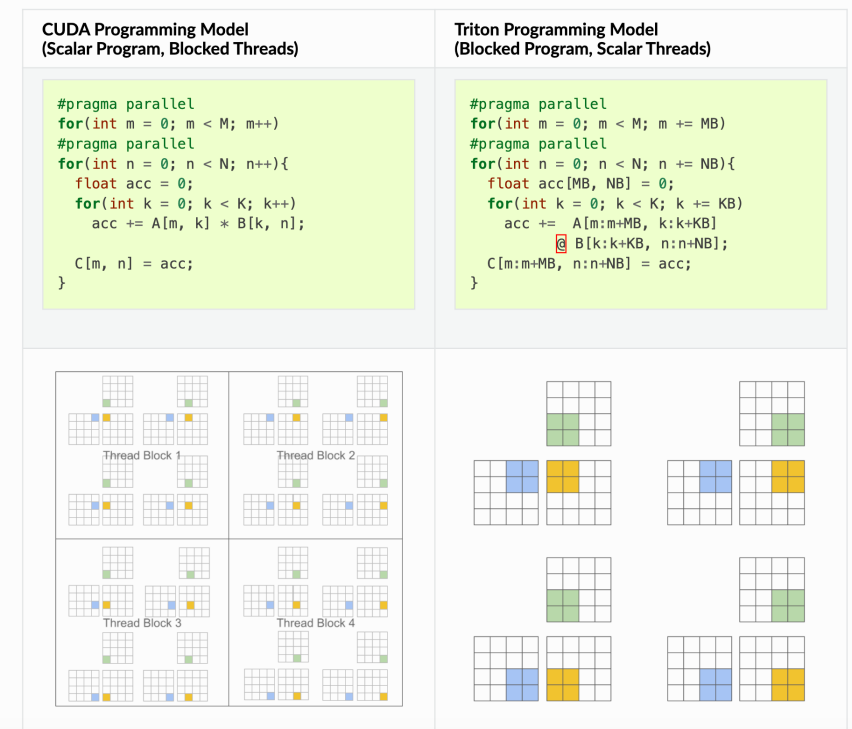
用计算机进行计算，用程序描述计算

并行编程 PARALLEL PROGRAMMING

如何做并行计算？Tile 模型 & Block 编程 (by DSLs)



***SIMT**: programmer thinks in threads. **Tile**: programmer thinks in blocks and tiles.*



用计算机进行计算，用程序描述计算

并行编程 PARALLEL PROGRAMMING

如何做并行计算？Perspectives (by Yourself)

第一性原理、自上而下和自下而上

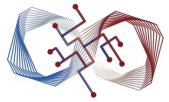
- 关注数据和指令：体系结构 / 硬件 视角 -> SIMD，(SIMT)
- 关注如何描述计算：编程模型 -> SPMD (MPI, OpenMP), (SIMT), (Tile)
- SIMT 更自下而上?

  “组成”式并行 Thread Level Programming
- Tilelang/DSLs 更自上而下?

  “拆分”式并行 (Tile? CTA?) / Block Level Programming

用计算机进行计算，用程序描述计算

并行编程 PARALLEL PROGRAMMING



如何做并行计算?

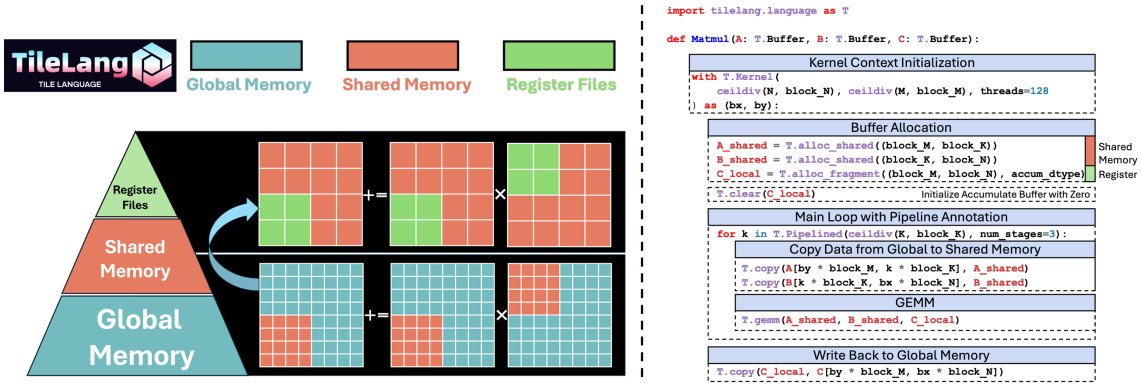
第一性原理、自上而下和自下而上：数据和计算

好了，去研究有趣的并行计算吧！

Perspective Yourself

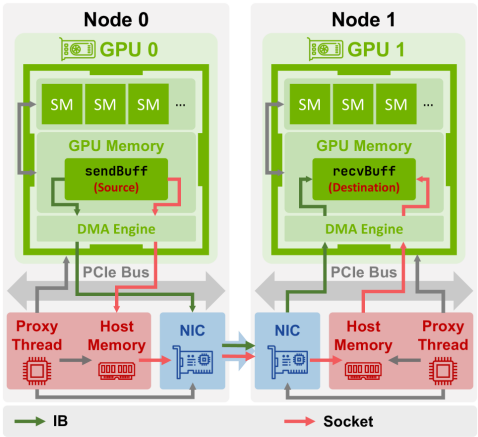
用计算机进行计算，用程序描述计算

AI 中的并行 EVERYTING PARALLEL

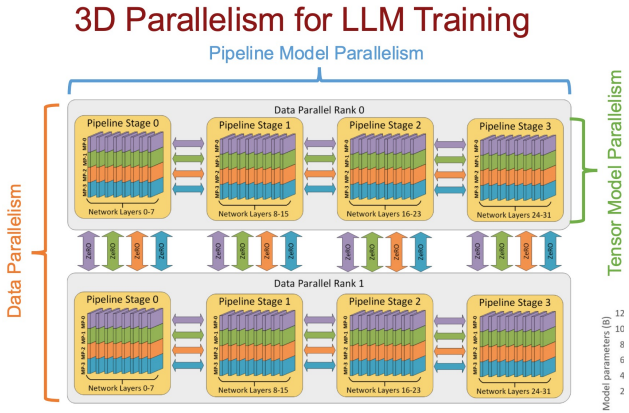


(a) Efficient GEMM with Multi-Level Tiling on GPUs (b) Describing Tiled GPU GEMM with TileLang

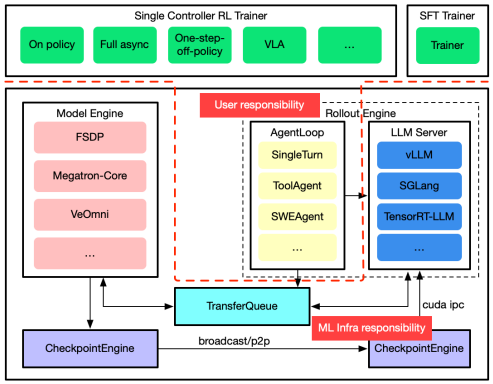
算子



集群通信



分布式训练



强化学习

SGL

LLM

推理框架

好了，去研究有趣的 AI Infra 吧!

参考资料 REFERENCE

- Introduction to Parallel Computing Tutorial
- CUDA Programming Guide
- CMU 15-779 03-CUDA-programming.pdf

1.1 CUDA Programming Model

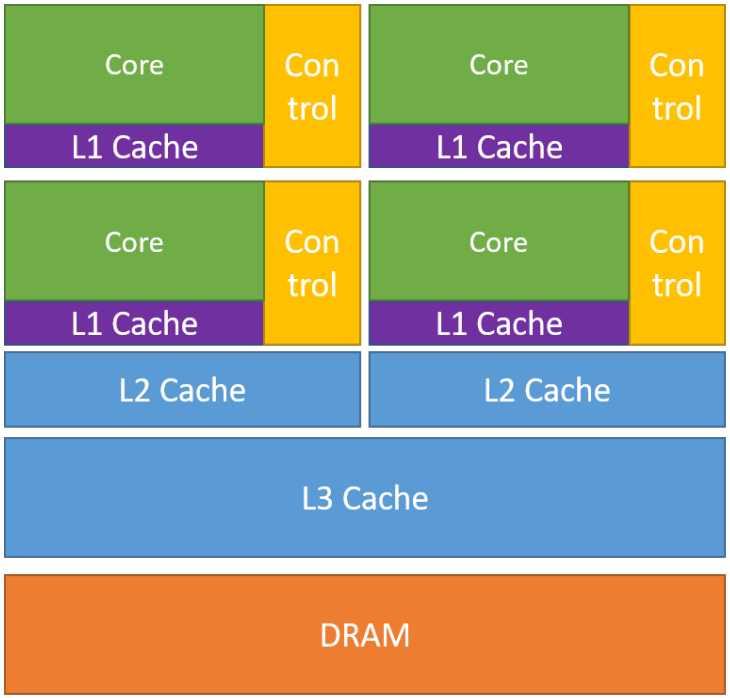
positive1
2026. 07. 25

Weiming Supercomputing Team
Linux Club of Peking University

Linux俱乐部项目

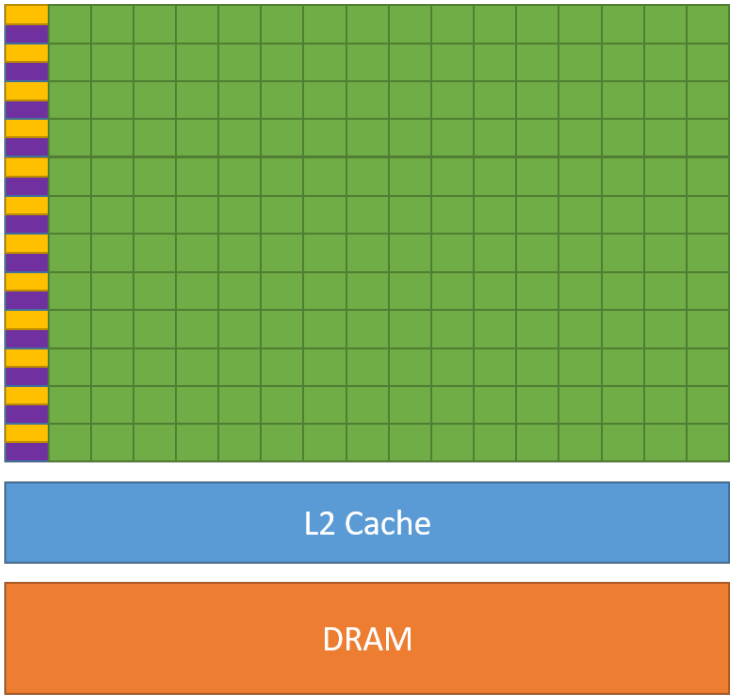
为什么要用 GPU?

GPU 一胜



CPU

大量控制，少量计算



GPU

少量控制，大量计算

为什么要用 GPU？

GPU 二胜

int32 加法，CPU 延迟为 1 个时钟周期，GPU 延迟为 4 个时钟周期

概念	含义	GPU
延迟	完成一个任务从开始到结束所需的时间	😭
吞吐量	单位时间内能完成的任务数量	😊

GPU 标称的算力是吞吐量。从硬件厂商的角度说，提高吞吐量比降低延迟容易。GPU 不擅长降低延迟，但它非常擅长隐藏延迟。GPU 的设计思路是：“既然延迟降不下来，那我就同时处理足够多的任务，让计算单元在等待数据的时候总有别的活干。”

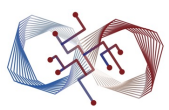
为什么要用 GPU？

GPU 三胜

GPU 的主存通常是 HBM（High Bandwidth Memory）。一条 HBM3E 的带宽可以达到 DDR5 的 20 倍。

要达到 HBM 标称的访问速度，必须做地址连续的大块访问（coalesced access）。随机小粒度访问会浪费带宽。

“N 方过百万”实现了吗?



递推式	GPU
$a_{n+1} = a_n/(1+\sqrt{a_n})$	😭
$a_{n,m} = a_{n-1,m} + a_{n-1,m-1}$	😕
$a_n = \sum_{i<n} 1/(a_i+n)$	😊

CUDA 是什么?

In 2006, NVIDIA introduced the *Compute Unified Device Architecture* (CUDA) to enable any computational workload to use the throughput capability of GPUs independent of graphics APIs.

核心概念

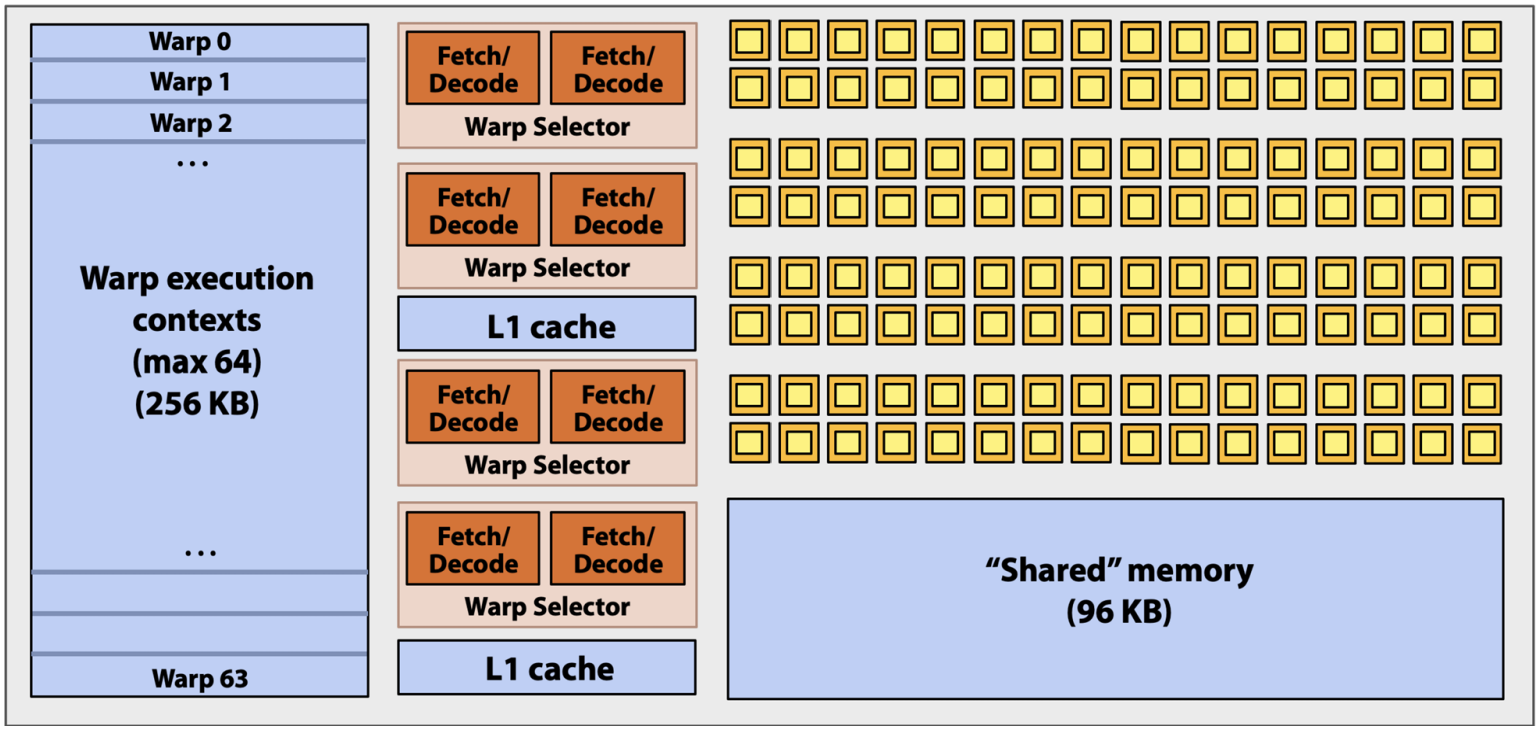
$$\text{thread} \subset \text{warp} \subset \text{block} \subset \text{SM} \subset \text{grid}$$

- thread 是程序运行的基本单元
- 1 个 warp 是 32 个 thread，这 32 个 thread 执行相同的指令
- block 又叫 CTA（Cooperative Thread Array），是线程间合作的范围

CUDA 是什么?

核心概念 $\text{thread} \subset \text{warp} \subset \text{block} \subset \text{SM} \subset \text{grid}$

- SM（Streaming Multiprocessor）是执行 block/warp 的硬件

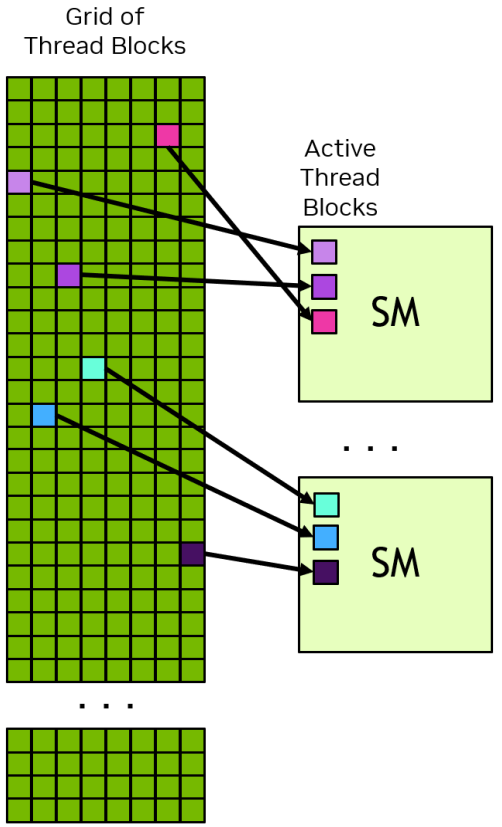


CUDA 是什么?

核心概念

$$\text{thread} \subset \text{warp} \subset \text{block} \subset \text{SM} \subset \text{grid}$$

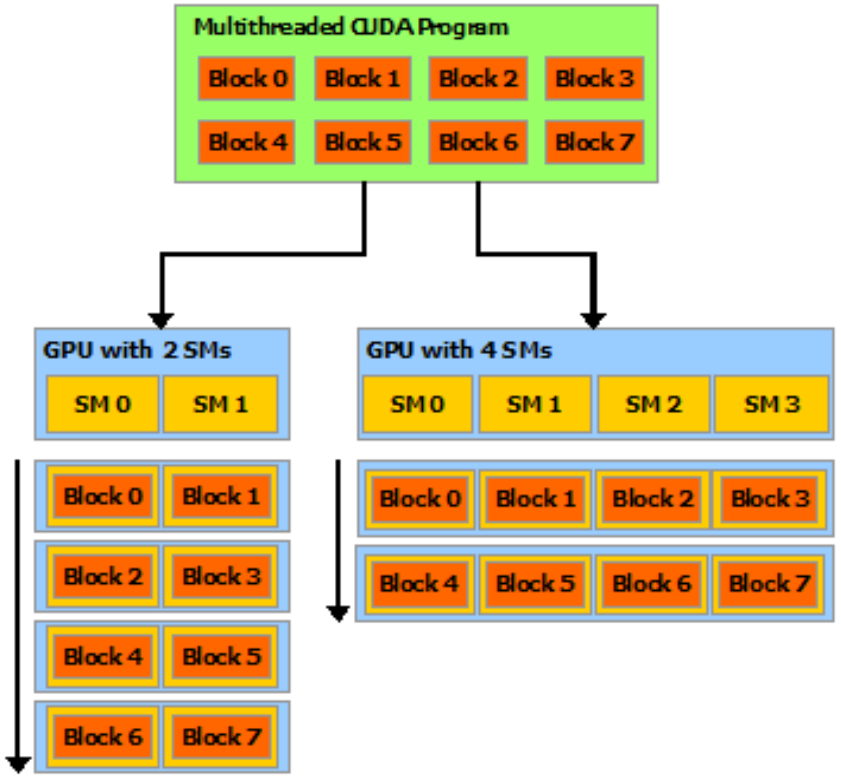
- grid 是 block 的集合
- block 数量可以远多于 SM 数量
- block 的调度顺序是未定义的



CUDA 是什么?

自动可扩展性

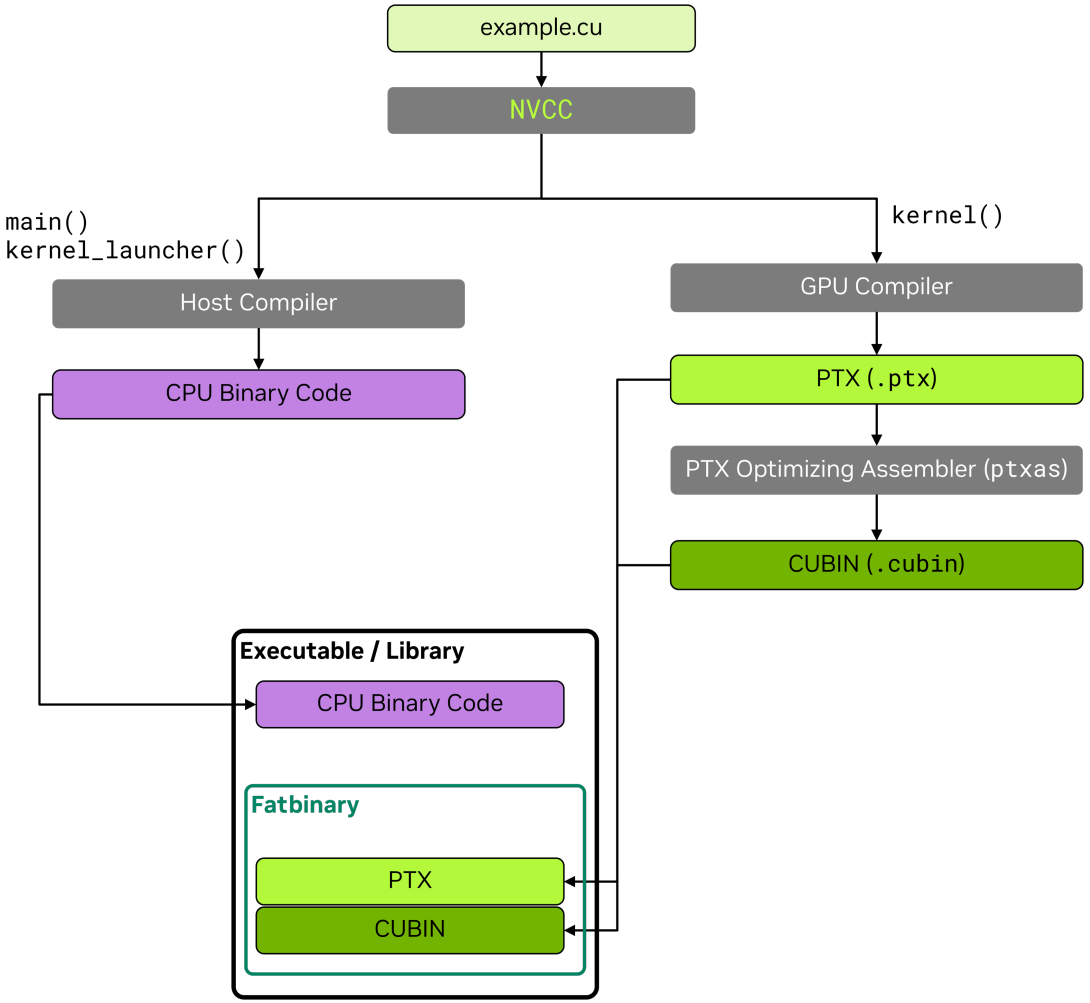
- 每个 block 都是可以独立执行的程序，block 之间没有数据依赖
- block 可以按任意顺序调度到任意多的 SM 上，程序（员）不用关心总共有几个 SM
- 不需要重新编译，不需要改代码



CUDA 是什么?

编译原理

- CUDA 扩展了 C/C++ 的语法，代码要保存为 `.cu` 文件，编译器要用 `nvcc`
- PTX（Parallel Thread eXecution）是“高级”汇编，后向兼容性较好
- CUBIN（CUda BINary）是 GPU 的“可执行文件”



CUDA 是什么?

工作流程概览

1.在 CPU 上准备数据

2.将数据从 CPU 内存拷贝到 GPU 内存

3.配置 kernel 启动参数（`<<<grid, block>>>`）

4.在 GPU 上并行执行 kernel

5.将结果从 GPU 内存拷回 CPU 内存

Kernel：在 GPU 上运行的函数

新的关键字

参考：Execution Space Specifier

CPU 是 host，GPU 是 device。

- `__host__`: 这类函数与正常的函数没有区别。其只能被 host 上执行的函数（`__host__`）调用，并在 host 上执行。
- `__global__`: 这类函数可以被任何函数调用，并在 device 上执行。
- `__device__`: 这类函数只能被 device 上执行的函数（`__device__` 或 `__global__`）调用，并在 device 上执行。

Kernel：在 GPU 上运行的函数

定义 Kernel

Kernel 应当用 `__global__` 修饰，返回类型必须是 `void`。以下代码产生 N 个 CUDA thread 并行计算向量加法。

```
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    VecAdd<<<1, N>>>(A, B, C);
}
```

Kernel：在 GPU 上运行的函数

内存管理 API

```
int main()
{
    int n = 1000000;
    size_t bytes = n * sizeof(float);
    float *h_a = (float*)malloc(bytes);
    for (int i = 0; i < n; i++) h_a[i] = 1.0f;
    float *d_a;
    cudaMalloc(&d_a, bytes);
    cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(h_a, d_a, bytes, cudaMemcpyDeviceToHost);
    cudaFree(d_a);
    free(h_a);
}
```

Kernel：在 GPU 上运行的函数

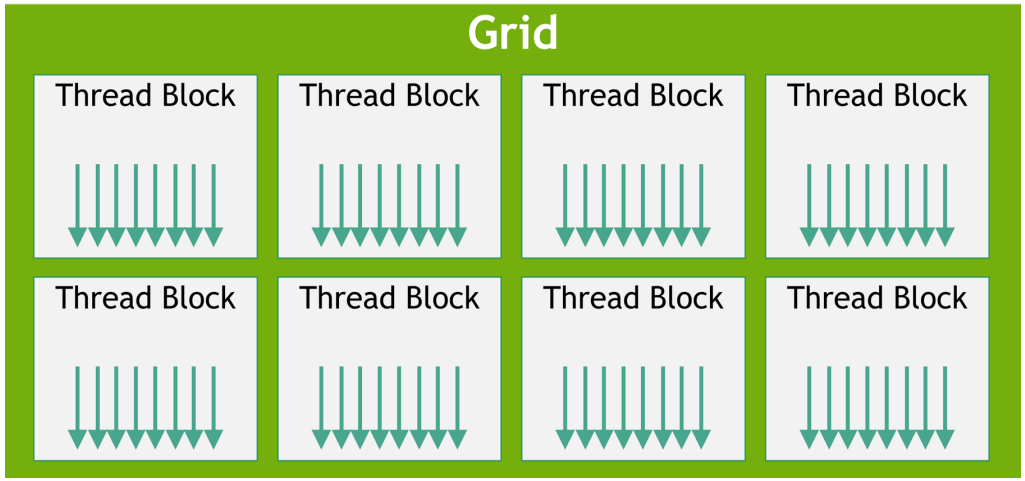
Kernel 启动参数

`<<<gridDim, blockDim>>>`

- grid 的维数就是 block 的数量
- block 的维数就是每个 block 的 thread 数量
- 可以是 `int` 或 `dim3` 类型，以支持 1~3 维的分块

Kernel：在 GPU 上运行的函数

Kernel 中的内置变量



变量	含义
`threadIdx`	当前 thread 在 block 内的下标
`blockIdx`	当前 block 在 grid 内的下标
`blockDim`	block 的维数
`gridDim`	grid 的维数

Kernel：在 GPU 上运行的函数

矩阵加法

```
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
int main()
{
    dim3 threadsPerBlock(N, N);
    MatAdd<<<1, threadsPerBlock>>>(A, B, C);
}
```

Kernel：在 GPU 上运行的函数

使用多个 block

这里加入了边界处理。矩阵大小经常不是块长的整数倍，处理方式就是加个 if。

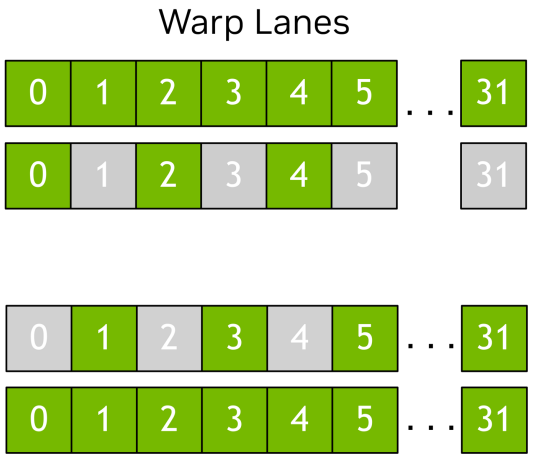
```
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}
int main()
{
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Warp, SIMT 与 SIMD

什么是 warp?

1 个 warp 是 32 个 thread。这 32 个 thread 必须执行相同的指令。如果出现了分支，分支的路径会串行化，走当前路径的线程被激活，未走当前路径的线程被 mask 掉。

```
if(threadIdx.x%2 == 0)
    {a = r(t); }
else
    {a = q(t); }
y = f(a);
```



同一个 block 里的 thread 被分到哪个 warp 由如下的 thread ID 决定：

`ID=threadIdx.x+threadIdx.y*blockDim.x+threadIdx.z*blockDim.x*blockDim.y`

`ID/32`（下取整）相同的线程就在同一个 warp。

SIMT 与 SIMD

SIMD: Single Instruction Multiple **Data**

SIMT: Single Instruction Multiple **Thread**

数据是被动的，线程是主动的。从 CC 7.0 起，每个 thread 有自己的 program counter，也就可以执行不同的指令。而 SIMD 只有一个 program counter。

但是，一个 warp 里每次激活的 thread 应该有相同的 program counter。如果条件分支太多，将导致 program counter 的不同取值太多，那么 SIMT 也是低效的。

最有利于 SIMT 发挥效率的编程方式仍然是 SIMD。

~~你也可以 argue 说 program counter 不本质，因为可以软件模拟，用 gather load 造出一个 SIMD 虚拟机。~~

Warp Divergence 案例

优化方法：尽量使 warp 内的所有线程走相同的分支，即让数据分布与 warp 边界对齐。

```
__global__ void badKernel(float* data, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        if (i % 2) data[i] = sqrt(data[i]);
        else data[i] = exp(data[i]);
    }
}
__global__ void betterKernel(float* data, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int half = (N + 1) / 2;
    if (i < half) {
        int target = 2 * i;
        data[target] = exp(data[target]);
    } else if (i < N) {
        int target = 2 * (i - half) + 1;
        data[target] = sqrt(data[target]);
    }
}
```

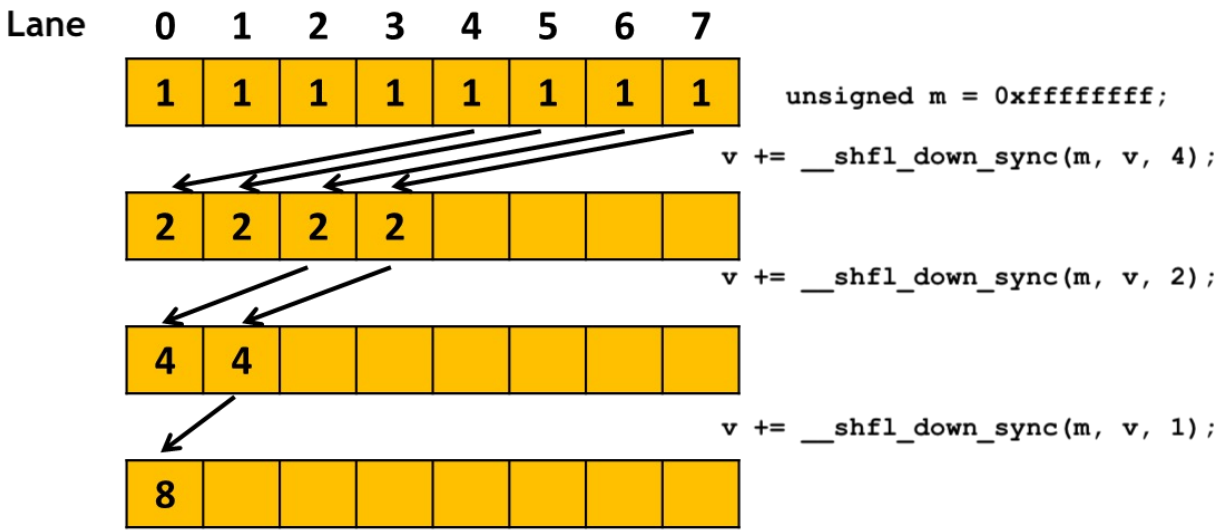
Warp-Level Primitives

Warp 内可以高效进行数据交换和同步。这段代码实现了 reduce 操作。

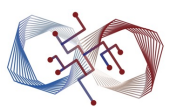
```
#define FULL_MASK 0xffffffff
for (int offset = 16; offset > 0; offset /= 2)
    val += __shfl_down_sync(FULL_MASK, val, offset);
```

操作后 lane 0 的 `val` 是操作前 32 个 lane 的 `val` 之和，其它 lane 的 `val` 未定义。

来源：[Using CUDA Warp-Level Primitives]



Thread Block



Block 内协作

Block 内的 thread 比较类似 CPU 的 thread，可以用 shared memory，也有原子操作、内存屏障、线程同步等指令。

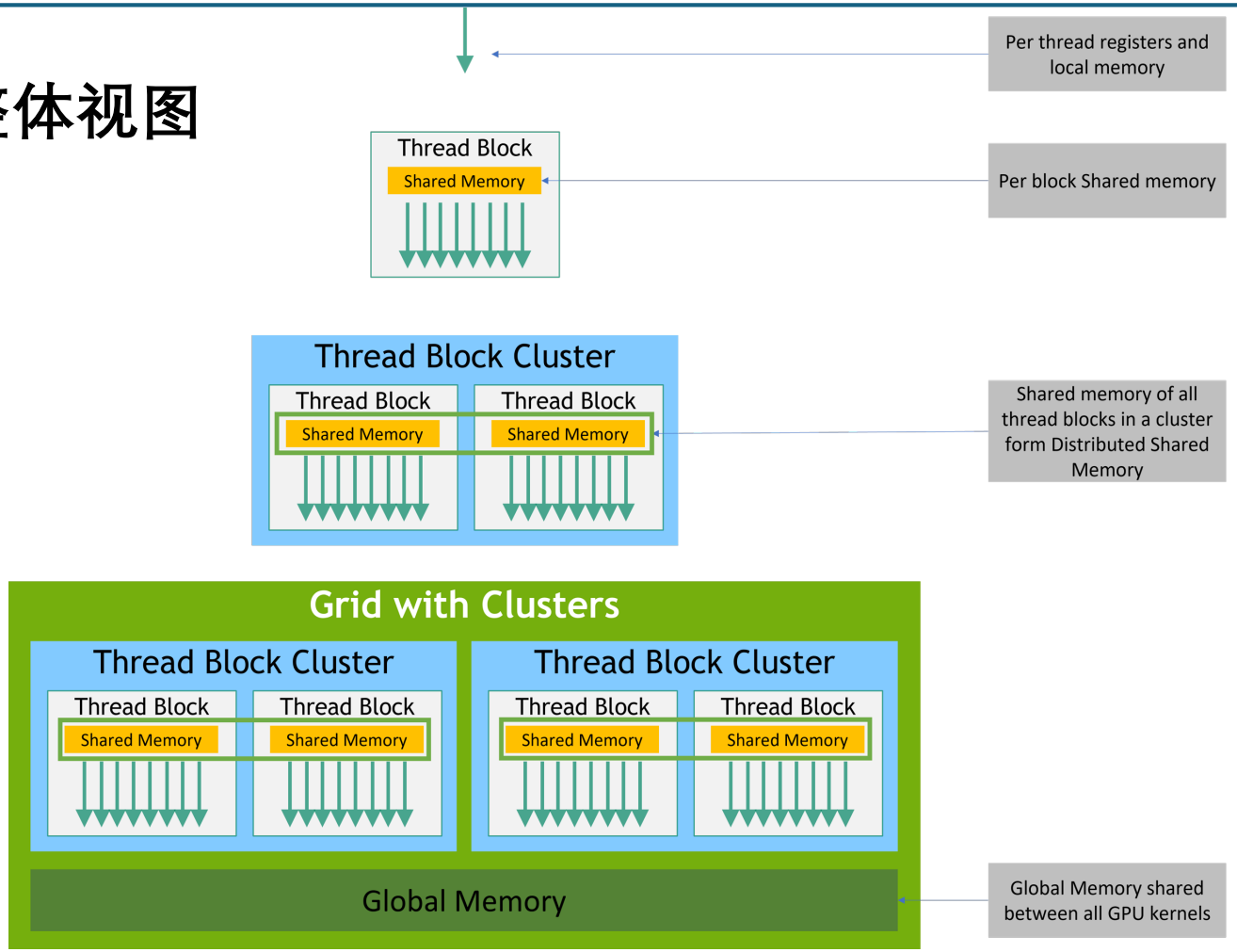
```
// assuming blockDim.x is 128
__global__ void syncthreads_valid_behavior(int* input_data, int* output_data) {
    __shared__ int shared_data[128];
    shared_data[threadIdx.x] = input_data[threadIdx.x];
    if (blockIdx.x > 0) { // CORRECT, uniform condition across all block threads
        __syncthreads();
        output_data[threadIdx.x] = shared_data[127 - threadIdx.x];
    }
}
__global__ void syncthreads_invalid_behavior(int* input_data, int* output_data) {
    __shared__ int shared_data[128];
    shared_data[threadIdx.x] = input_data[threadIdx.x];
    for (int i = 0; i < blockDim.x; ++i) {
        if (i == threadIdx.x) { // WRONG, non-uniform condition
            __syncthreads();    // Undefined Behavior
        }
    }
    output_data[threadIdx.x] = shared_data[127 - threadIdx.x];
}
```

Thread Block Cluster

从 CC 9.0 开始，CUDA 引入了一个可选的中间层次：Thread Block Cluster。Cluster 中的 block 保证在同一个 GPC（Graphics Processing Cluster）上调度，可以实现跨 block 通信。

内存层次结构

整体视图


Per thread registers and local memory
Thread Block
Shared Memory
Per block Shared memory
Thread Block Cluster
Thread Block
Shared Memory
Thread Block
Shared Memory
Shared memory of all thread blocks in a cluster form Distributed Shared Memory
Grid with Clusters
Thread Block Cluster
Thread Block
Shared Memory
Thread Block
Shared Memory
Thread Block Cluster
Thread Block
Shared Memory
Thread Block
Shared Memory
Global Memory
Global Memory shared between all GPU kernels

内存层次结构

各层内存一览

来源：2.3. Writing SIMT Kernels — CUDA Programming Guide

Memory Type	Scope	Lifetime	Location
Global	Grid	Application	Device
Constant	Grid	Application	Device
Shared	Block	Kernel	SM
Local	Thread	Kernel	Device
Register	Thread	Kernel	SM

内存层次结构

Shared Memory

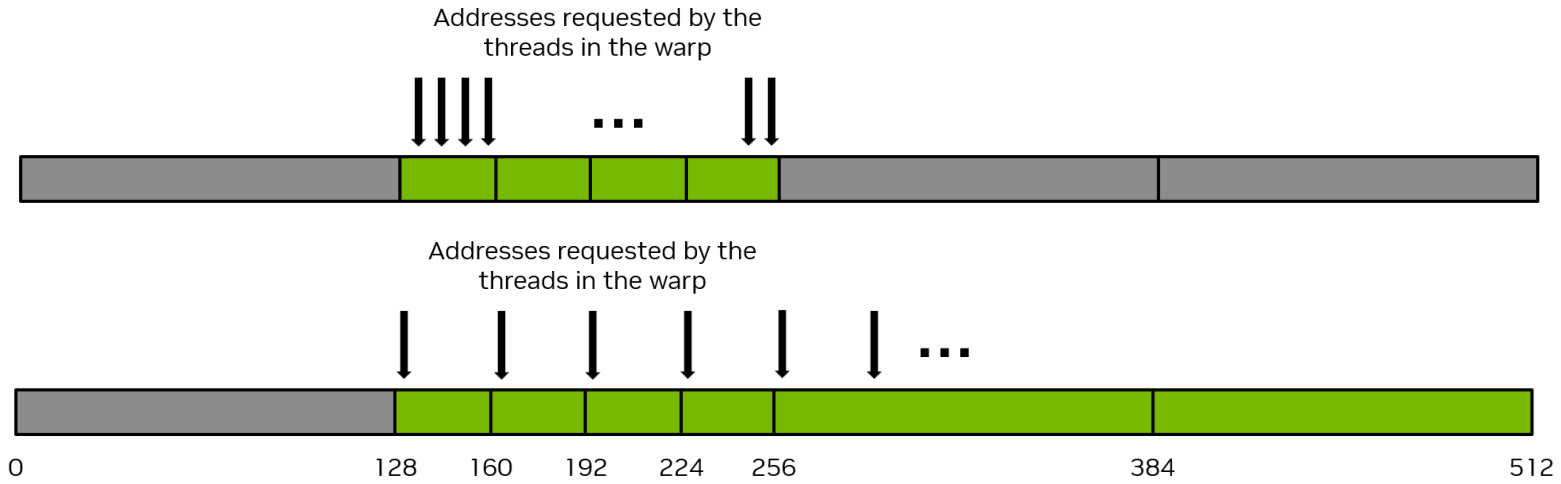
Shared Memory 与 L1 Cache 是同一块物理存储器的不同划分，可配置比例。以下代码为使用 shared memory 的分块矩阵乘法。来源：CUDA-From-Correctness-To-Performance-Code/gemm_gpu_tiling.cu

- `__restrict__`
- `__share__`
- x y 转置

```
constexpr int TILE_SIZE = 32;
__global__ void gemm_gpu_tiling_kernel(
    int* __restrict__ C,        // [n, m], on gpu
    const int* __restrict__ A,  // [n, k], on gpu
    const int* __restrict__ B,  // [k, m], on gpu
    int n, int m, int k         // multiples of TILE_SIZE
) {
    __shared__ int a_tile[TILE_SIZE][TILE_SIZE];
    __shared__ int b_tile[TILE_SIZE][TILE_SIZE];
    int my_c_result = 0;
    for (int tile_index = 0; tile_index < k/TILE_SIZE; ++tile_index) {
        a_tile[threadIdx.y][threadIdx.x] = A[(blockIdx.x*TILE_SIZE +
threadIdx.y)*k + (tile_index*TILE_SIZE + threadIdx.x)];
        b_tile[threadIdx.y][threadIdx.x] = B[(tile_index*TILE_SIZE +
threadIdx.y)*m + (blockIdx.y*TILE_SIZE + threadIdx.x)];
        __syncthreads();
        for (int i = 0; i < TILE_SIZE; ++i) {
            my_c_result += a_tile[threadIdx.y][i] * b_tile[i][threadIdx.x];
        }
        __syncthreads();
    }
    C[(blockIdx.x*TILE_SIZE + threadIdx.y)*m + (blockIdx.y*TILE_SIZE +
threadIdx.x)] = my_c_result;
}
```

全局内存访问的 coalescing

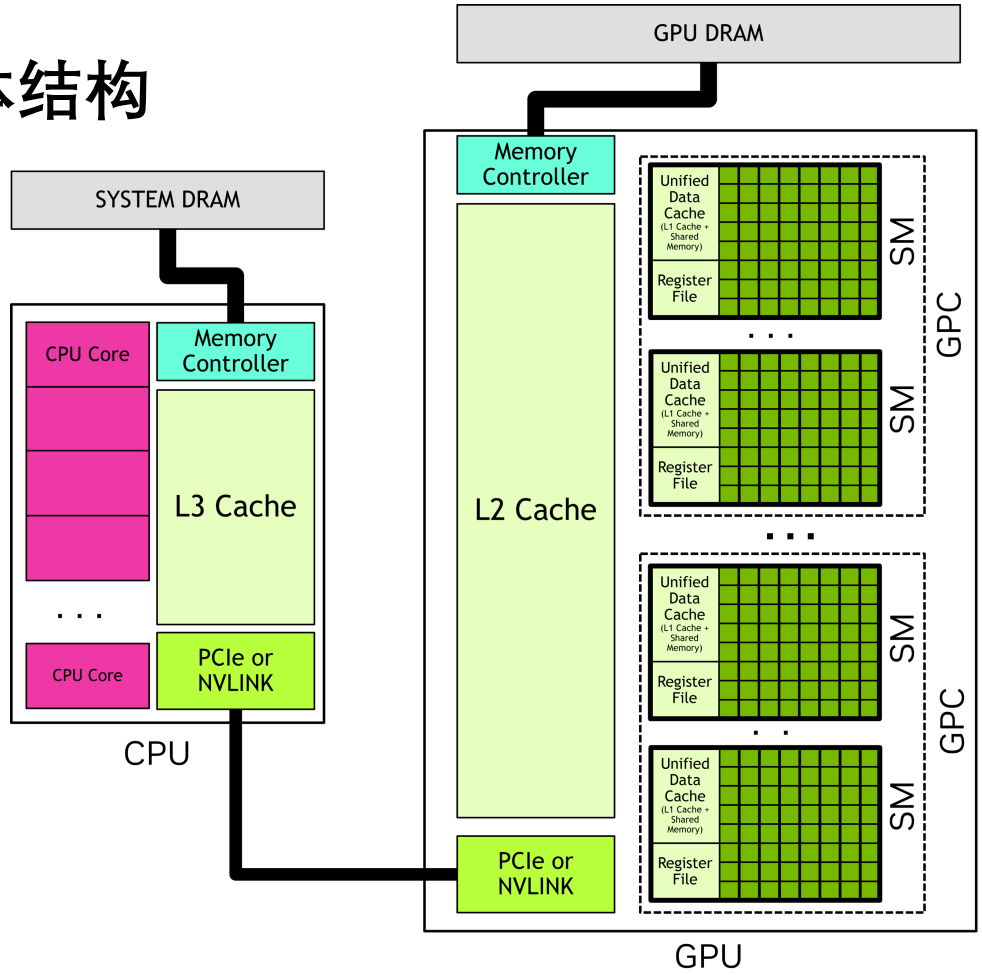
全局内存的访问效率取决于访问模式。GPU 会将 warp 中 32 个线程的访存请求合并为尽量少的 transactions。一次 transaction 是 32 字节，起始地址是 32 字节的倍数。



这有点像 CPU 的缓存行机制，不过 GPU 自己的缓存行还要复杂一点：L1 缓存行 128 字节，L2 缓存行 32 字节。

GPU 的硬件实现

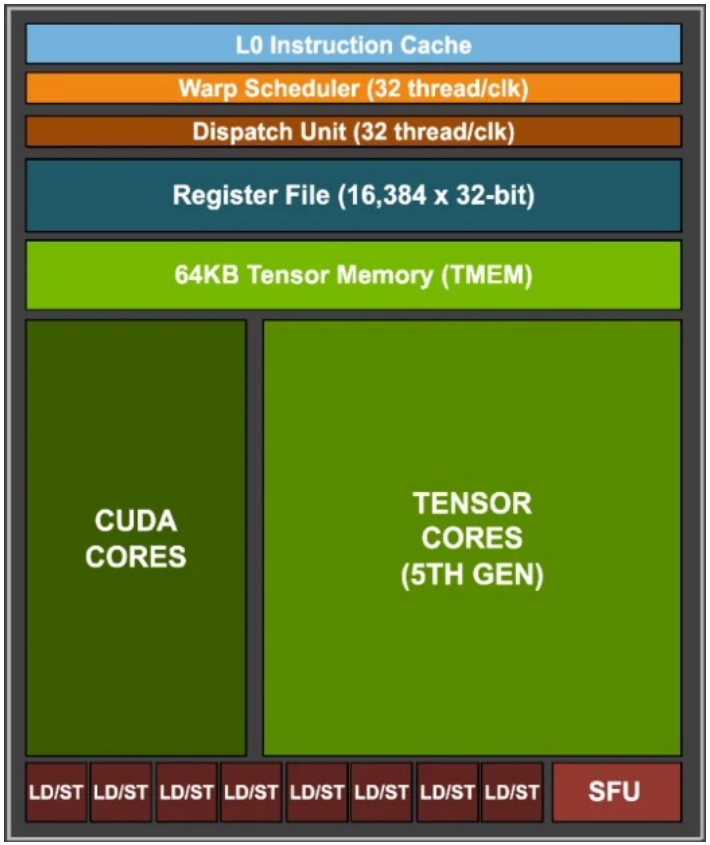
GPU 的整体结构



GPU 的硬件实现

SM 的内部结构——以 Blackwell Ultra 为例

- SFU：Special Function Units（**近似** 倒数，平方根倒数,sin，……）
- Tex：Texture
- 共 128 个 CUDA Cores 用于 32 位及更低精度计算



GPU 的硬件实现

Warp 调度

问题：CUDA Core 的运算速度极快，但 Global Memory 的延迟是几百个周期。如果 warp 发了一个访存请求后就干等着，计算单元就空闲了。

解决方案：Warp 调度器维护着比计算单元多得多的 warp。

- 当一个 warp 因为访存被阻塞时，调度器立即切换到另一个可执行的 warp
- 硬件维护了所有 warp 的寄存器状态，上下文切换开销很低
- 这使得计算单元和内存带宽都能尽量保持满负载
- 用吞吐量掩盖延迟

```
时间轴 →
warp 0:  [计算] [ 等待内存数据 ]      [计算]
warp 1:         [计算] [ 等待内存数据 ]
warp 2:                [计算] [ 等待内存数据 ]
warp 3:                       [计算] [ 等待...]
```

Occupancy

SM 的资源是有限的

考虑一个 CC 10.0 的 SM。来源：2.3. Writing SIMT Kernels — CUDA Programming Guide

Occupancy = 实际活动 warp 数 / 最大支持 warp 数

高 occupancy 有助于隐藏延迟。

Since register pressure is a critical issue in many CUDA codes, the use of restricted pointers can negatively impact performance by reducing occupancy.

Resource	Value
`maxBlocksPerMultiProcessor`	32
`sharedMemPerMultiprocessor`	233472
`regsPerMultiprocessor`	65536
`maxThreadsPerMultiProcessor`	2048
`sharedMemPerBlock`	49152
`regsPerBlock`	65536
`maxThreadsPerBlock`	1024

Occupancy

科学调块长

```
int minGridSize, blockSize;
cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize, myKernel, 0, 0);
int numBlocks;
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocks, myKernel, blockSize, 0);
```

优化方向

- **最大化并行执行**：足够的 thread/block 来利用全部 SM
- **优化内存访问**：合并访问（coalescing）、使用 shared memory
- **最小化 warp divergence**：保持 warp 内控制流一致
- **利用专用硬件**：Tensor Core

错误检查

```
cudaError_t err = cudaMalloc(&d_A, N * sizeof(float));
if (err != cudaSuccess) {
    fprintf(stderr, "CUDA error: %s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
```

```
#define CUDA_CHECK(expr_to_check) do {            \
    cudaError_t result  = expr_to_check;          \
    if(result != cudaSuccess)                     \
    {                                             \
        fprintf(stderr,                           \
                "CUDA Runtime Error: %s:%i:%d = %s\n", \
                __FILE__,                         \
                __LINE__,                         \
                result,\
                cudaGetErrorString(result));      \
    }                                             \
} while(0)
```

事件计时

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
myKernel<<<grid, block>>>(d_A, d_B, d_C);
cudaEventRecord(stop);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
printf("Kernel time: %f ms\n", milliseconds);

cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Stream

CUDA Stream 是一种命令队列。不同 Stream 的 kernel 可以并发执行，只要 SM 有资源。

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

kernel1<<<grid, block, 0, stream1>>>(...);
kernel2<<<grid, block, 0, stream2>>>(...);  // 可能与 kernel1 并发
// 第三个启动参数是动态 shared memory 大小
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);

cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```

统一内存

通过 cudaMallocManaged 分配统一内存，CPU 和 GPU 自动共享数据，无需手动 cudaMemcpy。

也可以用 __managed__ 修饰变量，但存在一些限制，详见 5.4. C/C++ Language Extensions — CUDA Programming Guide。

```
float *x;
cudaMallocManaged(&x, N * sizeof(float));
cudaFree(x);
```

多卡编程

```
cudaSetDevice(0);                    // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);               // Allocate memory on device 0

cudaSetDevice(1);                    // Set device 1 as current
float* p1;
cudaMalloc(&p1, size);               // Allocate memory on device 1

cudaSetDevice(0);                    // Set device 0 as current
MyKernel<<<1000, 128>>>(p0);         // Launch kernel on device 0

cudaSetDevice(1);                    // Set device 1 as current
cudaMemcpyPeer(p1, 1, p0, 0, size); // Copy p0 to p1
MyKernel<<<1000, 128>>>(p1);         // Launch kernel on device 1
```

多卡编程

```
cudaSetDevice(0);                    // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);               // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0);         // Launch kernel on device 0

cudaSetDevice(1);                    // Set device 1 as current
cudaDeviceEnablePeerAccess(0, 0);    // Enable peer-to-peer access
                                     // with device 0

// Launch kernel on device 1
// This kernel launch can access memory on device 0 at address p0
MyKernel<<<1000, 128>>>(p0);
```

参考资料

- CUDA-From-Correctness-To-Performance-Code/lecture.md
- CUDA Programming Guide
- CUDA C++ Programming Guide (Legacy)
- CUDA Best Practices Guide
- CMU 15-779 03-CUDA-programming.pdf

1.3 Triton / TileLang

CTA Level Programming

yxcatqwq
2026. 07. 25

Weiming Supercomputing Team
Linux Club of Peking University

Linux俱乐部项目

Review: 关于 cuda kernel 映射到 GPU

任务划分

- 一个 Kernel 启动生成一个 Grid
- Grid 被划分为多个独立调度的 Block / CTA
- 每个 Block 内包含多个 Thread，并以 Warp 为单位执行

数据存放

- Global Memory：容量大、延迟高，所有 Block 可访问
- Shared Memory：Block 内共享，用于数据复用
- Register：Thread 私有，访问最快

需要在代码中手动指定：

- 每个 Block 和 Thread 的分工情况
- 数据如何在 Global Memory、Shared Memory 和 Register 之间搬运
- 手动完成从“数据分块”到“线程执行”的映射

为什么需要 Tile-Level Programming?

SIMT 的问题

- 线程协作、数据复用、shared memory 都需要手动管理
- 写起来繁琐、容易出错、难以优化

Tile-Level

- 把 block 内线程的协作抽象成分块(tile)操作
- 由编译器负责把 tile 操作映射到具体线程，简化了实现难度

Tile-Level programming

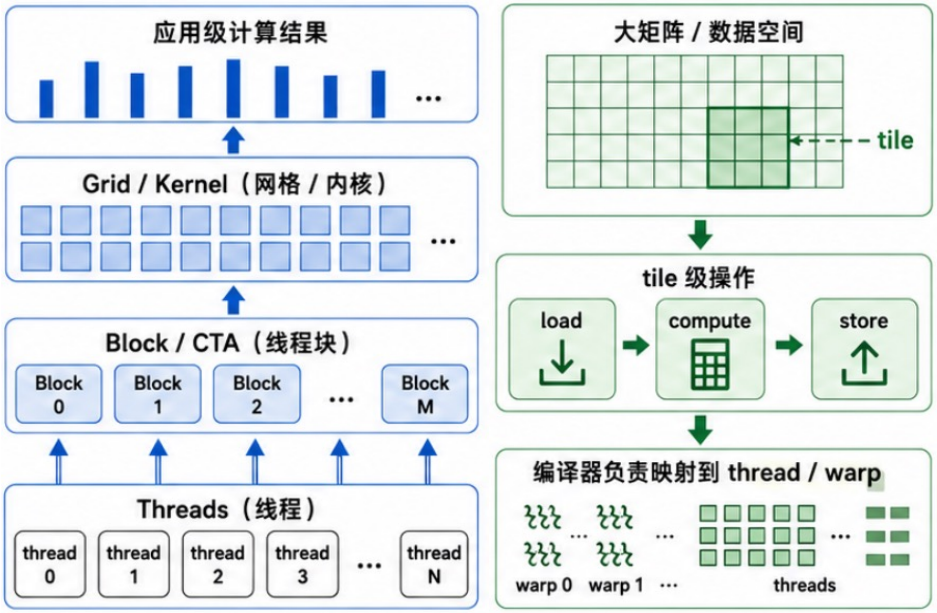
1. 确定当前 tile 的位置
2. 从 global memory 加载 tile
3. 对 tile 做计算
4. 将结果写回 global memory
5. 处理边界

Triton / TileLang 是这个思想的两种 DSL 表达。

Triton 编程模型

- CUDA Kernel 从 Thread 出发，显式计算每个线程负责的数据
- Triton Kernel 从 Program 出发，每个 Program 负责一个数据 Tile
- 程序员描述 Tile 的位置、形状以及 load / compute / store
- 编译器负责将 Tile 内的计算映射到 Thread 和 Warp

Triton 保留数据分块的决策，将线程级映射交给编译器



CUDA　　Triton

Triton 示例：向量加法

执行过程

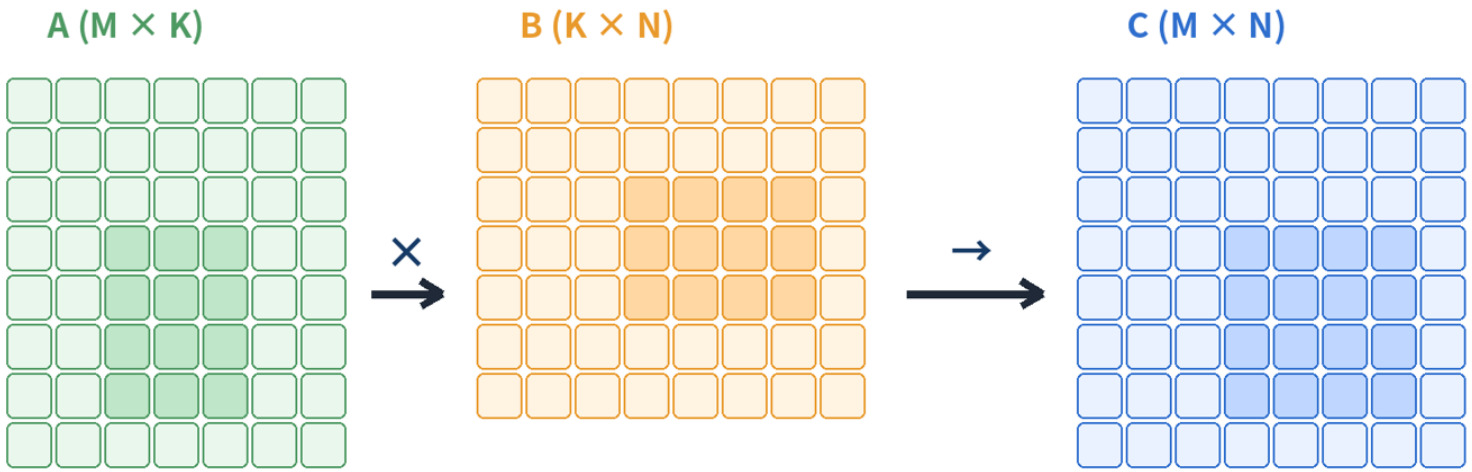
- 每个 program 负责 BLOCK_SIZE 个连续元素
- offsets 表示当前 program 负责的全局下标
- mask 处理最后一个不完整 tile
- tl.load 从 x 和 y 中读取 tile
- out = x_val + y_val 逐元素加法
- tl.store 写回 z

```python
import torch
import triton
import triton.language as tl


@triton.jit
def add_kernel(x, y, z, n: tl.constexpr, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)
    offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n

    x_val = tl.load(x + offsets, mask=mask, other=0.0)
    y_val = tl.load(y + offsets, mask=mask, other=0.0)
    out = x_val + y_val

    tl.store(z + offsets, out, mask=mask)


def add(x, y):
    z = torch.empty_like(x)
    n = x.numel()
    block_size = 1024
    grid = (triton.cdiv(n, block_size),)
    add_kernel[grid](x, y, z, n, BLOCK_SIZE=block_size)
    return z
```

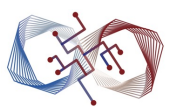
Triton 示例：矩阵乘法

A: (M, K)，B: (K, N)，C: (M, N)

每个 program 负责计算 C 的一个 BLOCK_M × BLOCK_N tile



Triton 示例：矩阵乘法



```python
@triton.jit
def matmul_kernel(
    A, B, C,
    M: tl.constexpr, N: tl.constexpr, K: tl.constexpr,
    stride_am: tl.constexpr, stride_ak: tl.constexpr,
    stride_bk: tl.constexpr, stride_bn: tl.constexpr,
    stride_cm: tl.constexpr, stride_cn: tl.constexpr,
    BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr,
):
    pid_m = tl.program_id(0)
    pid_n = tl.program_id(1)

    offs_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
    offs_n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
    offs_k = tl.arange(0, BLOCK_K)

    acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)

    for k0 in range(0, K, BLOCK_K):
        a_ptrs = A + offs_m[:, None] * stride_am + (k0 + offs_k[None, :]) * stride_ak
        b_ptrs = B + (k0 + offs_k[:, None]) * stride_bk + offs_n[None, :] * stride_bn

        a = tl.load(a_ptrs, mask=(offs_m[:, None] < M) & (k0 + offs_k[None, :] < K), other=0.0)
        b = tl.load(b_ptrs, mask=(k0 + offs_k[:, None] < K) & (offs_n[None, :] < N), other=0.0)

        acc += tl.dot(a, b)

    c_ptrs = C + offs_m[:, None] * stride_cm + offs_n[None, :] * stride_cn
    c_mask = (offs_m[:, None] < M) & (offs_n[None, :] < N)
    tl.store(c_ptrs, acc, mask=c_mask)
```

```python
def matmul(A, B):
    M, K = A.shape
    K2, N = B.shape
    assert K == K2

    C = torch.empty((M, N), device=A.device, dtype=torch.float32)
    block_m, block_n, block_k = 16, 16, 32
    grid = (triton.cdiv(M, block_m), triton.cdiv(N, block_n))

    matmul_kernel[grid](
        A, B, C,
        M, N, K,
        A.stride(0), A.stride(1),
        B.stride(0), B.stride(1),
        C.stride(0), C.stride(1),
        BLOCK_M=block_m, BLOCK_N=block_n, BLOCK_K=block_k,
    )
    return C
```

Triton 的局限性

- 编译器是黑盒：shared memory 布局、bank conflict 全靠编译器
- 生成的 PTX / SASS 不透明，性能问题难定位
- 性能天花板：距 CUTLASS / cuBLAS 手工优化仍有差距
- 复杂 pipeline、warp specialization 难表达
- 复杂 kernel写起来别扭
- 报错和性能回退难分析，依赖反复 autotune

Triton 用放弃部分控制权换开发效率。

当控制权成为瓶颈时，就需要 TileLang 或更底层工具。

TileLang 编程模型

TileLang 更强调显式写出层次结构：

- kernel / program：GPU kernel 入口
- global memory：输入输出张量所在显存
- shared memory tile：CTA 内共享的缓存块
- loop over tiles：沿某维度分块循环

Triton 更接近张量表达式，TileLang 更接近手写 CUDA 的 kernel 结构

TileLang 的核心设计模型：三层抽象

TileLang 提供了三个层次的编程抽象，每个层次对应不同的控制粒度和适用场景。

维度	Level 1 (Hardware-Agnostic)	Level 2 (Tile-Level)	Level 3 (Thread-Level)
并行控制	隐式 (由编译器决定)	显式 Grid/Tile/Block 内并行	显式 Thread/Warp/WarpGroup
内存控制	隐式 (Global Memory 直访为主)	显式 Shared Memory/Local Register/Fragment	显式 Register 与 ISA 级访存
计算映射	自动调度	T.gemm 等 TileOp + Software Pipeline	Inline PTX/ISA、手写微内核
典型性能	中 (依赖编译器)	高	最高 (局部热点)
可移植性	高	中-高	低-中 (随硬件变化)
复杂度	低	中	高
适用场景	原型、通用算子、教学	主力实现、性能工程	瓶颈路径、特殊指令

CUDA to TileLang

CUDA

```cuda
__global__ void matmul(half* A, half* B, half* C,
                       int M, int N, int K) {
    int tx = threadIdx.x, ty = threadIdx.y;
    int row = blockIdx.y * BM + ty;
    int col = blockIdx.x * BN + tx;
    __shared__ half As[BM][BK], Bs[BK][BN];
    float acc = 0.0f;
    for (int k0 = 0; k0 < K; k0 += BK) {
        if (row < M && k0 + tx < K)
            As[ty][tx] = A[row * K + k0 + tx];
        if (k0 + ty < K && col < N)
            Bs[ty][tx] = B[(k0 + ty) * N + col];
        __syncthreads();
        #pragma unroll
        for (int k = 0; k < BK; ++k)
            acc += As[ty][k] * Bs[k][tx];
        __syncthreads();
    }
    if (row < M && col < N)
        C[row * N + col] = acc;
}
```

TileLang

```tilelang
with T.Kernel(...) as (bx, by):
    As = T.alloc_shared((BM, BK), "float16")
    Bs = T.alloc_shared((BK, BN), "float16")
    C_local = T.alloc_fragment((BM, BN), "float32")
    T.clear(C_local)
    for ko in T.Pipelined(T.ceildiv(K, BK), num_stages=3):
        T.copy(A[...], As)
        T.copy(B[...], Bs)
        T.gemm(As, Bs, C_local)
    T.copy(C_local, C[...])
```

CUDA 展开线程级细节
TileLang 保留 Tile 级数据流与计算结构

TileLang primitives

定义与启动

- `@tilelang.jit`：生成并编译可调用 Kernel
- `@T.prim_func`：定义 TileLang/TIR 函数
- `T.Kernel(...)`：创建 Grid，一个实例对应一个 CTA

内存层次

- 函数参数 `T.Tensor`：Global Memory
- `T.alloc_shared`：CTA 共享的输入 Tile
- `T.alloc_fragment`：寄存器中的计算结果

数据流与计算

```
Global Memory ---> Shared Memory ---> Register Fragment ---> Global Memory
            T.copy                T.gemm                  T.copy
```

- `T.Pipelined`：沿 K 维分块，并重叠搬运与计算
- `T.copy`：表达不同内存层次之间的数据搬运
- `T.gemm`：表达 Tile 级矩阵乘加

TileLang Matmul

TileLang 的核心结构：

- 显式分配 shared memory tile
  A_shared、B_shared 存储输入数据块
- 分配 register fragment 作为累加器
  C_local 累积中间结果
- K 维分块循环，支持 pipeline
  每轮从 global 搬运数据到 shared
  调用 gemm 更新 C_local
- 循环结束后写回 global memory

强调数据搬运和层次化计算

```python
import tilelang
import tilelang.language as T


@tilelang.jit
def matmul(
    M: int,
    N: int,
    K: int,
    BLOCK_M: int,
    BLOCK_N: int,
    BLOCK_K: int,
    dtype: str = "float16",
    accum_dtype: str = "float32",
):
    @T.prim_func
    def main(
        A: T.Buffer((M, K), dtype),
        B: T.Buffer((K, N), dtype),
        C: T.Buffer((M, N), accum_dtype),
    ):
        with T.Kernel(
            T.ceildiv(N, BLOCK_N),
            T.ceildiv(M, BLOCK_M),
            threads=128,
        ) as (bx, by):
            A_shared = T.alloc_shared((BLOCK_M, BLOCK_K), dtype)
            B_shared = T.alloc_shared((BLOCK_K, BLOCK_N), dtype)
            C_local = T.alloc_fragment((BLOCK_M, BLOCK_N), accum_dtype)

            T.clear(C_local)

            for k in T.Pipelined(T.ceildiv(K, BLOCK_K), num_stages=3):
                T.copy(A[by * BLOCK_M, k * BLOCK_K], A_shared)
                T.copy(B[k * BLOCK_K, bx * BLOCK_N], B_shared)
                T.gemm(A_shared, B_shared, C_local)

            T.copy(C_local, C[by * BLOCK_M, bx * BLOCK_N])

    return main
```

小结

Triton

- 快速写出简洁 kernel
- 适合 elementwise、reduction、matmul、attention

TileLang

- 显式表达内存层次和数据复用
- 适合讲清高性能 kernel 结构

CUDA C++

- 控制粒度最细
- 适合极端性能调优或特殊硬件机制

参考资料

- NVIDIA CUDA C Programming Guide
  https://docs.nvidia.com/cuda/cuda-c-programming-guide/
- Triton Language Documentation
  https://triton-lang.org/
- Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations (2019)
  https://www.eecs.harvard.edu/~htk/publication/2019-mapl-tillet-kung-cox.pdf
- TileLang
  https://github.com/tile-ai/tilelang
- NVIDIA Deep Learning Performance Documentation
  https://docs.nvidia.com/deeplearning/performance/