在 SIMT 和 DSA 角度看英伟达生态

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

1. SIMD、SIMT、DSA区别与思考

- 概念澄清
- 借鉴点1：流水编码
- 借鉴点2：SIMT前端
- 借鉴点3：分支预测
- 借鉴点4：交互方式

1、三者概念澄清

SIMT 、SIMD、SPMT 关系

- SIMD、SIMT、DSA、SPMD 等词虽然经常放到一起讨论，但指代关系还是比较混乱，有时候指的是执行指令方式、有时候指的是硬件体系结构、有时候又是指系统编程模型。在每个层面边界也不是很清晰，于是就有了一定偷换概念的空间。

1. SIMD：代指指令的执行方式和对应映射的硬件体系结构；
2. SIMT：SIMD 指令为主，具备 warp scheduler 等硬件模块，支持 SPMD 编程模型的硬件架构；
3. SPMD：这里指代一种具体的类似于 CUDA 的编程模型；
4. DSA：主要指NPU/TPU等专门针对AI的特殊硬件架构；

SIMD vs. SIMT 执行模式

- **SIMD**：单顺序的指令流执行 -> 每条指令多个数据输入同时执行
  - [VLD, VLD, VADD, VST], VLEN
- **SIMT**：标量指令的多个指令流 -> 动态地把线程按wrap分组执行
  - [LD, LD, ADD, ST], NumThreads

- **SIMT的优势：**
  - 无需开发者费力把数据凑成合适的矢量长度
  - 从硬件设计上解决大部分 SIMD data path 的流水编排问题
  - 线程可以独立执行，使得每个 thread 相对灵活，允许每个线程有不同的分支
  - 一组执行相同指令的线程由硬件动态组织成 warp，加快了 SIMD 的计算并行度

2. 英伟达生态的思考点

SIMT 与 CUDA 的关系

- 英伟达在 SIMT 硬件架构为了维护 CUDA 生态做出了对架构调整的取舍，因此 NV 硬件架构针对 CUDA 具备约束，如保留 SM、Warp、Thread 等多级概念。CUDA 架构在近年来没有重大的改变，主要是维护编程体系的软件对外抽象易用性。
- DSA 之所以在硬件架构的指令和设计上激进，并非软件体系做的好，而是在开始并没有太多考虑编程体系的问题，自然没有软硬件协同的架构约束。

CUDA 解决硬件相关问题

- CUDA 成功之处：通过 SIMT 架构掩盖了1）流水编排、2）并行指令隐藏。

DSA 硬件架构执行方式

- **DSA硬件架构**：一般是指单核单线程 thread，线程 thread 内指令可以通过多核共享 Cache 协作。编程模型上缺乏统一的标准。硬件以 AI 加速芯片（TPU、NPU 等）为主。
- **DSA执行方式**：该类硬件目前裸接口一般是每个核一个线程，每个线程内串行调用 DSA 指令集，指令在硬件上通常会分发到不同的指令执行流水线上，正确性部分靠软件同步实现，部分靠硬件保证。

CUDA 客户能力区分

1. **初阶用户**：掌握 CUDA 并行编程能力，了解 NV SIMT 硬件基础架构，可以拿到并行指令、流水掩盖、并行计算三部分性能。
2. **中阶用户**：进一步运用 CUDA 提供的切块 Tiling、流水 Pipeline 能力，进一步获取更高的性能收益。
3. **高阶用户**：深入了解 SIMT 微架构细节，解决线程在 bank 冲突、精细化流水掩盖、精细化指令使用、极致的切块 Tiling 策略，从而实现极致性能。

深度开发易用性

CPU Program

```
1
2   void add_matrix(float* a, float* b, float* c, int N) {
3       int index;
4
5       for (int i = 0; i < N; ++i){
6           index = i + j * N;
7           c[index] = a[index] + b[index];
8       }
9   }
10
11  int main() {
12      add_matrix(a, b, c, N);
13  }
14
```

GPU Program

```
1
2   __global__ add_matrix(float* a, float* b, float* c, int N){
3       int i = blockIdx.x * blockDim.x + threadIdx.x;
4       int j = blockIdx.y * blockDim.y + threadIdx.y;
5       int index = i + j * N;
6       if (i < N && j < N){
7           c[index] = a[index] + b[index];
8       }
9   }
10
11  int main() {
12      dim3 dimBlock(blocksize, blocksize);
13      dim3 dimGrad(N/dimBlock.x, N/dimBlock.y);
14      add_matrix<<<dimGrad, dimBlock>>>(a, b, c, N);
15  }
16
```

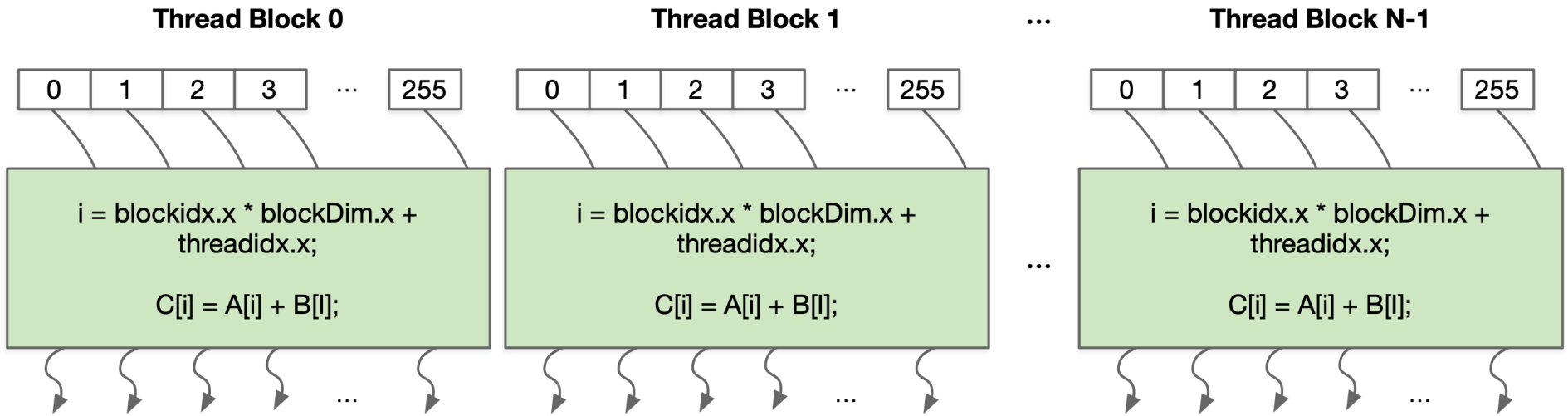
NV GPU和CUDA是SIMT最成功的实践

借鉴点 1

流水编排

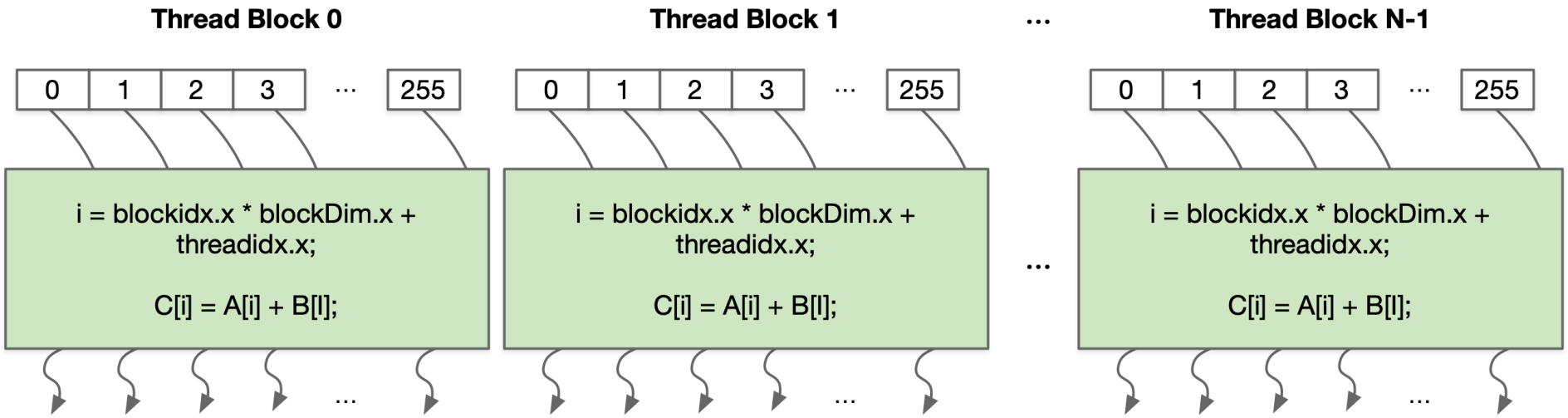
借鉴点1：从硬件设计上解决 SIMD data path 流水编排问题

- 程序执行最大的瓶颈是访存和控制流：单线程 CPU 需要大量资源进行分支预测、超前执行、缓存、预取等机制来缓解。SIMD 往往依赖 CPU 自身乱序、投机、缓存和预取等能力来缓解； NV GPU 依靠多线程交错执行执行提升整体计算性能。



借鉴点1：从硬件设计上解决 SIMD data path 流水编排问题

- SIMT 的上层 CUDA 编程模型形态先抛开不谈。即使在 DSA 上为 SIMD 硬件封装了 SIMT 前端，如果遇到执行指令有依赖，基础性能也会非常差，流水编排仍然需要开发者动手，想写出开箱性能较优的代码同样很难。

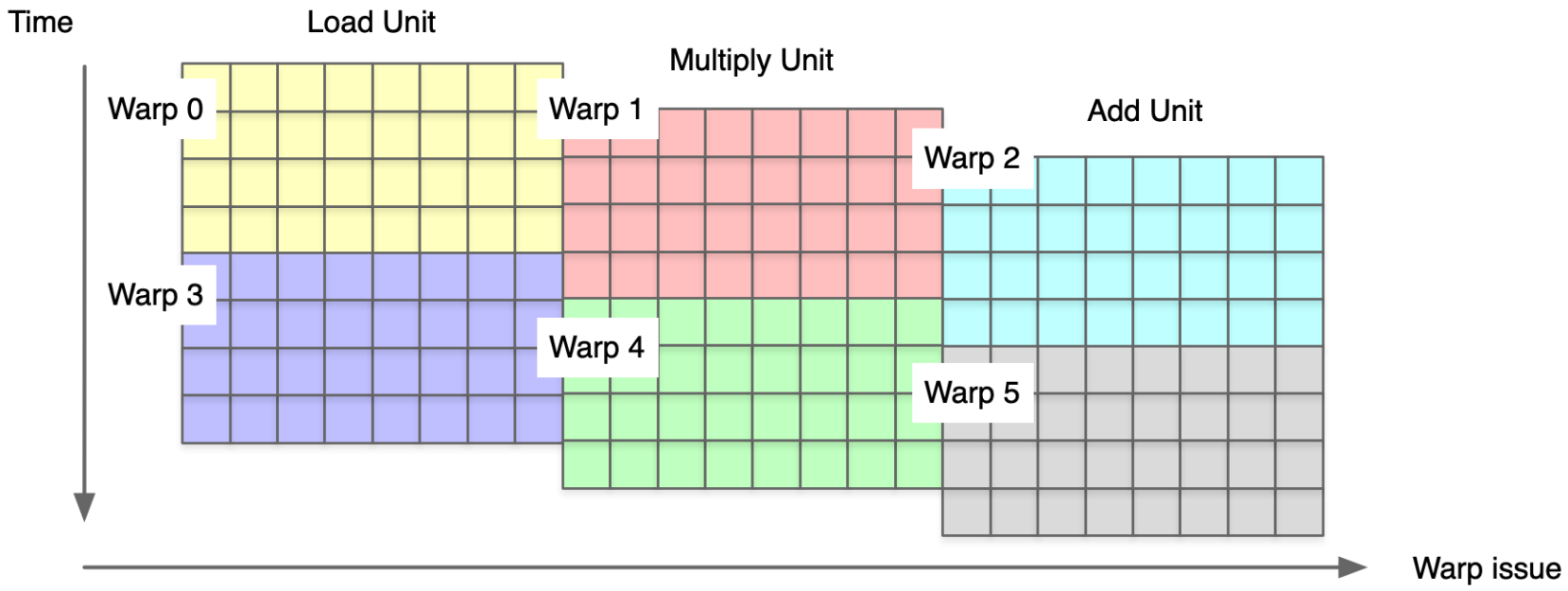


借鉴点 2

SIMT 前端硬件

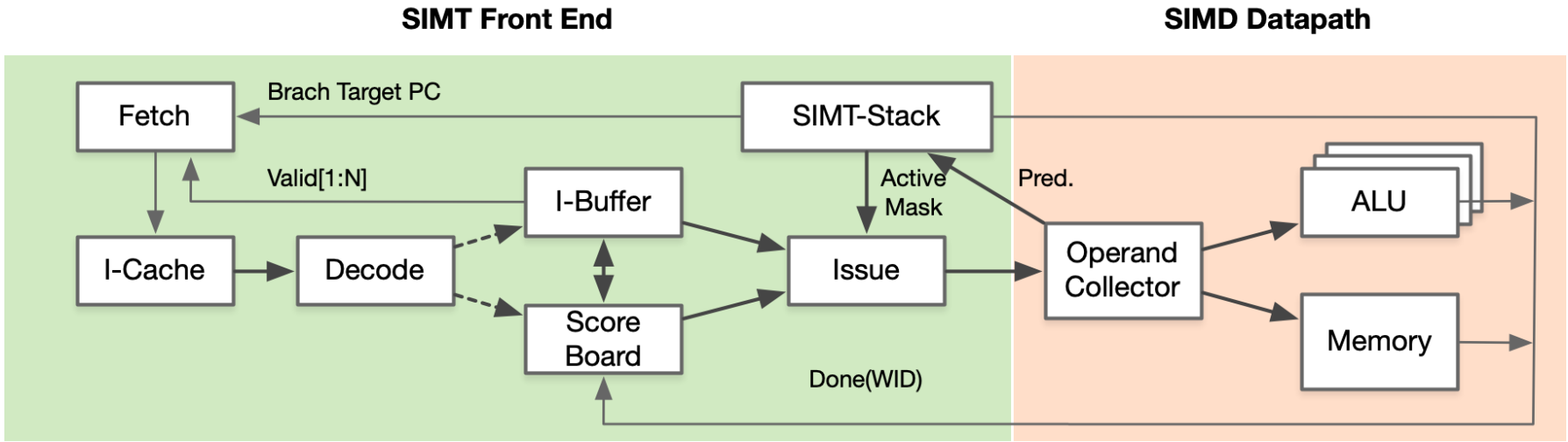
借鉴点2：增加 SIMT 前端硬件

- **通过 Warp 隐藏流水：**CUDA 编程模型中，每一个 thread block 内部需要有很多并行线程，隐式分成了若干个 Warp，每个 Warp 包含串行交错的访存和计算。GPU 通过 Warp Scheduler 动态交错执行，一组 Warp 流水阻塞了就切下一 Warp，隐式通过 Warp 的并行掩盖指令流水阻塞。开发者可以拿到较好的性能。



借鉴点2：增加 SIMT 前端硬件

1. DSA 硬件架构同样可以引入 Warp Scheduler 进行指令流水掩盖，让每个 DSA 核执行多个线程，相互掩盖流水线阻塞。
2. GPU 使用 Warp 来掩盖指令流水是基于运行时的具体信息，而开发者和编译器只能基于静态信息进行流水编排，很难做到足够均衡，使得 SIMD/DSA 进行手工/编译器自动流水编排相对困难，资深开发者也很难把流水编排足够好。



借鉴点2：增加 SIMT 前端硬件

- **流水阻塞掩盖：**SIMT 表达（让用户主动写多线程）+ warp scheduler（多线程相互掩盖流水阻塞）
- **SIMD指令掩盖：**SIMT表达（写通用单线程）+warp分组（组成SIMD指令）
- NV CUDA 没有解决 DSA 指令掩盖，目前通过给开发者 Warp 概念，透传指令 API 来解决表达和使用的问题，因此 CUDA 的上手门槛并不低，需要了解 NV GPU 硬件细节。

借鉴点 3

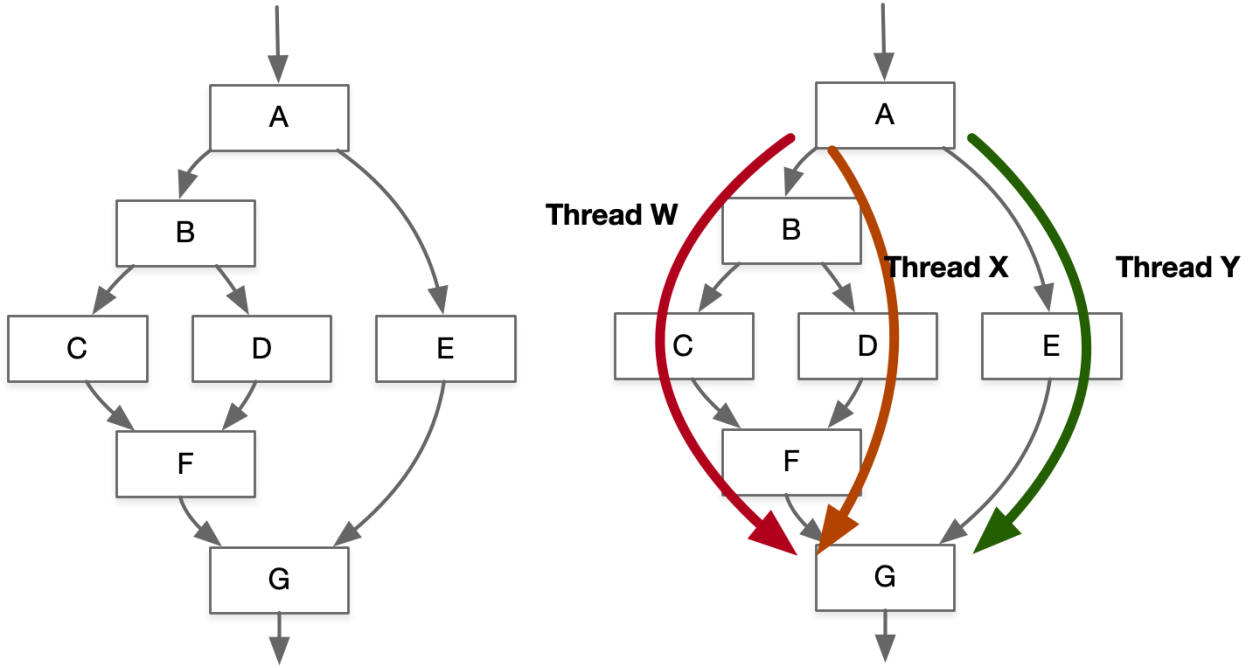
分支预测

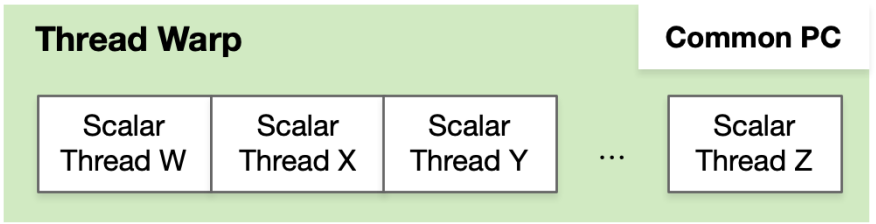
借鉴点3：分支预测机制

- SPMD 编程模型对于分支预测和控制流的容忍度高是支撑易用性的重要手段，减少分支和连续访存是软件层面、易用性需要关注的优化点。当然，在SIMD 的硬件上同样可以通过 Predicate/mask 和对 gather/scatter 指令 memory coalescing 来实现，通过编译器把分支预测让开发者无感，但是在线程有限的 SIMD 下性能的提升可能会是个难题。

线程在 Warp-base SIMD 可以执行不同的分支

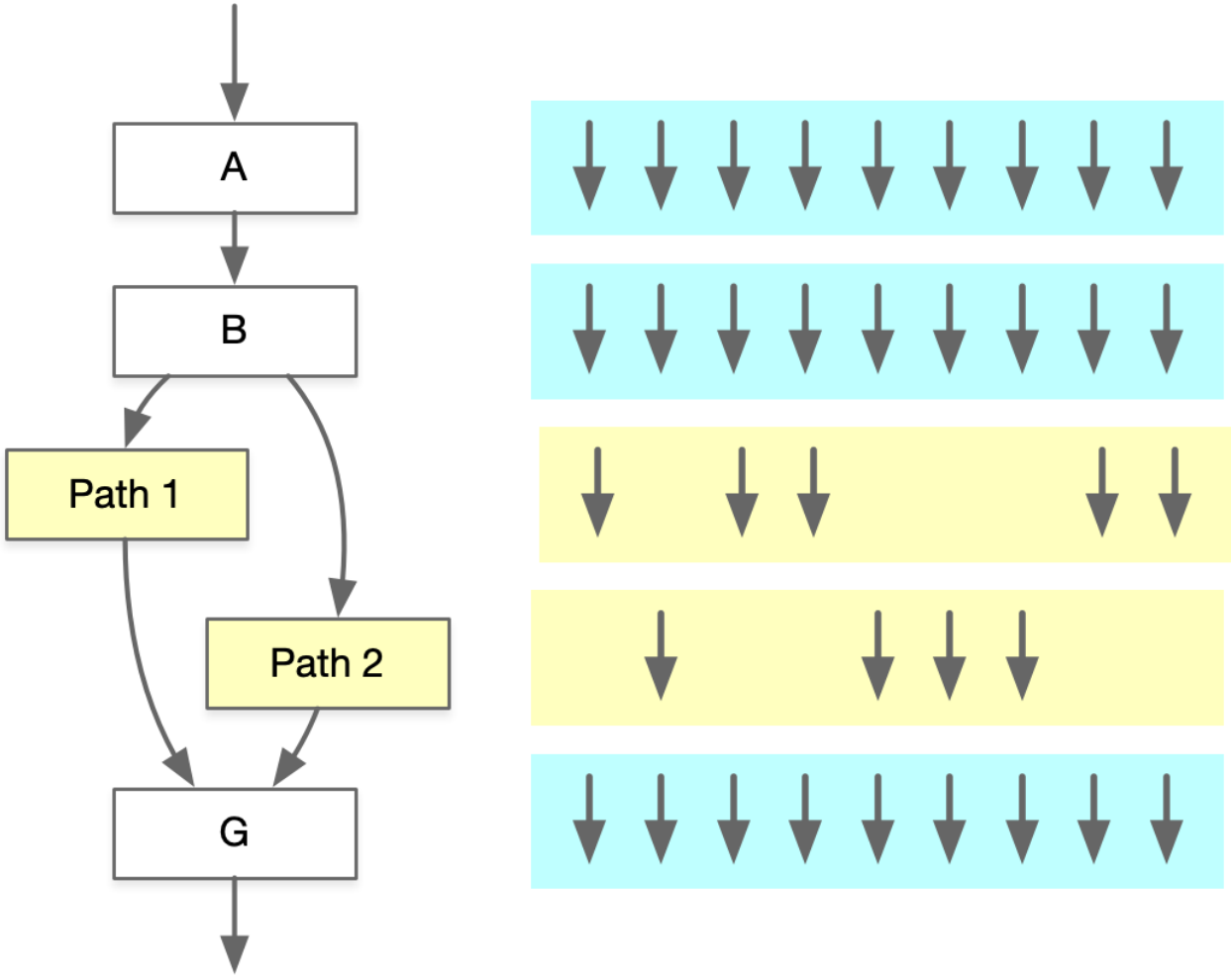
- 每个线程都可以执行带条件控制流指令（Conditional Control Flow Instructions）
- 不同线程间可以分别执行不同的控制流路径（Different Control Flow Paths）





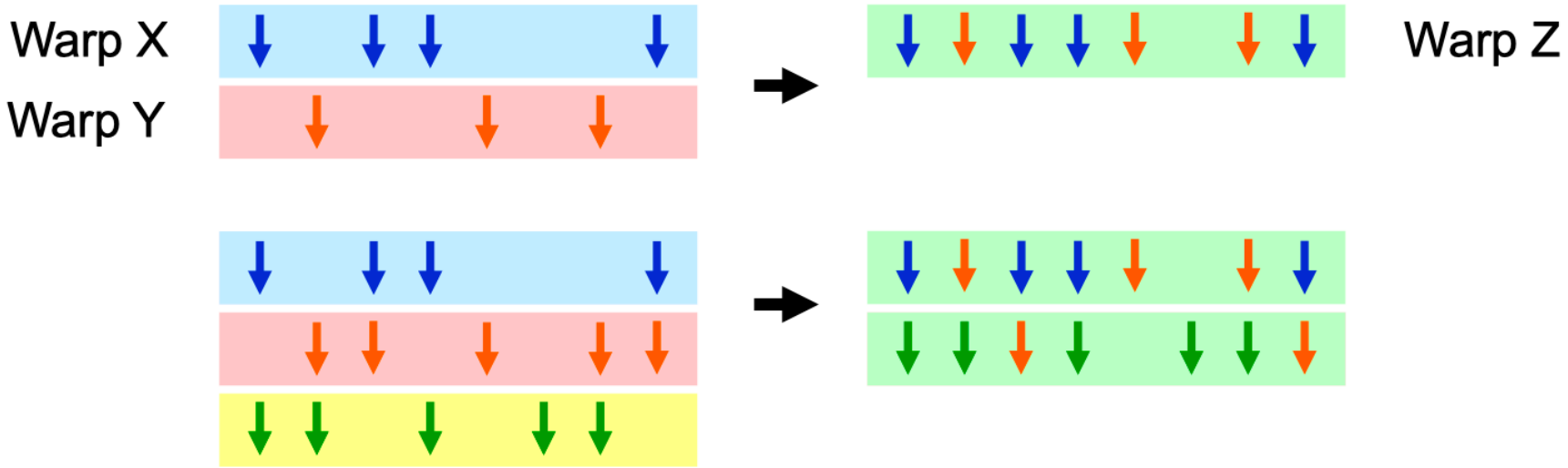
SIMT 的控制流问题

1. 使用 SIMD 流水线来节省控制逻辑上的面积：将 Scalar 线程放在 Warps 里面
2. 当 Warp 内部的线程分支到不同的执行路径时，就会发生分支执行冲突



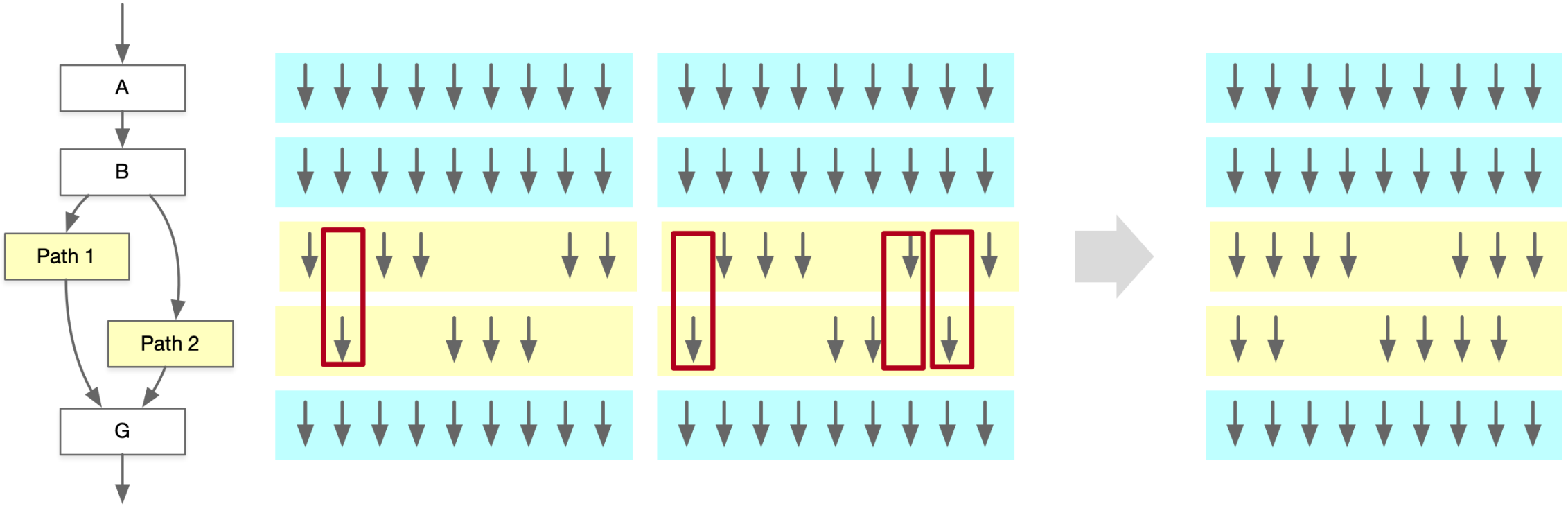
动态Warp Formation/Merging

- 在分支后动态合并执行相同指令的线程：
  - 从正在等待的 Warps 中形成新 Warps（Form new warps from warps that are waiting）
  - 分支下每条路径线程用于创建新 Warp（Enough threads branching to each path enables creation new warps）

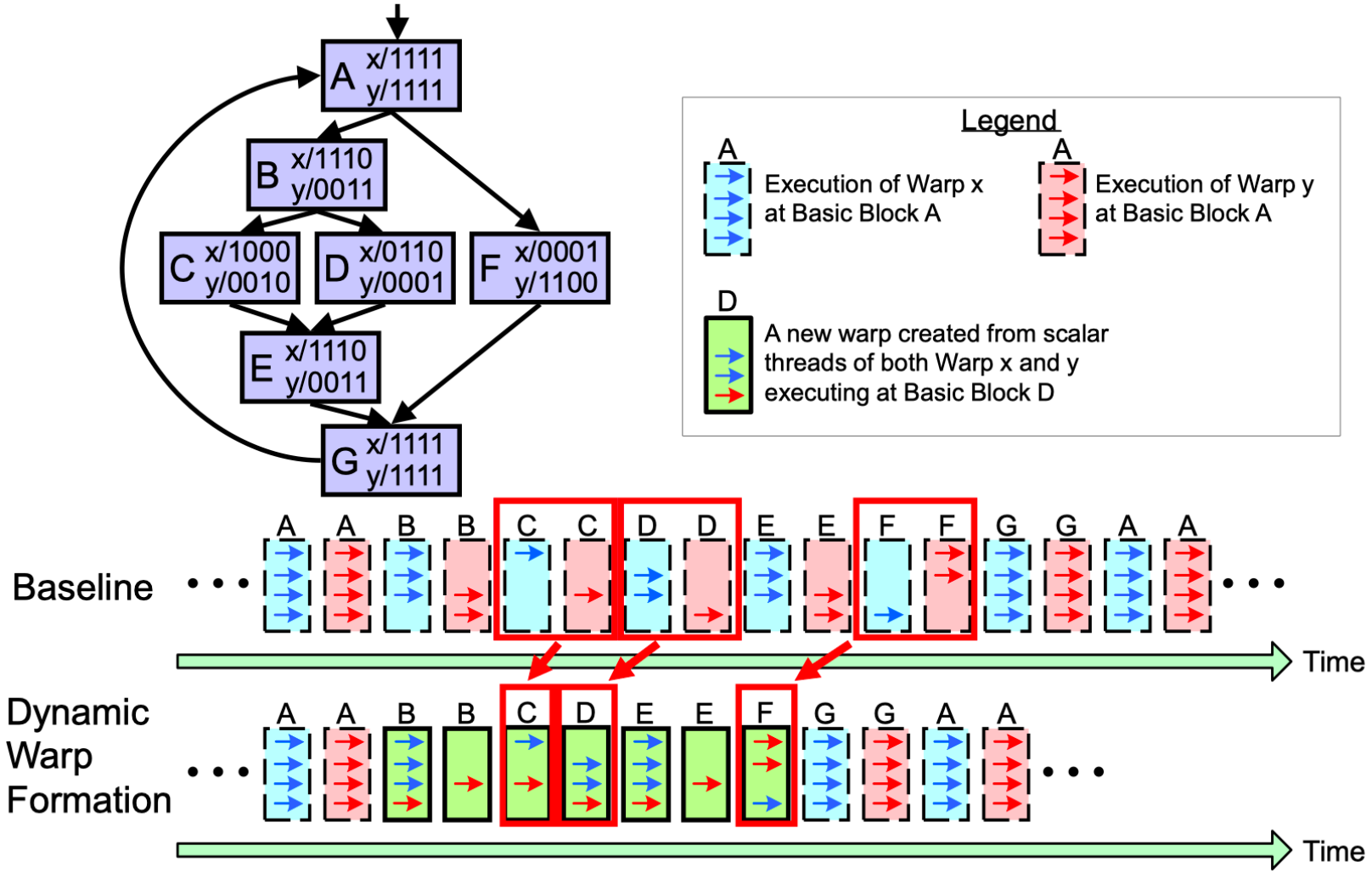


动态 Warp Formation/Merging

- 在分支后动态合并执行相同指令的线程



Dynamic Warp Formation Example



借鉴点 4

交互方式

借鉴点4：提供便利的host、device交互方式

- CUDA 中有诸多实现机制与 SIMT、SIMD、DSA 的硬件架构本身并没有太多关联关系。CUDA 中的所有特性也不是 SIMT 架构独有的。因此不存在技术上选择 SIMT、SIMD、DSA 等问题。
- 如 CUDA Runtime 提供 host 和 device 的 C++ 交互方式，如寒武纪 BANG C 语言在这个层面就参考了 CUDA。

```
1
2    for (int i = 0; i<10000; ++i){
3        C[i] = A[i] + B[i];
4    }
5
```

```
5
6    __global__ void KernelFuncion(...) {
7        int tid = blockDim.x * blockIdx.x + thredIdx.x;
8        int varA = a[tid];
9        int varB = b[tid];
10       varC[tid] = varA + varB;
11   }
```

借鉴点4：开发编程易用性

- 初阶用户而言，CUDA 易用性是极致的，入门开发者任意写一个简单的 Kernel，就能够获得比 CPU 高 5~10 倍的峰值性能。
- DSA 硬件架构，因为在流水和指令使用上缺乏完备、隐式的支持，指令流水的支持需要开发者通过手工掩盖、切块等其他优化思想补齐这部分性能，而使用底层指令则会让用户在写出正确的 Kernel 花费更多的时间。

思考

- **流水隐藏：**架构层面隐藏流水编排，提出一个形式上与 SPMD 没有关系的编程模式，而且易用性堪比 CUDA 的软件是可能得。反过来核心问题没解决，提出形式上与 CUDA 类似的编程模型也仍然会有易用性问题，开发者很难有一个足够好的初始性能。
- **软硬件架构：**对于 DSA 架构而言，建立一套开放的软硬件架构，联合其他 DSA 架构来对抗 CUDA 生态；另外一方面，对于 DSA 而言，需要明确面向不同层级开发者的易用机会和软件开发形态。

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