AI编译器-后端优化

指令与存储优化

ZOMI

Talk Overview

I. AI 编译器后端优化

- 后端优化概念
- 算子计算与调度
- 算子调度优化
- Auto-Tuning
- Polyhedral

算子调度优化方法

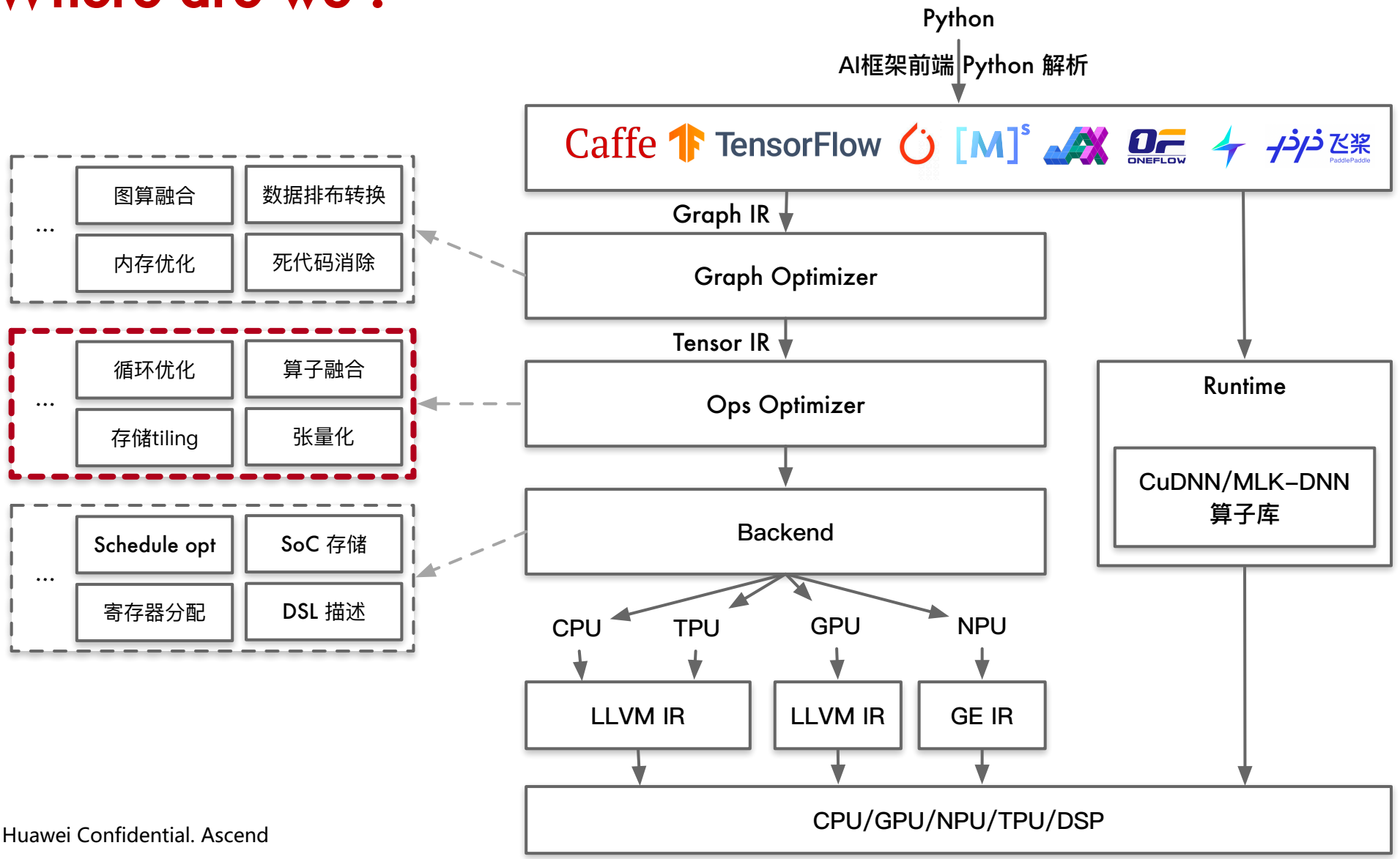
- 循环展开（Loop Unrolling）
- 循环分块（loop tiling）
- 循环重排（loop Reorder）
- 循环融合（loop Fusion）
- 循环拆分（loop Split）
- 向量化（Vector）
- 张量化（Tensor）
- 访存延迟（ Latency Hiding ）
- 存储分配（Memory Allocation）

循环优化（Loop Optimization）

指令优化（Instructions Optimization）

存储优化（Memory Optimization）

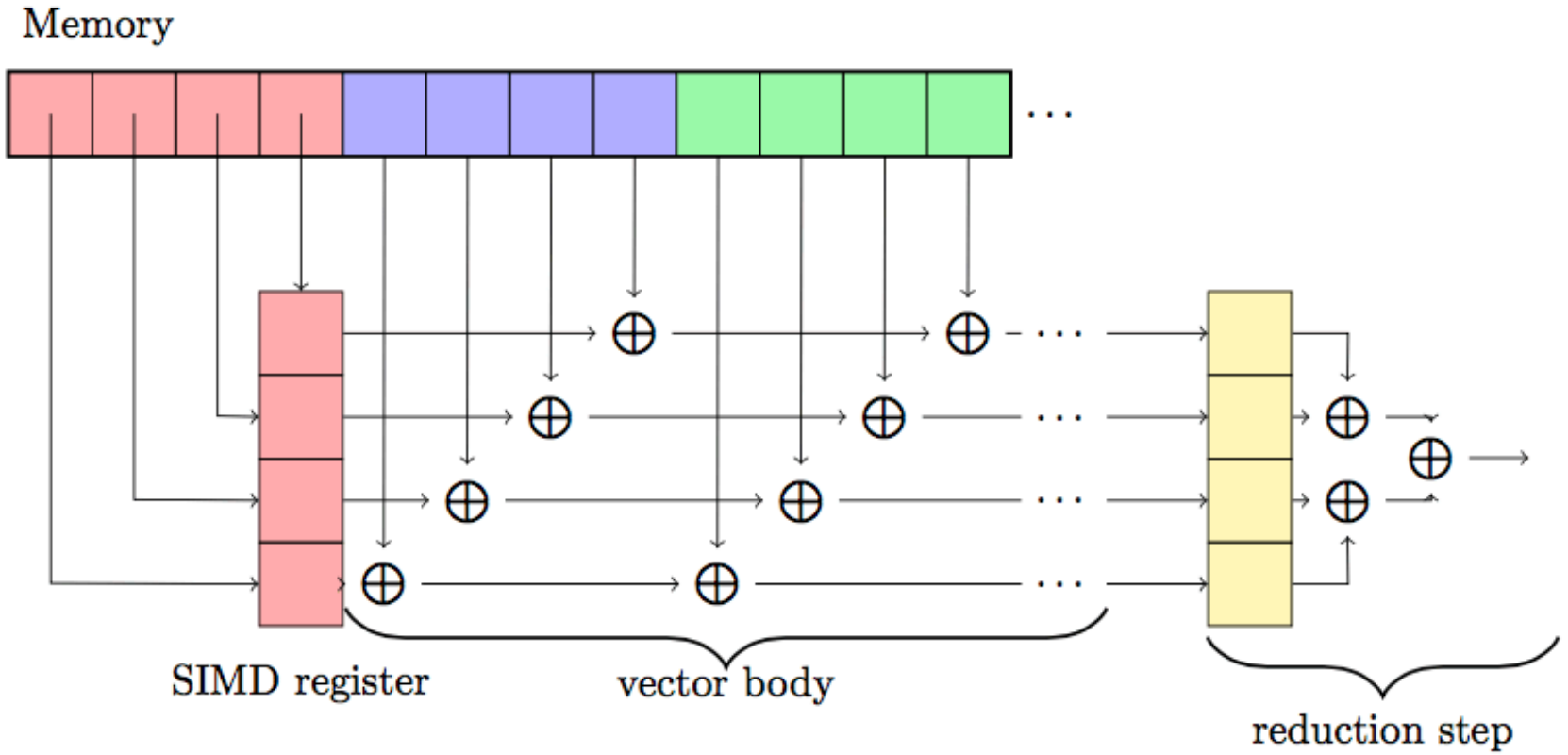
Where are we ?



指令优化
Instructions Optimization

向量化 Vectorization

- Vector reduction. The SIMD register holds the current value of vec_sum. In the reduction step, vec_sum is converted to a single sum value.



向量化 Vectorization

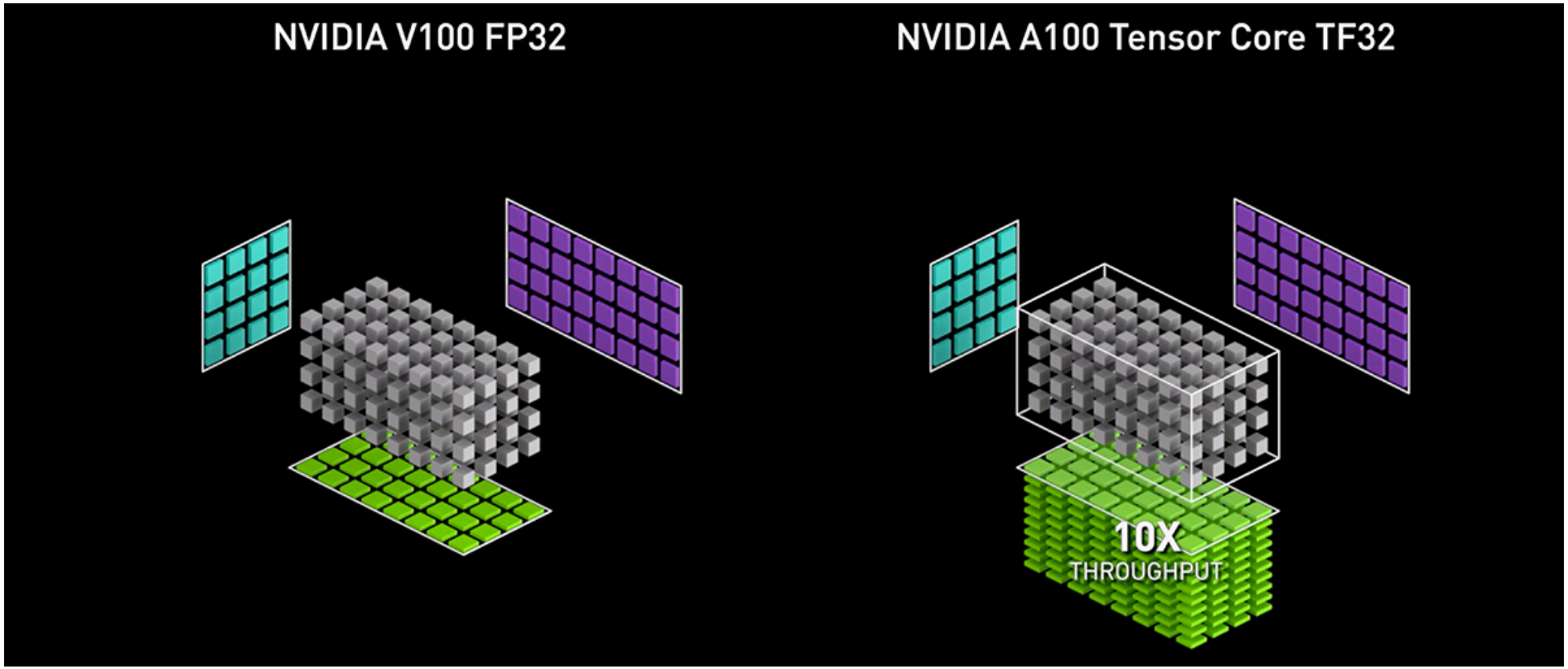
```
2    double sum = 0.0;
3    for (int i = 0; i < n; i++) {
4        sum += a[i];
5    }
6
7    # Change TO
8
9    double<4> vec_sum = { 0.0, 0.0, 0.0, 0.0 };
10   for (int i = 0; i < n; i+=4) {
11       double<4> a_val = load<4>(a + i);
12       vec_sum = add<4>(a, vec_sum);
13   }
14   sum = 0.0;
15   for (int i = 0; i < 4; i++) {
16       sum += vec_sum[i];
17   }
```

张量化 Tensorization

- Volta 架构中，一个SM由8个FP64 Cuda Cores，16个INT32 Cuda Core，16个FP32 Cuda Core，和128个Tensor Core组成，一共有4个SM。

张量化 Tensorization

- Volta 架构中，一个SM由8个FP64 Cuda Cores，16个INT32 Cuda Core，16个FP32 Cuda Core，和128个Tensor Core组成，一共有4个SM。



张量化 Tensorization

- 主流CPU/GPU硬件厂商都提供了专门用于张量化计算的张量指令，如英伟达的张量核指令、英特尔的VN。利用张量指令的一种方法是调用硬件厂商提供的算子库，如英伟达的cuBLAS和cuDNN，以及英特尔的oneDNN等。
- 然而，当模型中出现新的算子或需要进一步提高性能时，这种方法的局限性便显露无遗。

张量化 Tensorization

1. 新的硬件体系带来了超越向量运算的新指令集，调度必须使用这些指令才能从加速中获益
2. 张量计算基元的输入是多维的，具有固定的或可变的长度，并指示不同的数据布局
3. 新的 AI 加速器正以它们自己的张量指令出现

TVM：将硬件指令的接口与调度分开，生成硬件接口

- 利用 Tensor 表达式表示高层运算，还能表示硬件底层特性
- Tensorization 提供调度表与特定的硬件原语，便于AI编译器扩展支持新硬件架构
- AI 编译器后端将Tensor操作映射为硬件实现或高度优化的手工微内核，从而显著提高性能

存储优化
Memory Optimization

访存延迟 Latency Hiding

- 延迟隐藏（Latency Hiding）是指将内存操作与计算重叠，最大限度地提高内存和计算资源利用率的过程。

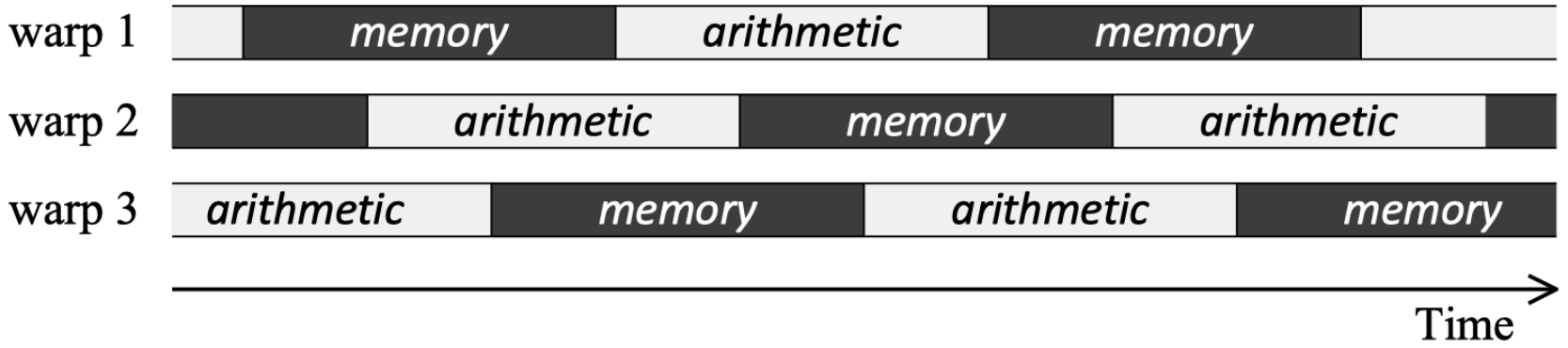


Figure 3.2: There are 3 concurrent warps, but only 1.5 concurrent memory access instructions on average.

访存延迟 Latency Hiding

CPU：

- 延迟隐藏可以通过多线程，或者硬件隐式数据预取实现

GPU：

- 依赖于 Wrap Schedule 对多线程的管理调度和上下文切换实现

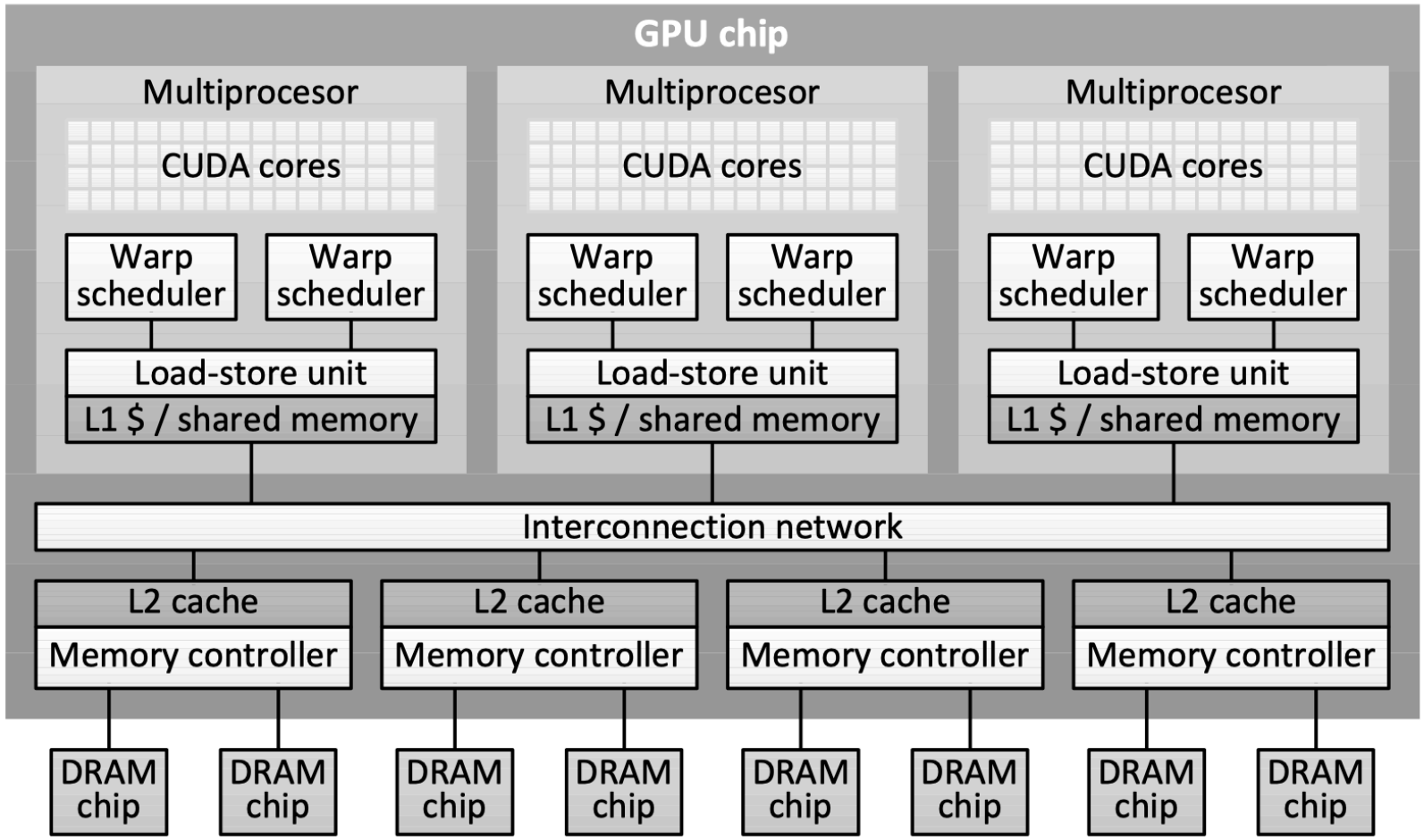
NPU/TPU：

- 采用解耦访问/执行（Decoupled Access/Execute，DAE）架构

访存延迟 Latency Hiding on GPUs

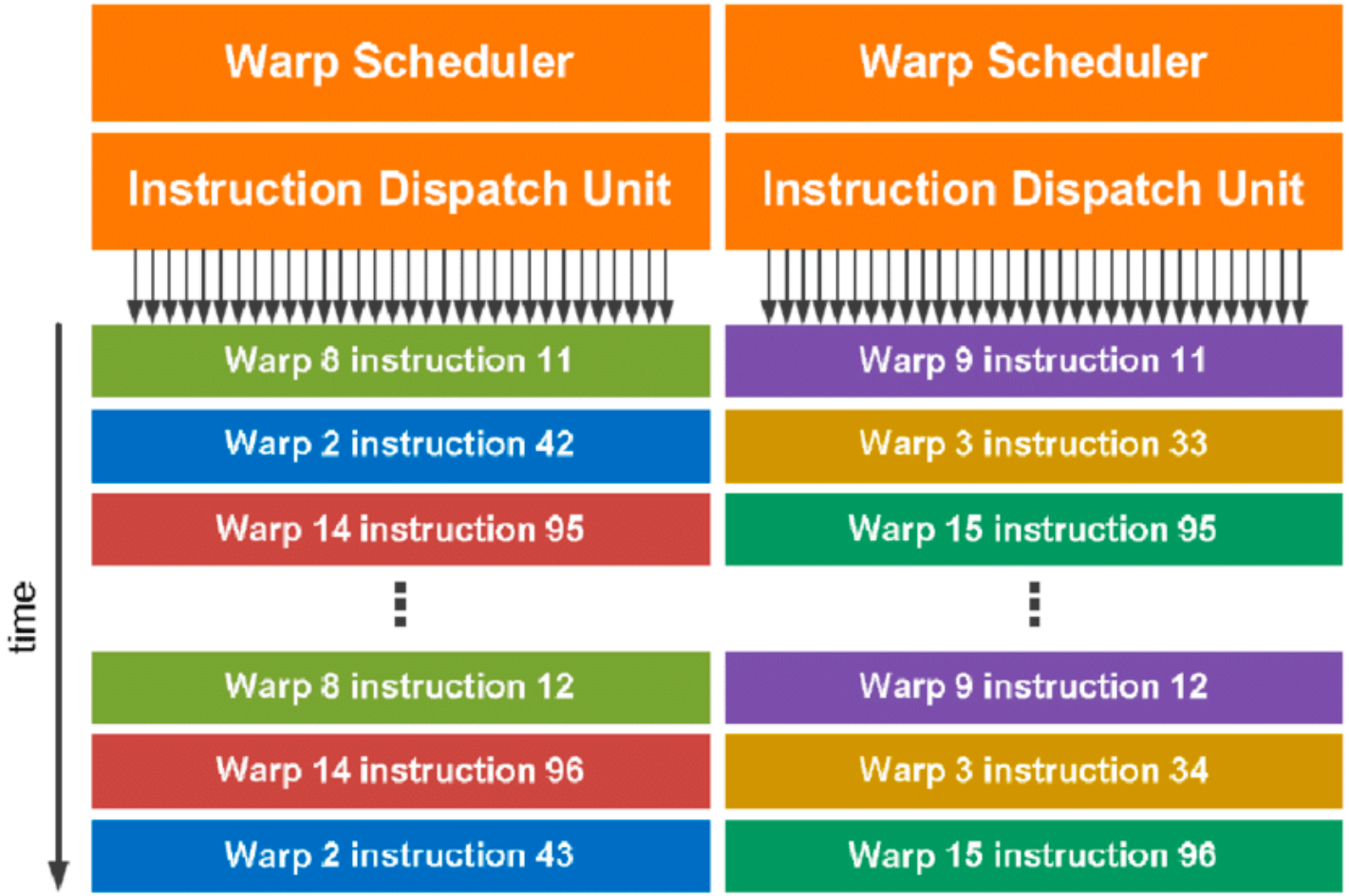
GPU core, a.k.a. SM, resou rces (typical values)

- Maximum number of warps per SM
- Maximum number of blocks per SM
- Shared memory usage
- Register usage

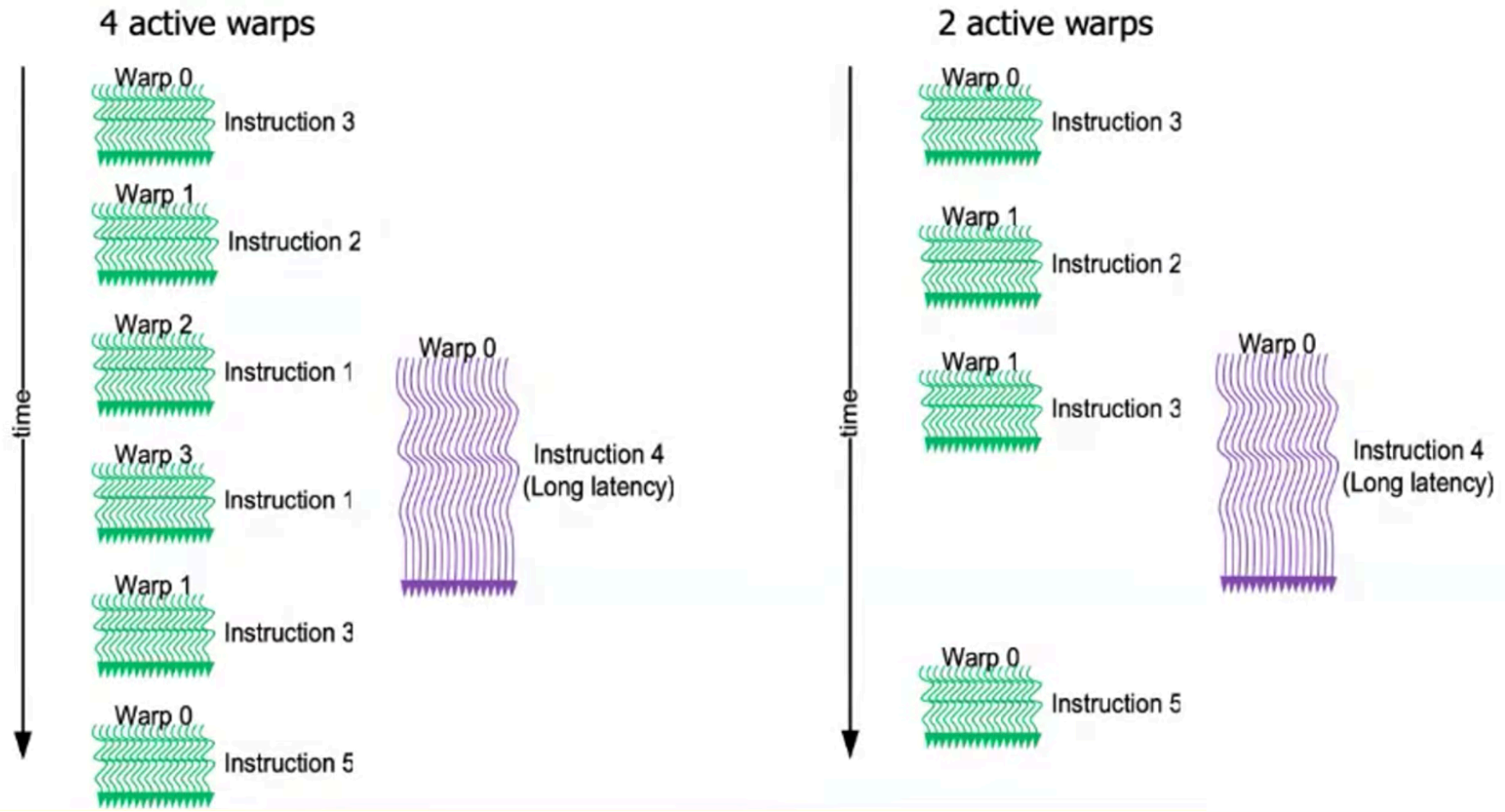


访存延迟 Latency Hiding on GPUs

- Warp：A set of threads that ex ecute the same instruction (on different data elements)
- Each scheduler issues 1 instructi on from one warp per cycle, if i t has a warp ready to execute.



访存延迟 Latency Hiding on GPUs



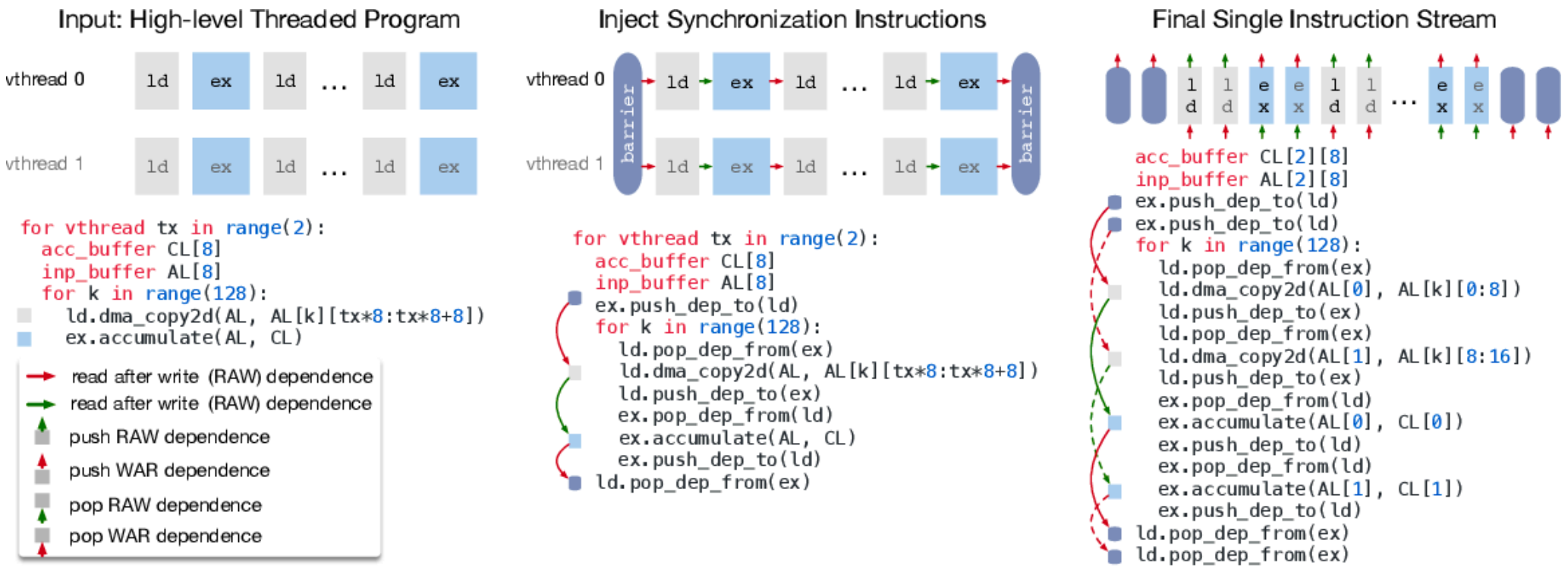
Decoupled Access/Execute Architecture Processor

解耦访问/执行架构：

- 将内存访问单元（MAU）与管道分离。执行处理器 (EP) 由直接寄存器访问 (DRA) 和直接缓存访问 (DCA) 组成，可用于管理寄存器、缓存和内存之间的数据传输。

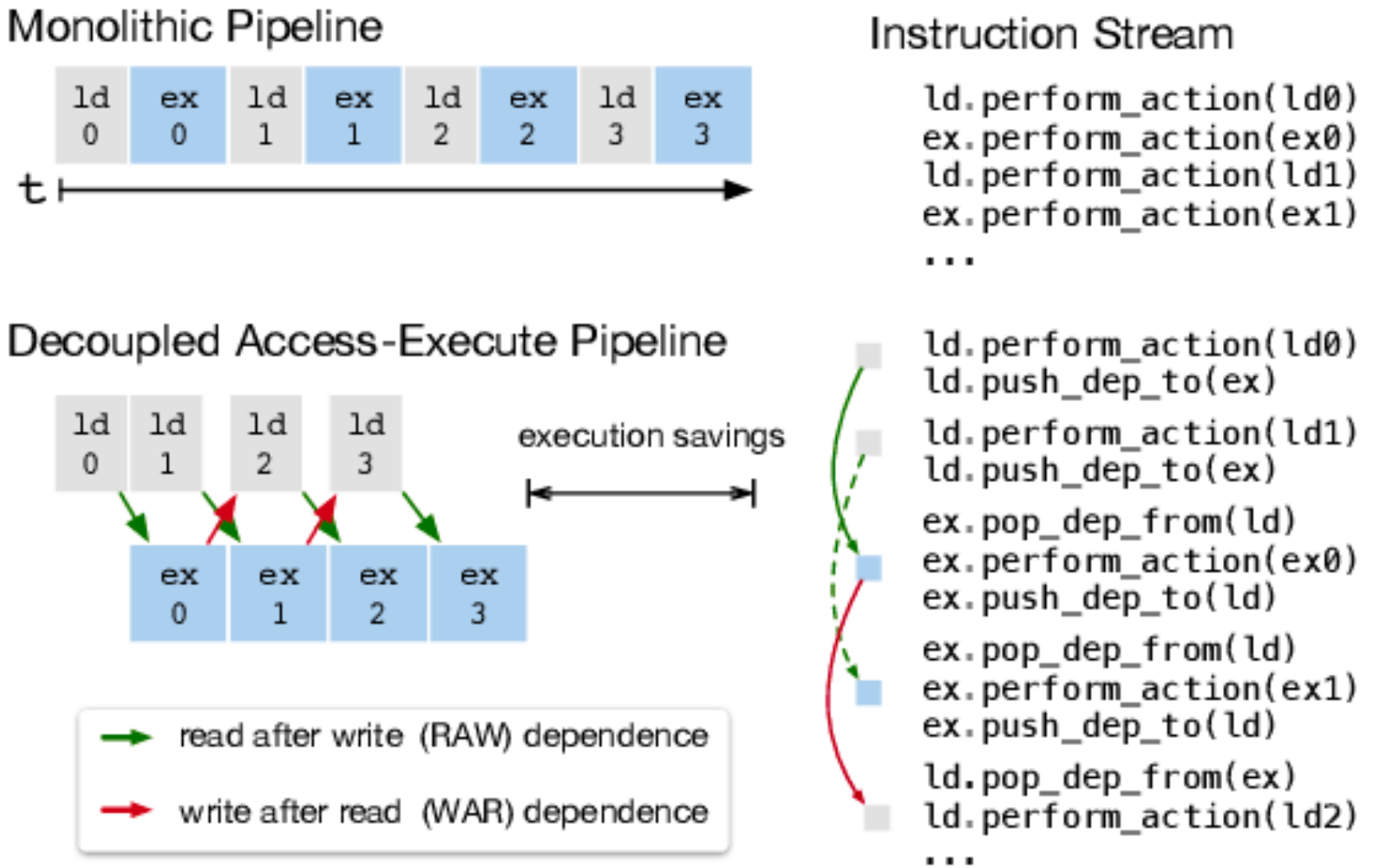
访存延迟 Latency Hiding on XPU

- TVM 将虚拟线程并行程序转换为单个指令流；这个流包含显式的低级同步，硬件可以解释这些同步，以恢复管道并行性，需要隐藏内存访问延迟。



访存延迟 Latency Hiding on XPU

- 在硬件中执行解耦访问，允许内存和计算重叠。执行正确性通过低级别的同步来实现，同步的形式是依赖Token入队/出列操作。



内存分配 Memory Allocation

- 局部变量
- 全局变量
- 堆变量

内存分配 Memory Allocation

- 局部变量：开发者定义普通变量时编译器在内存中的栈空间为其分配一段内存

```
2    int i, j zomi[10];
```

内存分配 Memory Allocation

- 全局变量（静态变量和全局变量）：编译器在内存中的静态存储区分配空间

```
2    int x, y; // Global 全局变量
3
4    int main(){
5        static int n, m; // static 静态变量
6    }
```

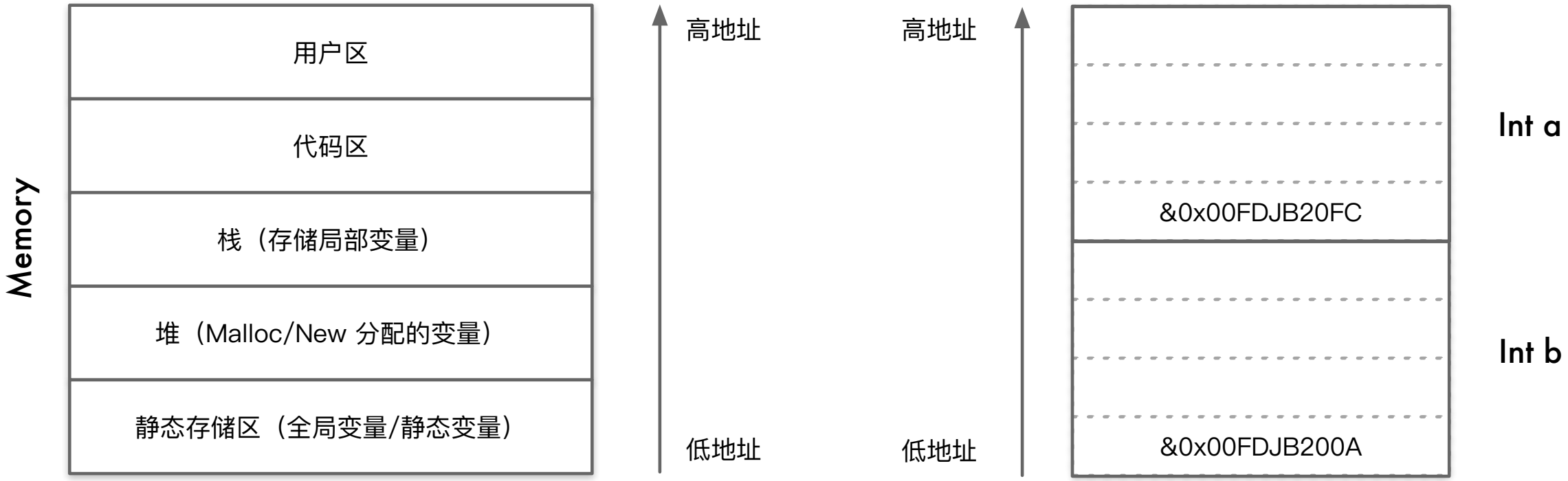
内存分配 Memory Allocation

- 堆变量：开发者用malloc或new在堆上申请一段内存空间

```
8    int *x = new int[10];
9    int* ptr = (int*) malloc(sizeof(int));
```

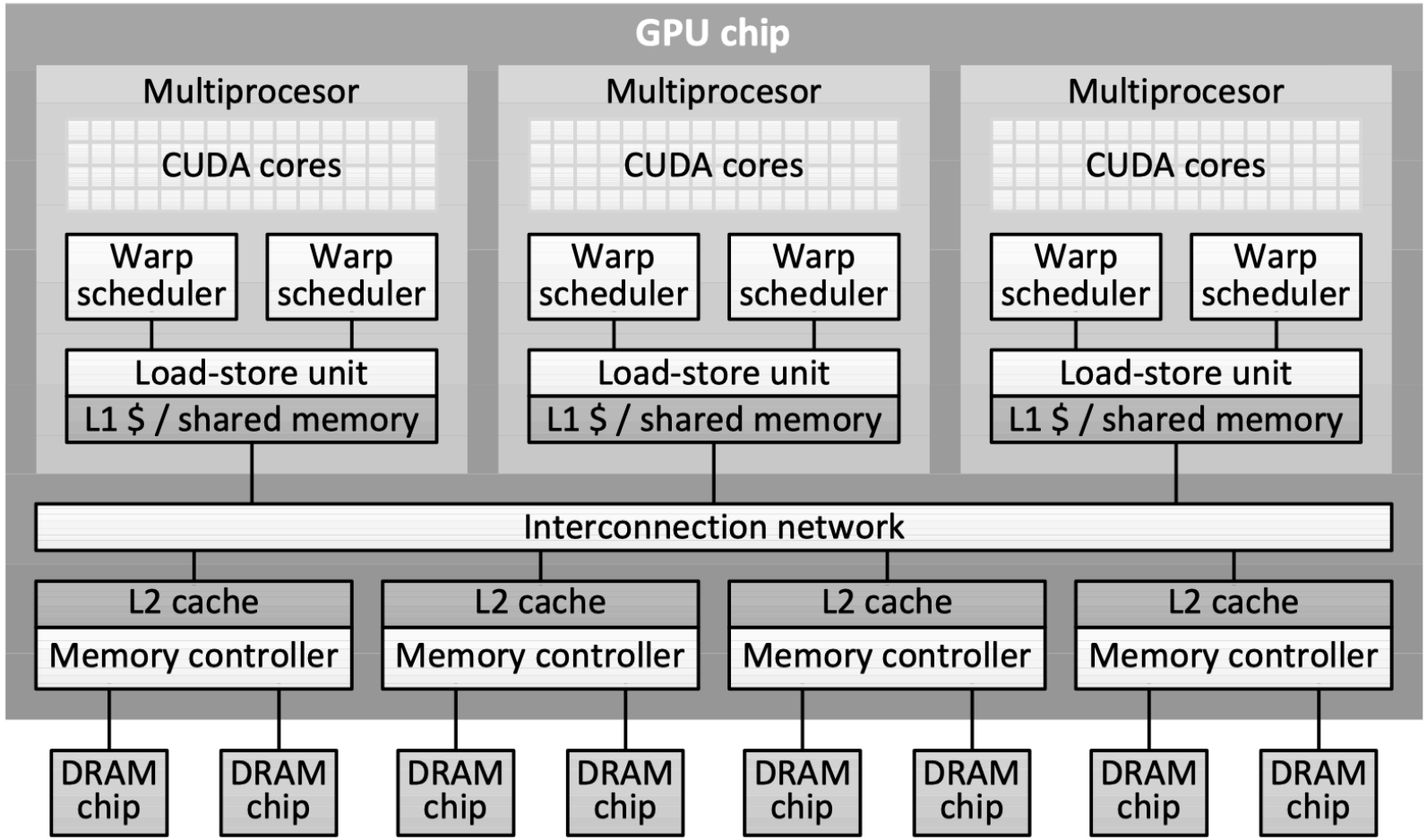
内存分配 Memory Allocation on CPU

- 传统编译器通过将内存逻辑划分为不同区段来提供程序员访问内存的权限，而每段空间都有各自的大小，一旦开发者在使用过程中将该大小耗尽，就会出现内存不足错误。



内存分配 Memory Allocation on GPU

- Shared Memory
- Local Memory



Inference

- Li, Mingzhen, et al. "The deep learning compiler: A comprehensive survey." *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2020): 708-727.
- Ning, Chao, and Fengqi You. "Optimization under uncertainty in the era of big data and deep learning: When machine learning meets mathematical programming." *Co mputers & Chemical Engineering* 125 (2019): 434-448.
- Xu, Zhiying, et al. "ALT: Breaking the Wall between Graph and Operator Level Optimizations for Deep Learning Compilation." *arXiv preprint arXiv:2210.12415* (2022).
- Baghdadi, Riyadh, et al. "A deep learning based cost model for automatic code optimization." *Proceedings of Machine Learning and Systems* 3 (2021): 181-193.
- Chen, Tianqi, et al. "TVM: end-to-end optimization stack for deep learning." *arXiv preprint arXiv:1802.04799* 11.2018 (2018): 20.
- Purkayastha, Arnab A., et al. "LLVM-based automation of memory decoupling for OpenCL applications on FPGAs." *Microprocessors and Microsystems* 72 (2020): 102909.
- Loop Optimizations: how does the compiler do it? https://johnysswlab.com/loop-optimizations-how-does-the-compiler-do-it/
- Loop Optimizations: taking matters into your hands https://johnysswlab.com/loop-optimizations-taking-matters-into-your-hands/
- Understanding Latency Hiding on GPUs - UC Berkeley EECS. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.pdf