SIMD vs SIMT

ZOMI

Course chenzomi12.github.io

GitHub github.com/chenzomi12/DeepLearningSystem

Talk Overview

1. **AI 计算体系**
   - 深度学习计算模式
   - 计算体系与矩阵运算
2. **AI 芯片基础**
   - 通用处理器 CPU
   - 通用图形处理器 GPU
   - AI专用处理器 NPU/TPU
3. **GPU详解**
   - 英伟达GPU架构发展
   - Tensor Core和NVLink
4. **国外 AI 芯片**
   - 特斯拉 DOJO 系列
   - 谷歌 TPU 系列
5. **国内 AI 芯片**
   - 壁仞科技芯片架构
   - 寒武纪科技芯片架构
6. **AI芯片的思考**
   - SIMD&SIMT与编程体系
   - AI芯片的架构思路与思考

Talk Overview

I. SIMD & SIMT 区别与联系

- SIMD 单指令多数据
- SIMT 单指令多线程
- NVIDIA CUDA实现

1. SIMD

单指令多数据

SIMD: Single Instruction Multiple Data 是什么

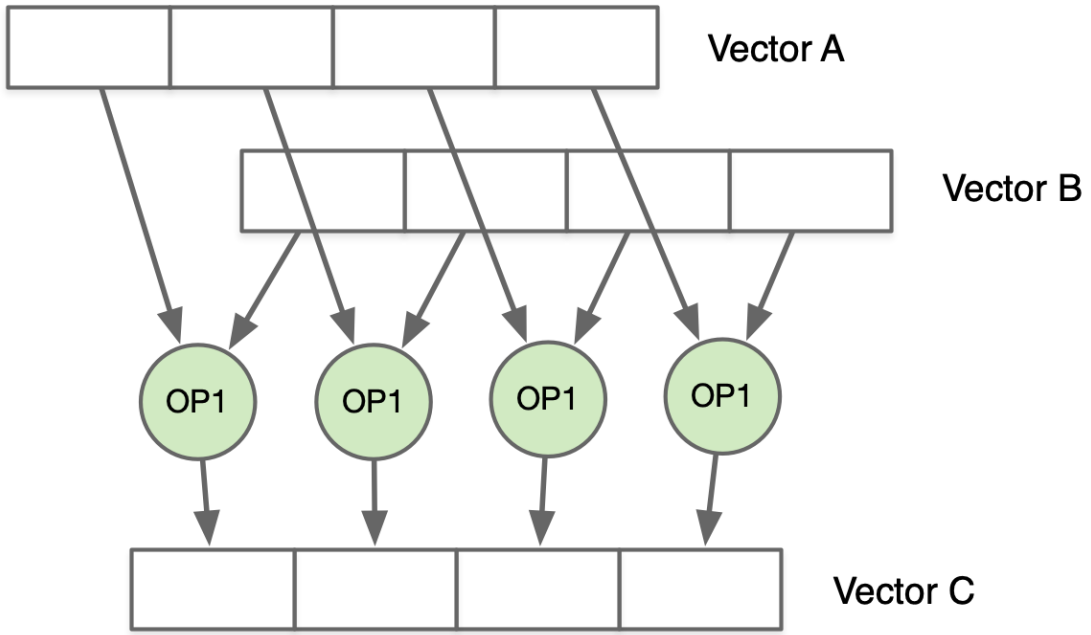
- SIMD 对多个进行同样操作的处理元素同时进行同等的计算操作。利用了数据级别的并行性，而不是并发性；有多个计算，但是只有一个进程在运行。
- SIMD 允许使用单一命令对多个数据值进行操作。属于提升 CPU 计算能力数据并行方法。仅需要更宽位的 ALU 和较小的控制逻辑。

- PS：SIMD 仍然是单线程，不是多线程操作，硬件上仅需要一个计算核心，只不过一次操作多个数据，需要与 GPU 的多线程区分。

SIMD 计算本质

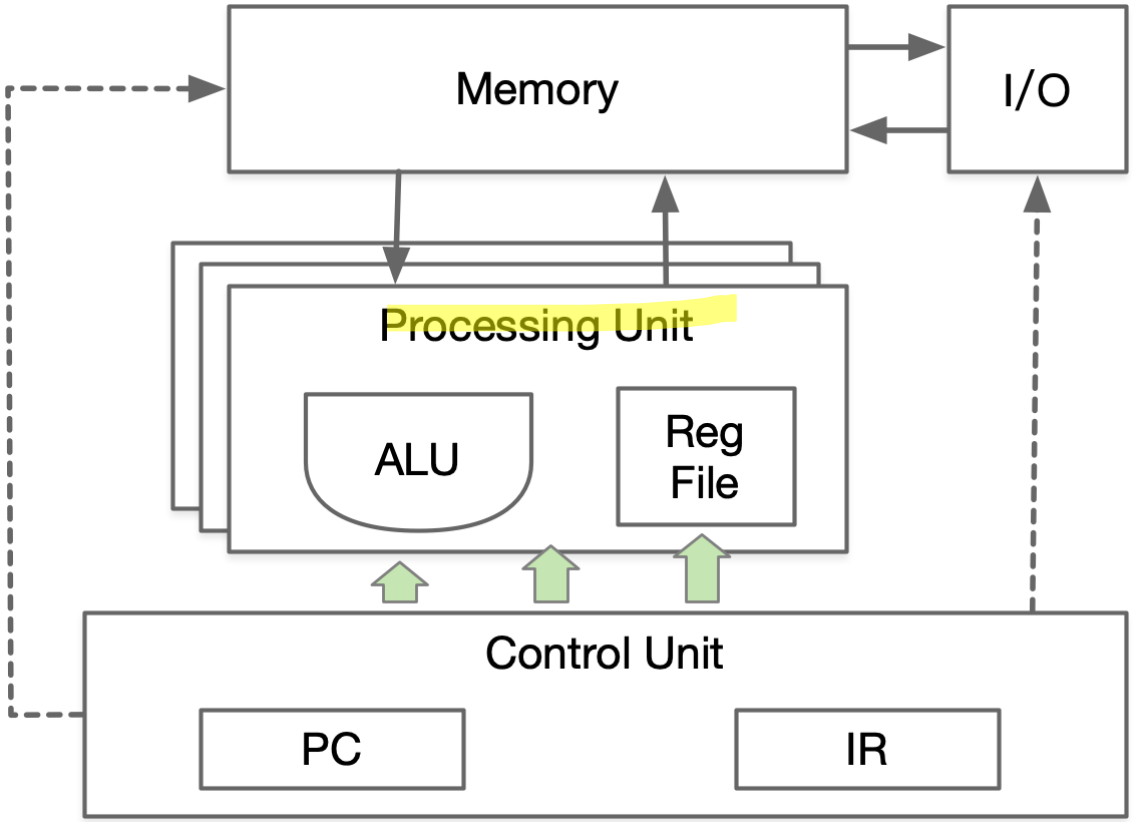
- SIMD 指在多个数据上并行进行相同操作的硬件部件。例如将两个 vector 作为操作数，对于两个 vector 的操作数进行相同操作，下面以 vector 为4个元素例：

$$C[0:3] = A[0:3] \times B[0:3]$$



SIMD 计算本质

- SIMD 实现一次乘法可以完成多个元素的计算。因此这要求硬件上，增加 ALU 单元的数量，同时也需要增加同一个功能单元的数据通路数量，才能够实现相同时钟周期内提升计算的吞吐量。



SIMD 约束

- **缺点**：SIMD 使用独立线程，该线程能同时进行多个数据的计算。由于 ALU 宽度限制，其计算要求数据是类型、格式、大小必须严格对齐。
- **优点**：一定程度提升计算性能，利用内存数据总线宽度多个数据可以同时从内存 Read/Write。如 $C[0:3] = A[0:3] \times B[0:3]$ 操作使用 SIMD 代码为原来的 1/4，执行周期也降为 1/4。

```
1   t1 = LD B, i
2   t2 = LD C, i
3   t3 = t1 + t2
4   ST A, i, t3
5   t1 = LD B, i+1
6   t2 = LD C, i+1
7   t3 = t1 + t2
8   ST A, i+1, t3
9   t1 = LD B, i+2
10  t2 = LD C, i+2
11  r3 = t1 + t2
12  ST A, i+2, t3
13  t1 = LD B, i+3
14  t2 = LD C, i+3
15  t3 = t1 + t2
16  ST A, i+3, t3
```

```
1
2   v1 = LD B, i, 4
3   v2 = LD C, i, 4
4   v3 = v1 + v2, 4
5   ST A, i, 4, v3
6
```

SIMD DEMO

- Intel从MMX开始支持SIMD，ARM通过NEON将SIMD扩展引入ARM-Cortex架构。NEON SIMD 单元 128-bit 宽，包含 16 个 128-bit 寄存器，能够被用来当做32个64-bit寄存器。这些寄存器能被当做是同等数据类型的vector。

```
2   //对四个数据同时进行乘法操作
3   A[3:0]=B[3:0] * C[3:0]
4
5   //一个寄存器128bit，可存放4x32bit，s15寄存器存放向量B
6   vldmia.32 r0!, {s15}
7   //通过 s14 寄存器存放向量C
8   vldmia.32 r1!, {s14}
9   // s15=s15*s14
10  vmul.f32 s15, s15, s14
11  // 保存s15的计算结果
12  vstmia.32 r2!, {s15}
13
```

2. SIMT

单指令多线程

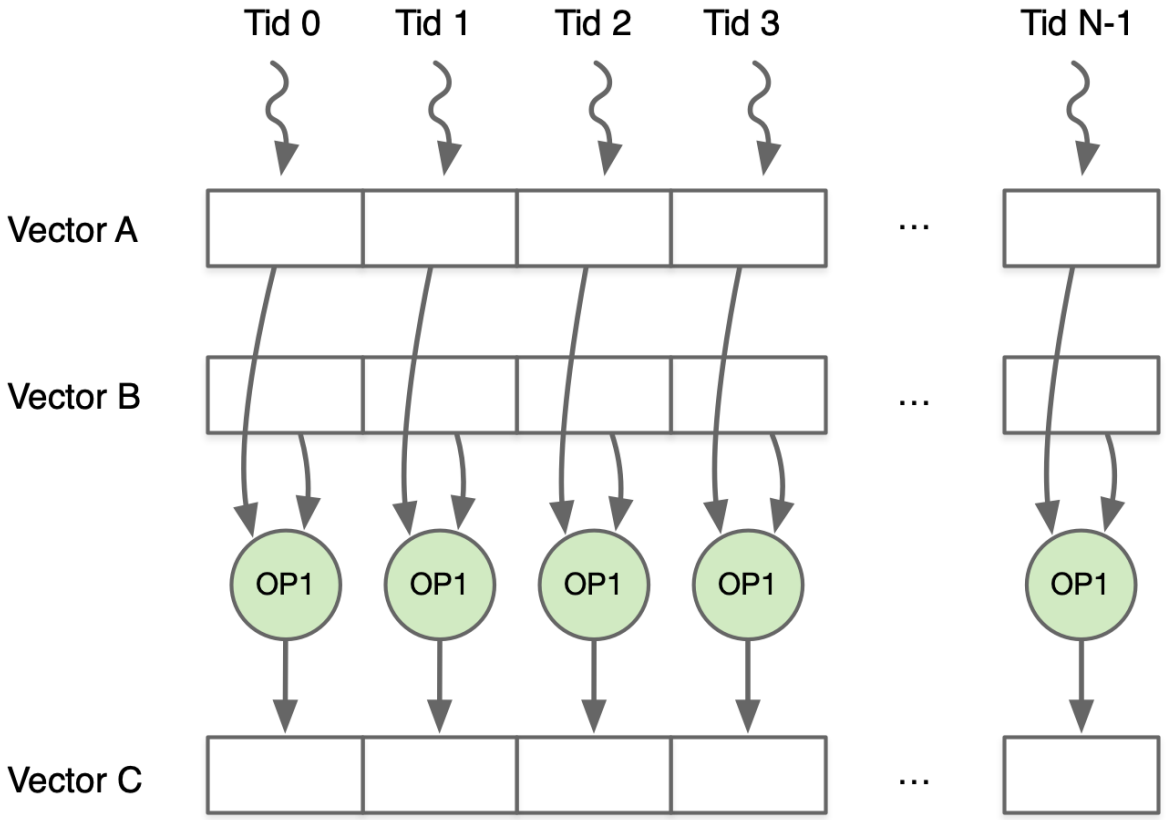
SIMT: Single Instruction Multiple Threads

- SIMT 是 NVIDIA 提出基于 GPU 的新概念。二者都通过将同样的指令广播给多个执行单元来实现并行。主要的不同在于 SIMD 要求所有的 vector element 在统一的同步组里（一个线程内）同步执行，而 SIMT 允许多个线程在一个 warp 中独立执行。
- SIMT 类似 CPU 上的多线程。最简单的理解 SIMT 在多个计算核心 core 系统，每一个 core 有独立的寄存器文件 RF、计算单元 ALU，但是没有独立指令缓存 instruction cache、解码器、Program Counter register，命令从统一的 Instruction cache 广播给多个 SIMT core。

- PS：所有 core 各自独立，在不同的数据上执行相同的计算操作，即执行命令相同。多个线程各有各的处理单元，和 SIMD 公用一个 ALU 不同。

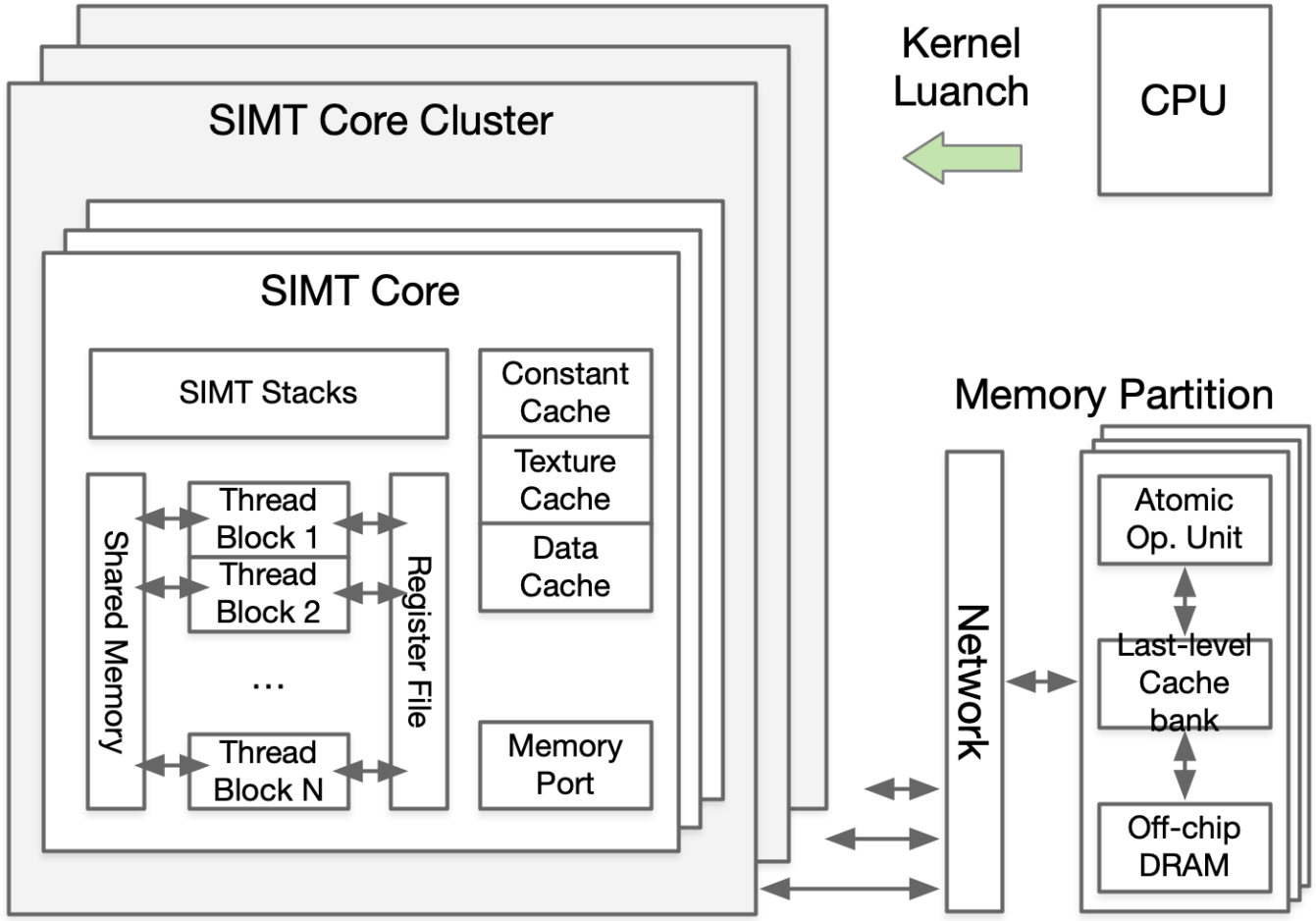
SIMT 计算本质

- 假设有两个等长数组，需要每个元素逐一对应相乘。SIMT 给每个元素分配一个线程，一个线程只需要完成一个元素的加法，所有线程并发(Concurrent)执行完成后，两个数组的相加就完成。



SIMT 硬件结构

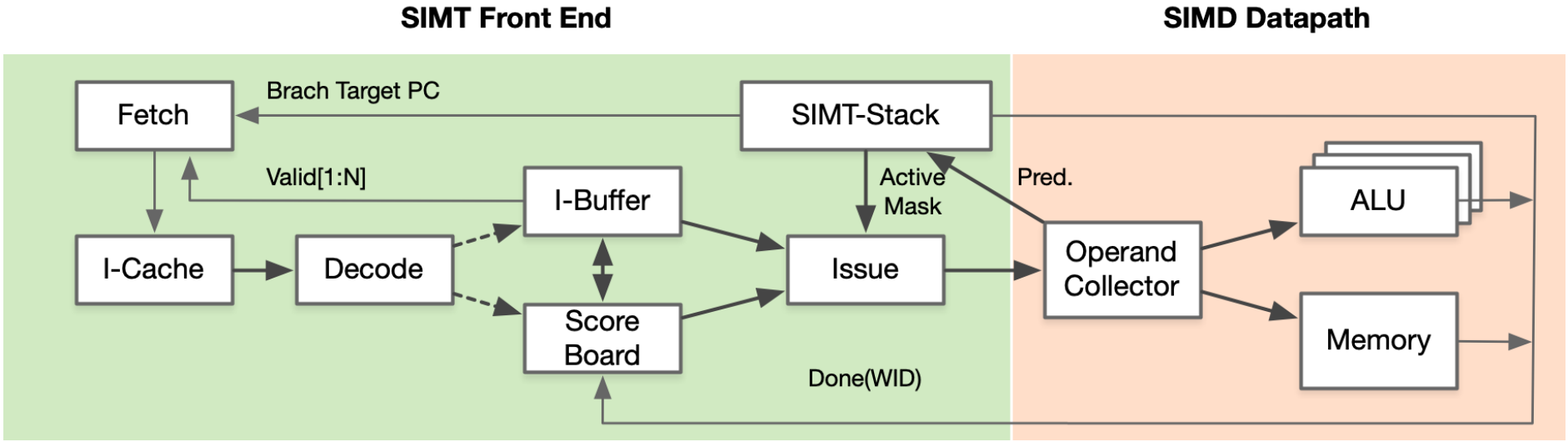
- SIMT 提供一个多核系统，每个核有独立的寄存器文件 RF、ALU、Data Cache，但是只有一个 Program Counter寄存器、一个指令Cache和一个译码器，指令被同时广播给所有的 SIMT 核。
- 以 GPU 为例：由多个 SIMT Core Cluster 组成，每个SIMT Core Cluster 由多个 Core 构成，Core 中有多个 Thread Block。



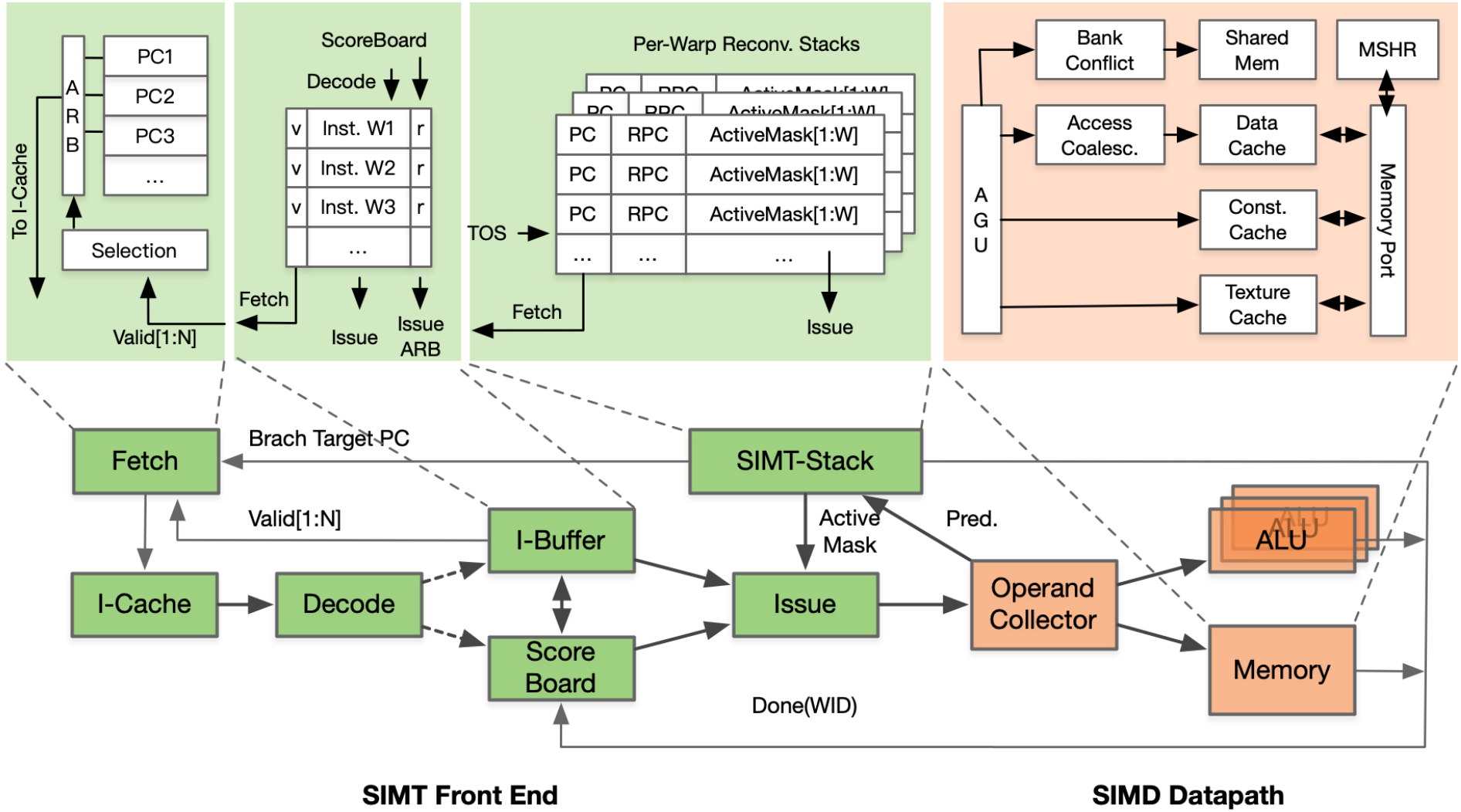
SIMT 硬件核心流水

- GPU 的 SIMT 流水线可以被分为 SIMT front-end 和 SIMD back-end。流水线中存在三个调度循环：取指循环、指令发射循环和寄存器访问循环。

1. 取指循环包含 Fetch、I-Cache、Decode和I-Buffer四个阶段；
2. 指令发射循环包含 I-Buffer、Score Board、Issue和SIMT-Stack四个阶段；
3. 寄存器访问循环包含 Operand Collector、ALU 和 Memory 三个阶段；



SIMT 硬件核心流水


PC1
PC2
PC3
...
A
R
B
To I-Cache
Selection
Valid[1:N]
ScoreBoard
Decode
v
Inst. W1
r
Inst. W2
Inst. W3
Fetch
Issue
Issue
ARB
Per-Warp Reconv. Stacks
PC
RPC
ActiveMask[1:W]
TOS
Fetch
Issue
Bank
Conflict
Shared
Mem
MSHR
Access
Coalesc.
Data
Cache
Const.
Cache
Texture
Cache
Memory Port
A
G
U
Fetch
Brach Target PC
SIMT-Stack
Valid[1:N]
I-Buffer
Active
Mask
Pred.
I-Cache
Decode
Issue
Operand
Collector
ALU
Score
Board
Memory
Done(WID)
SIMT Front End
SIMD Datapath

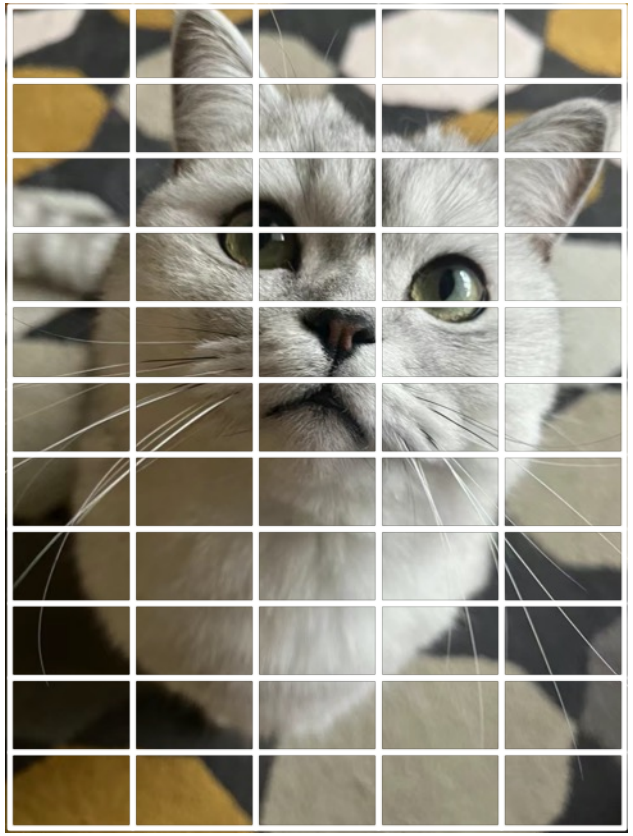
SIMD 和 SIMT 总结

1. SIMT 与SIMD 本质相同，都是单指令多数据。
2. SIMT 形式上多线程，本质上硬件执行还是单线程。
3. SIMT 比 SIMD 更灵活，允许一条指令对数据分开寻址；SIMD是必须连续连续取址。

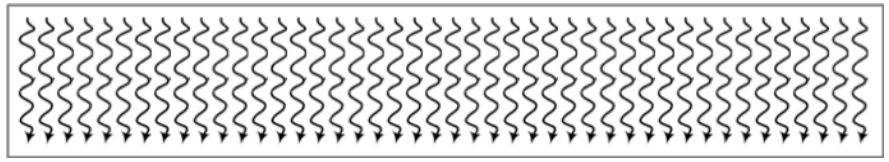
4. Anyway，SIMT 是 SIMD 的一种推广，编程模式上更加灵活。

3. NVIDIA CUDA 实现

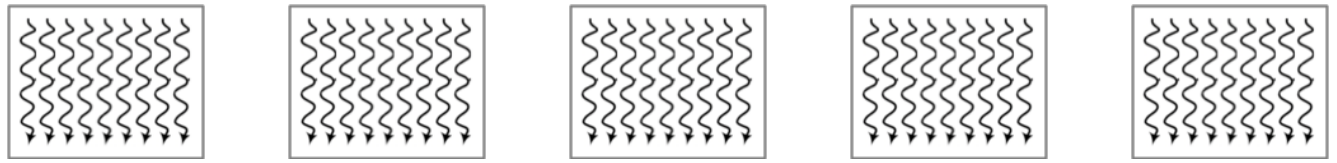
线程分层执行



网格 Grid 表示所有要执行的任务



网格 Grid 中包含了很多相同线程 Threads 数量的块 Blocks



块 Block 中的线程数独立执行

可以通过本地数据共享

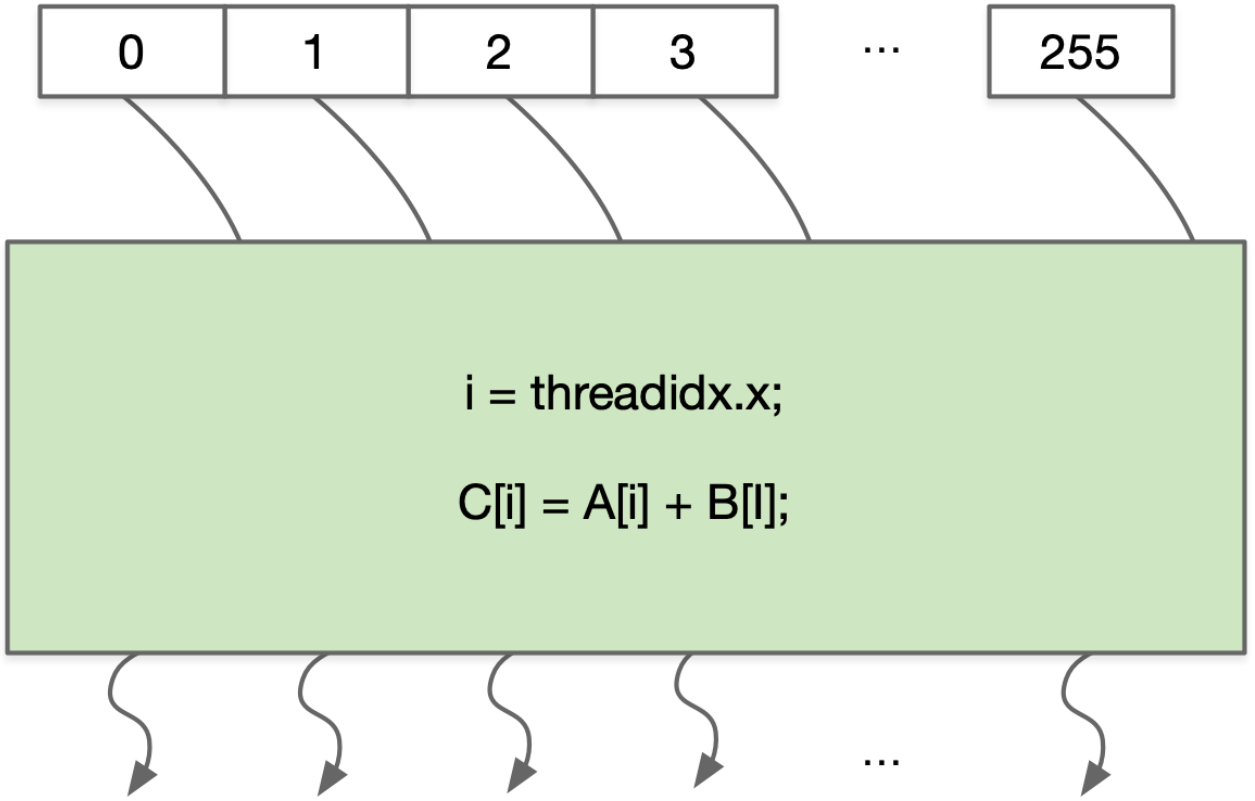
同步交换数据

CUDA 编程

- 与 SIMD 不同，SIMT允许程序员为独立、标量线程编写线程级的并行代码，还允许为协同线程编写数据并行代码。为了确保正确性，开发者可忽略 SIMT 行为，通过维护很少需要使一个 warp 块内的线程分支的代码，即可获得硬件并行带来的实现显著的性能提升。

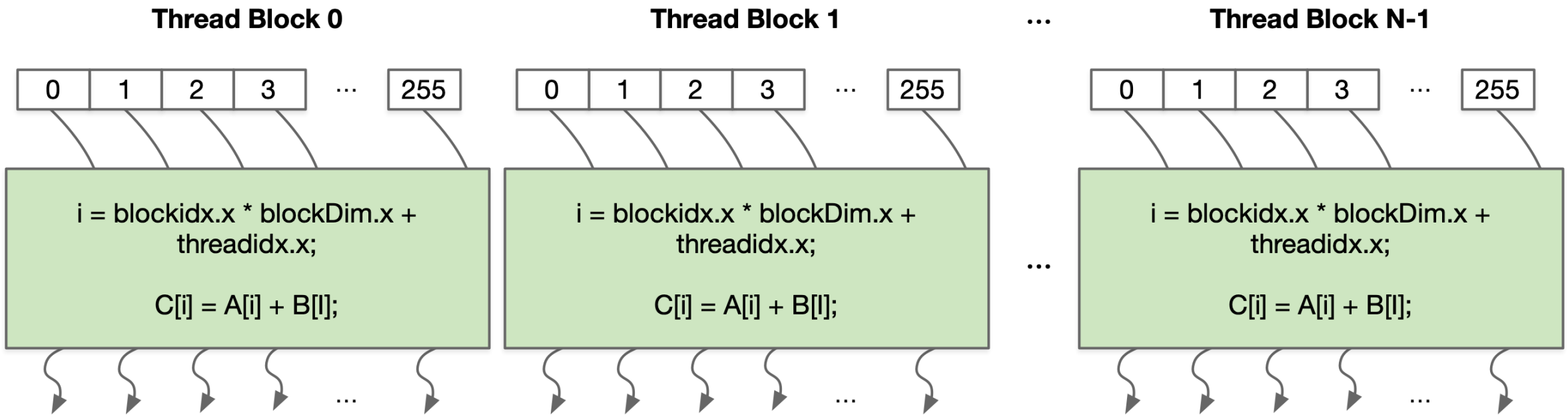
NVIDIA CUDA 实现

- 一个 Block 的所有线程执行同一个 kernel 代码，每一个线程有一个自己的 index(threadIdx.x) 用于计算内存地址和执行控制决策。



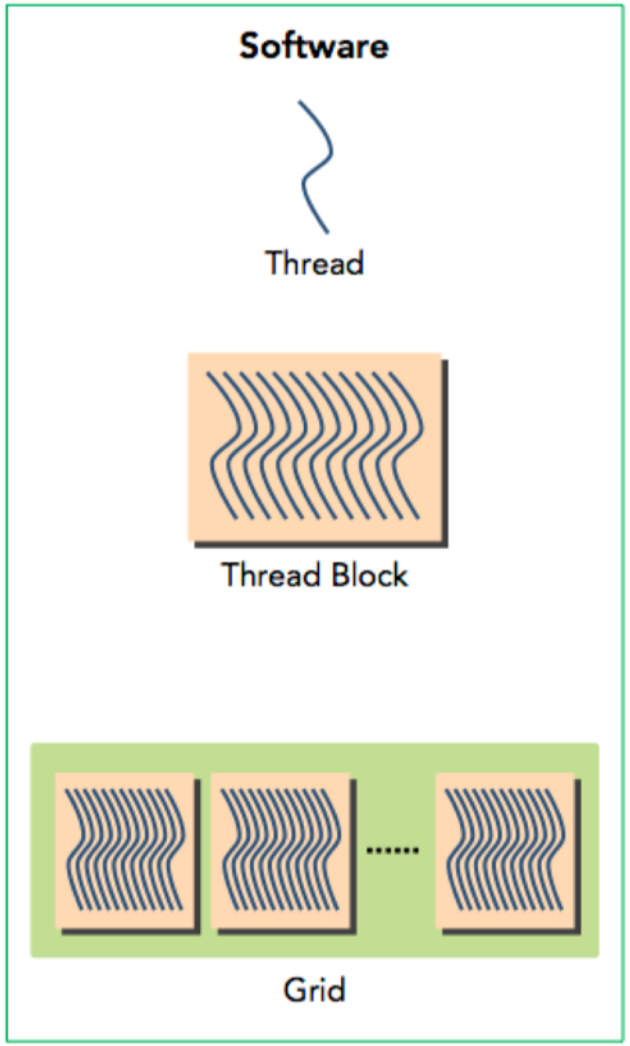
NVIDIA CUDA 实现

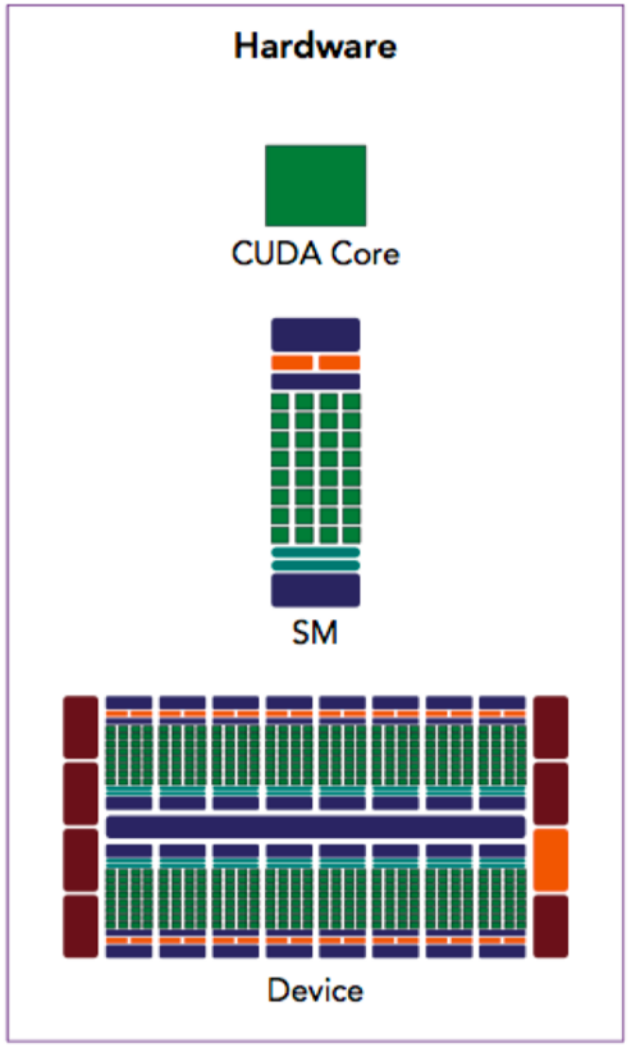
- 将一个 Grid 的线程组划分为多个 Thread Block，线程块就变成了 SM 的调度单元。



NVIDIA CUDA 实现

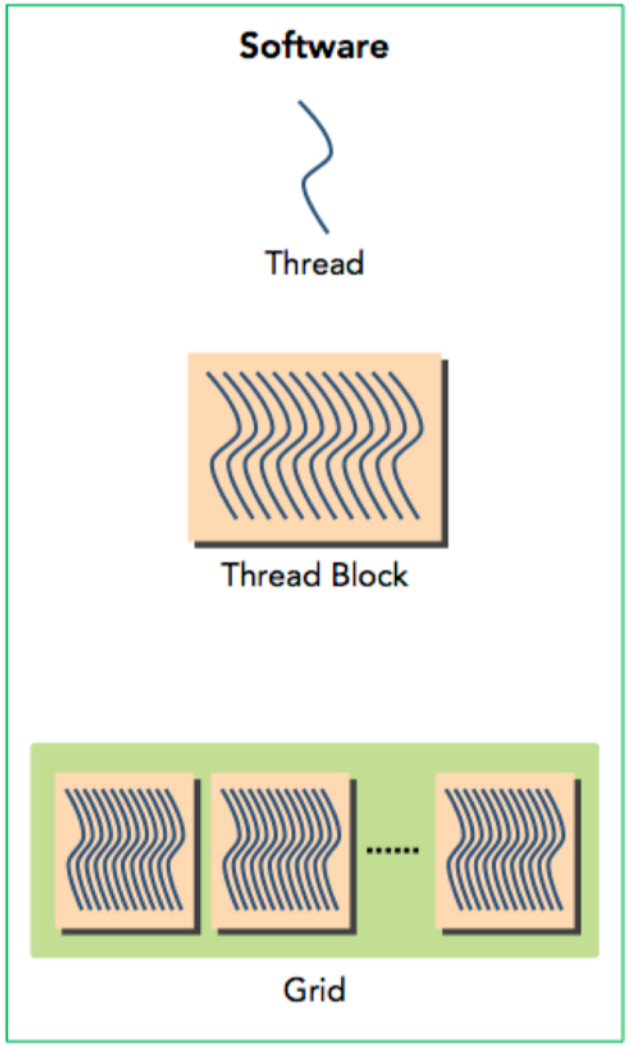
- 线程以线程块为单位被分配到 SM 上，SM 维护线程块和线程 ID，SM 管理和调度线程执行。
- 每个线程块又按照 32 个线程一个 Warp 被划分执行，Warp 是 SM 的调度单位，Warp里的线程执行SIMD。

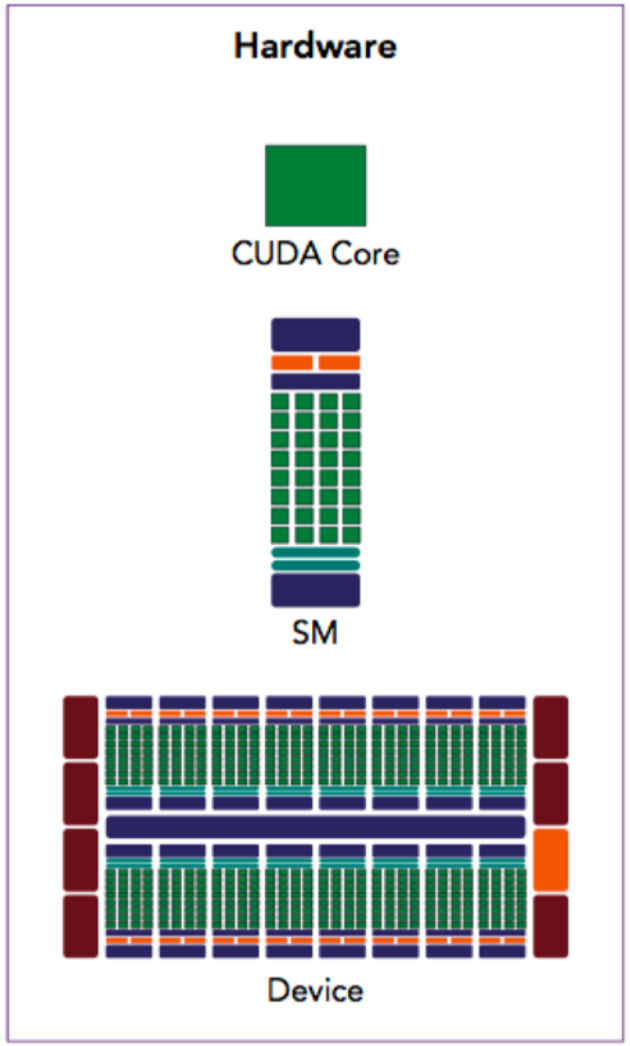




CUDA 跟 NVIDIA 硬件架构的关系

- Block 线程块只在一个 SM 上通过 Wrap 进行调度。
- 一旦在 SM 上调起了 Block 线程块，就会一直保留到执行完 Kernel。
- SM 可以同时保存多个 Block 线程块，快间并行的执行。



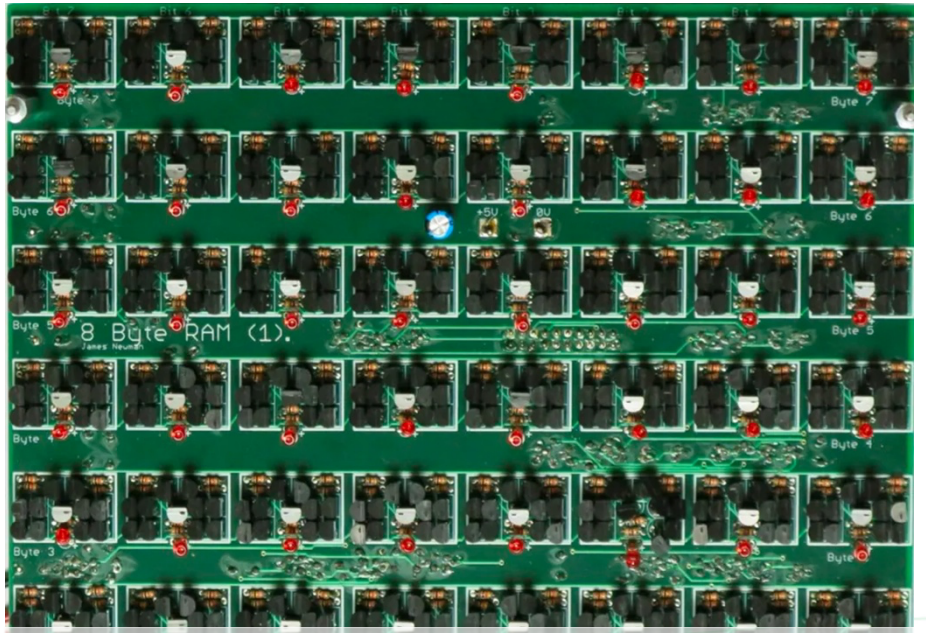


4. 编程本质 vs 硬件执行本质

什么是编程模型？什么是执行模型？

- Programming Model (Software)

- Execution Model (Hardware)



Thank you

把AI系统带入每个开发者、每个家庭、每个组织，构建万物互联的智能世界

Bring AI System to every person, home and organization for a fully connected, intelligent world.

ZOMI

Course chenzomi12.github.io

GitHub github.com/chenzomi12/DeepLearningSystem