上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

CS4302-01
Parallel and Distributed Computing

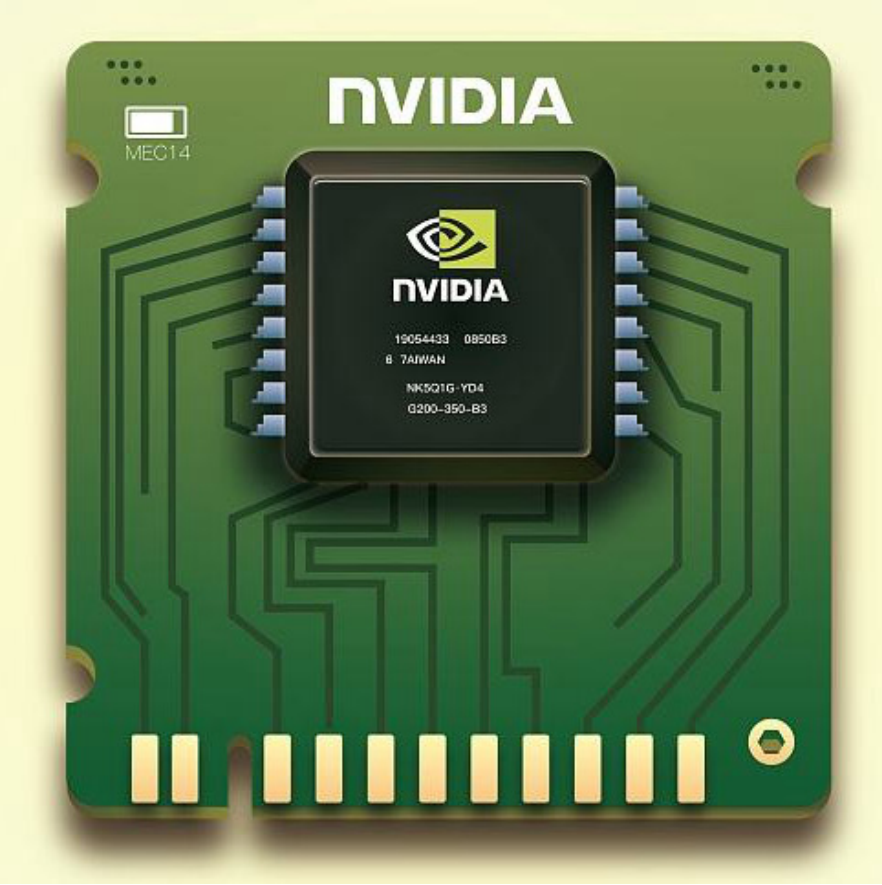
Lecture 5 GPU Architecture

Zhuoran Song

2023/10/10

What is GPU?

GPU全称是Graphics Processing Unit，图形处理单元。它的功能最初与名字一致，是专门用于绘制图像和处理图元数据的特定芯片

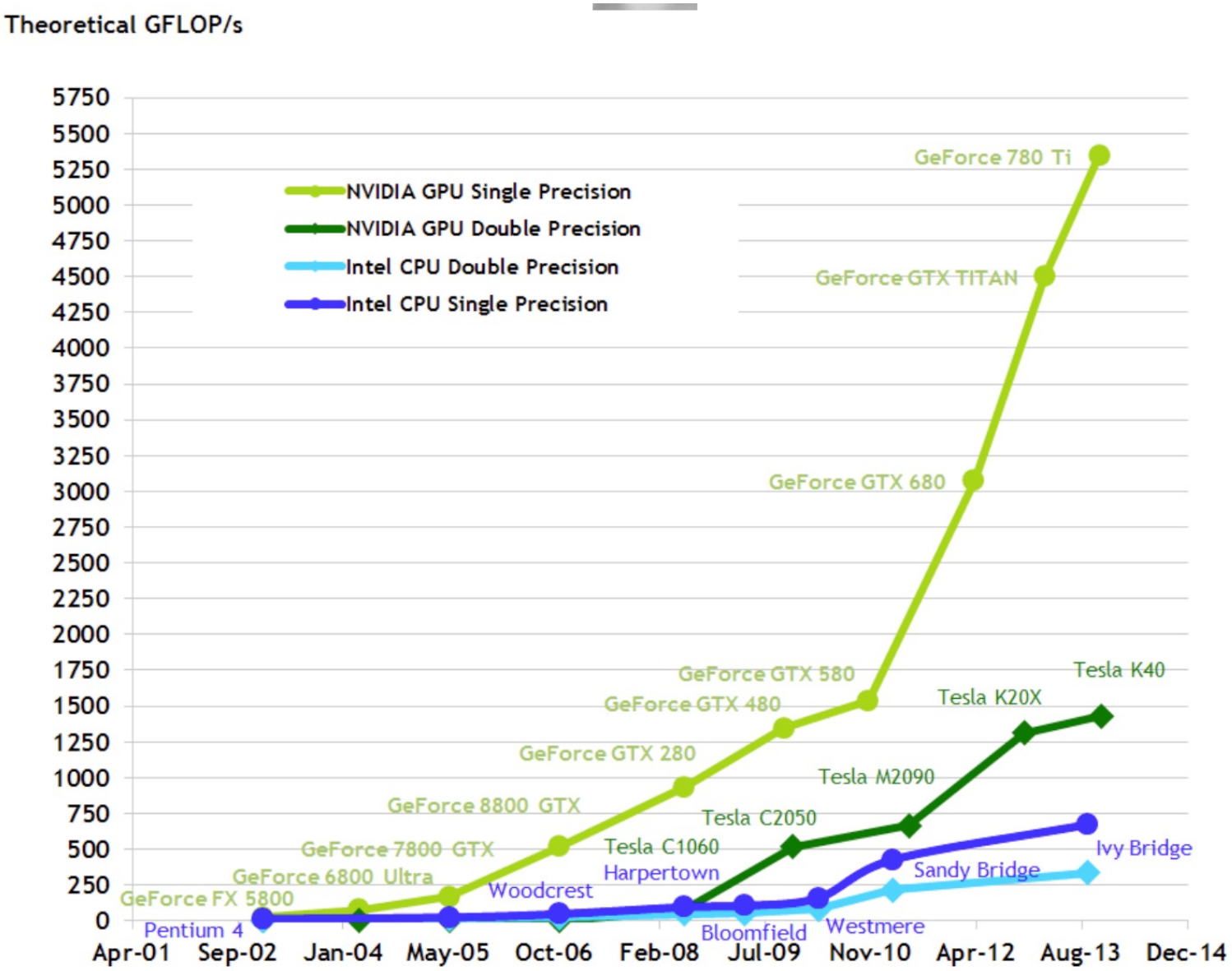


GPU发展历史

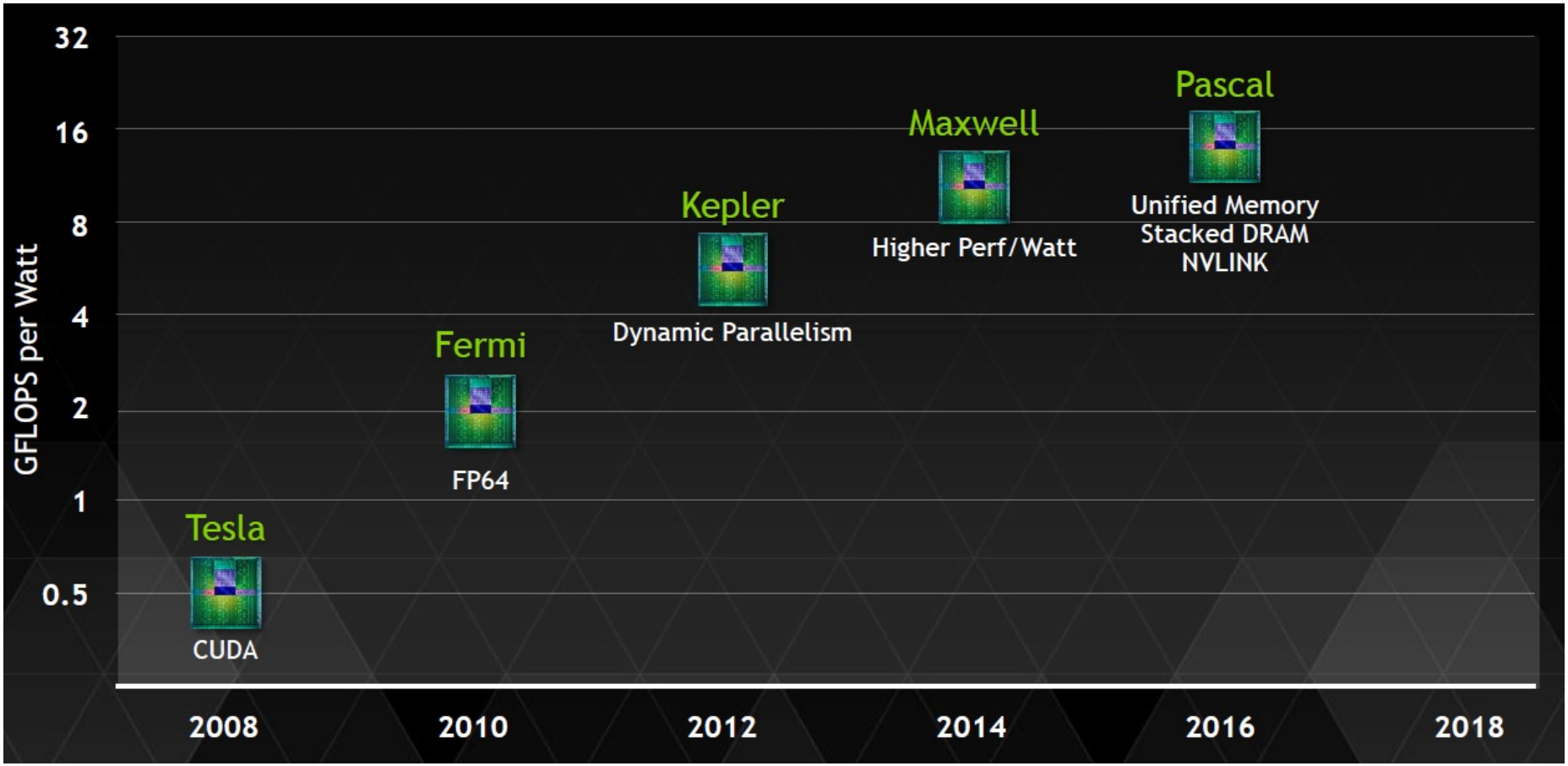
- 1995 — NV1
- 1997 — Riva 128 (NV3), DX3
- 1998 — Riva TNT (NV4), DX5
- 2001 – GeForce 3
- 2016 – GeForceGTX 1060 6GB 光追
- 2022 – H100 2TFLOPs

What is the difference between FLOPs and FLOPS?
FLOPs:Floating Point Operations
FLOPS:Floating Point Operations Per Second

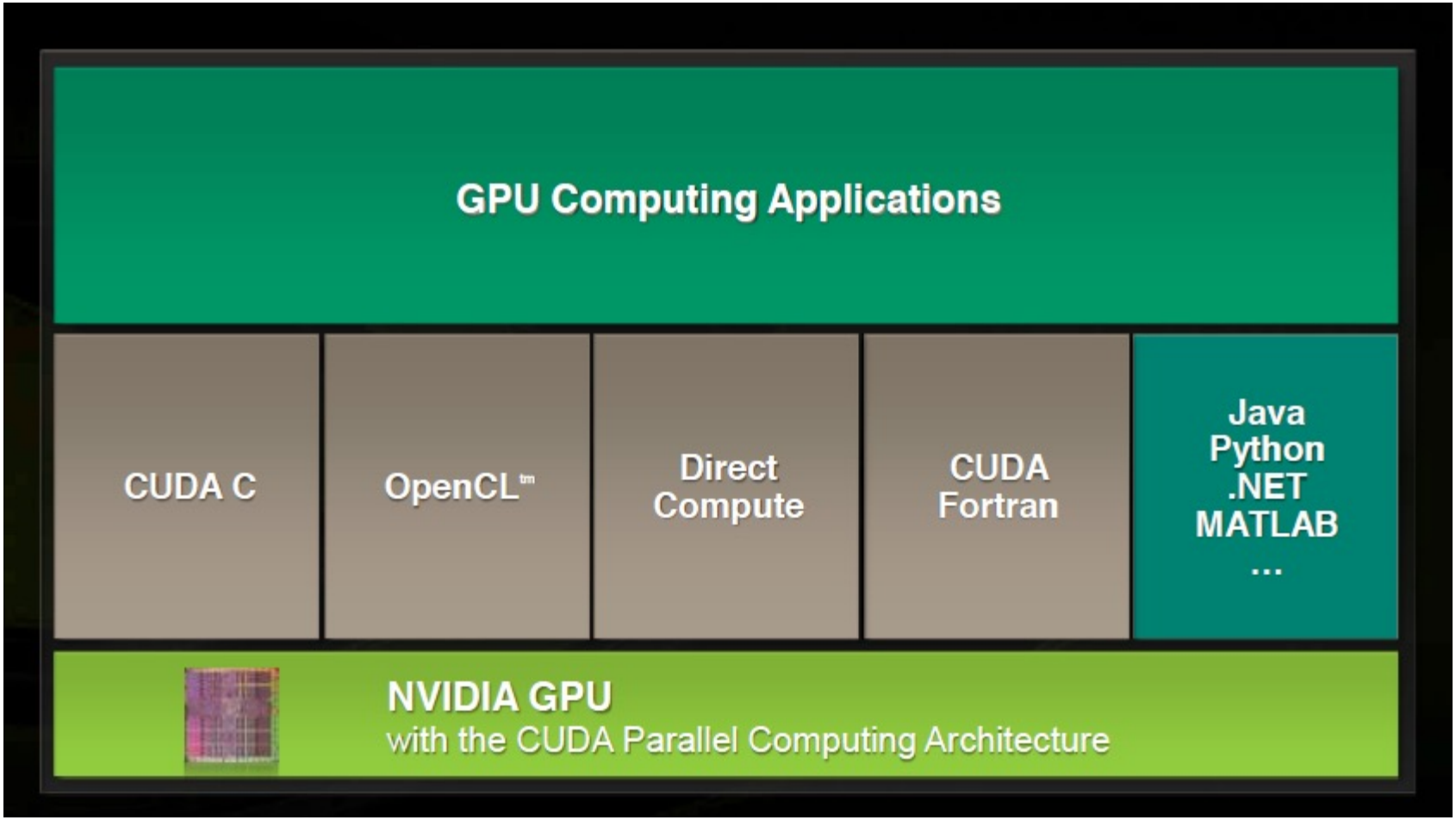
GPU发展趋势


Theoretical GFLOP/s
5750
5500
5250
5000
4750
4500
4250
4000
3750
3500
3250
3000
2750
2500
2250
2000
1750
1500
1250
1000
750
500
250
0
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Double Precision
Intel CPU Single Precision
GeForce 780 Ti
GeForce GTX TITAN
GeForce GTX 680
GeForce GTX 580
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 Ultra
GeForce FX 5800
Tesla K40
Tesla K20X
Tesla M2090
Tesla C2050
Tesla C1060
Harpertown
Woodcrest
Pentium 4
Bloomfield
Westmere
Sandy Bridge
Ivy Bridge
Apr-01
Sep-02
Jan-04
May-05
Oct-06
Feb-08
Jul-09
Nov-10
Apr-12
Aug-13
Dec-14

GPU发展趋势


32
16
8
4
2
1
0.5
GFLOPS per Watt
Tesla
CUDA
Fermi
FP64
Kepler
Dynamic Parallelism
Maxwell
Higher Perf/Watt
Pascal
Unified Memory
Stacked DRAM
NVLINK
2008
2010
2012
2014
2016
2018

GPGPU Platforms



初始CUDA代码

```
#include <stdio.h>

void cpu()
{
    printf("hello cpu!\n");
}

__global__ void gpu()
{
    printf("hello gpu!\n");
}

int main()
{
    cpu();
    gpu<<<1,1>>>();
    cudaDeviceSynchronize();
}
```

声明**1**个线程
打印**hello gpu**

Fermi Architecture

拥有16个SM

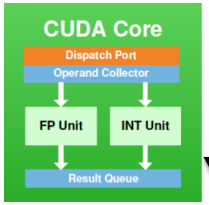
每个SM：
共32个Core
16组加载存储单元（LD/ST）
4个特殊函数单元（SFU）
Warp编排器（Warp Scheduler）
分发单元（Dispatch Unit）

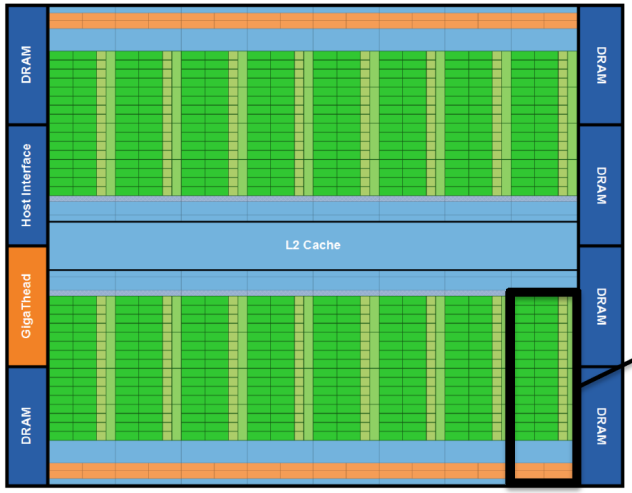
每个Core：
1个FPU（浮点数单元）
1个ALU（逻辑运算单元）

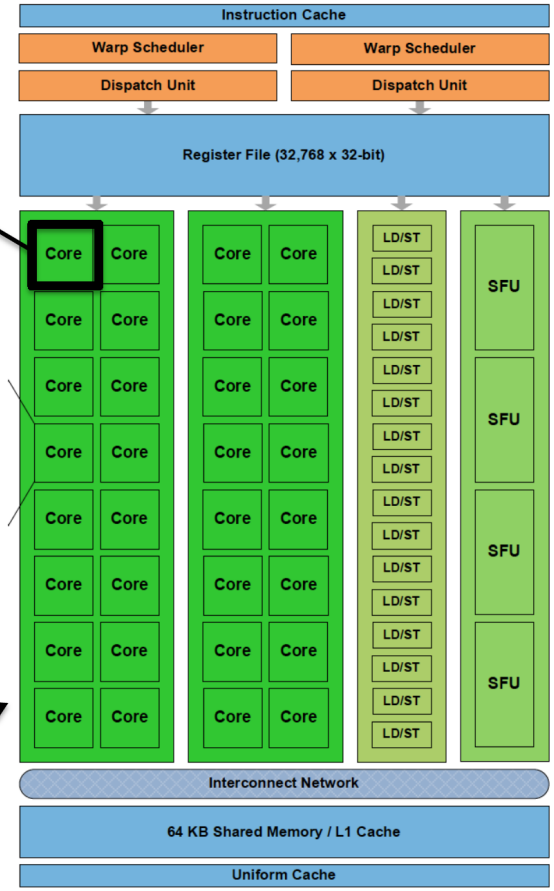
Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU).

More cores per multiprocessors and faster arithmetic operations

Memory Protection Support: Memory are protected by a Single - Error Correct Double - Error Detect (SECDED) ECC code

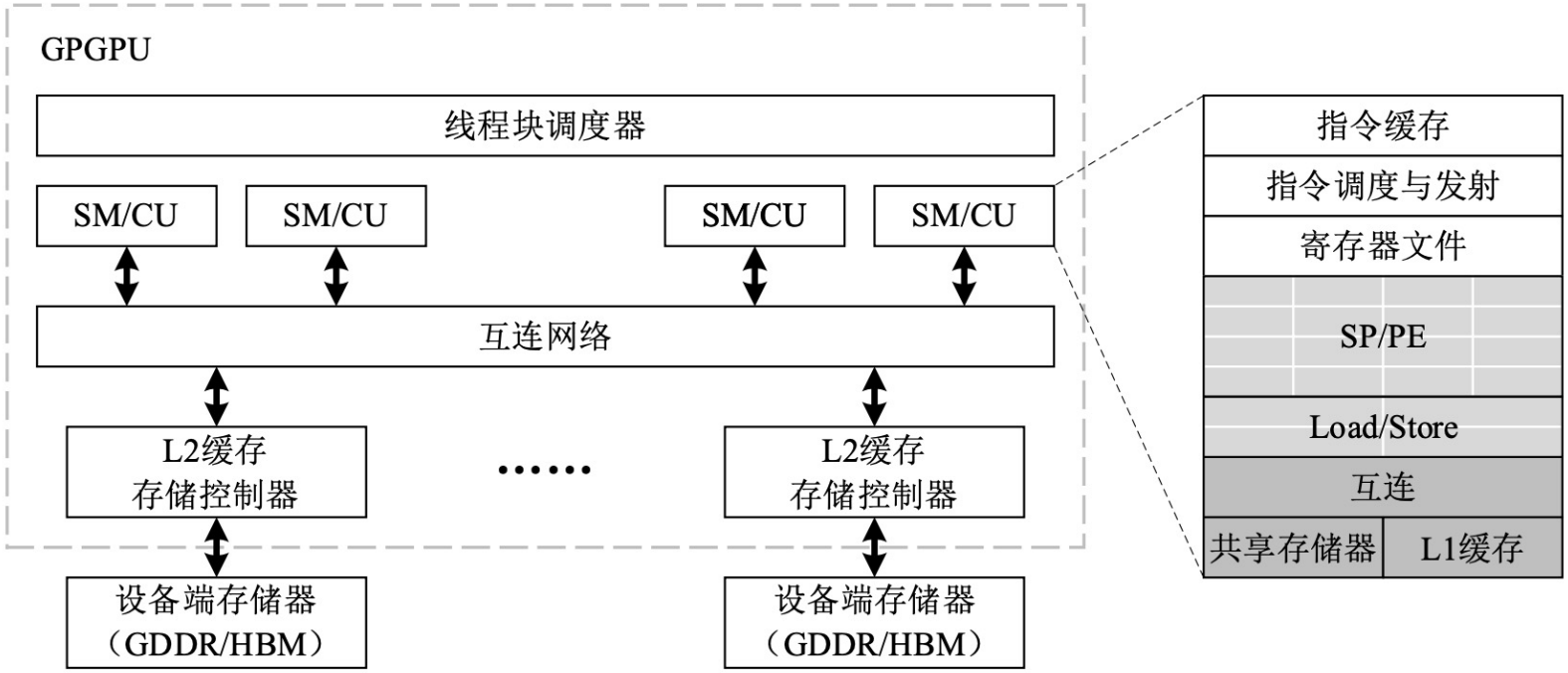






GPGPU Architecture

- 可编程流处理器（SM，streaming multiprocessor）
  - **流处理器（*SP*，*streaming processor*），*CUDA*核心**
  - **指令缓存**
  - **指令调度器**
  - **寄存器文件**
  - *L1 cache/shared memory*

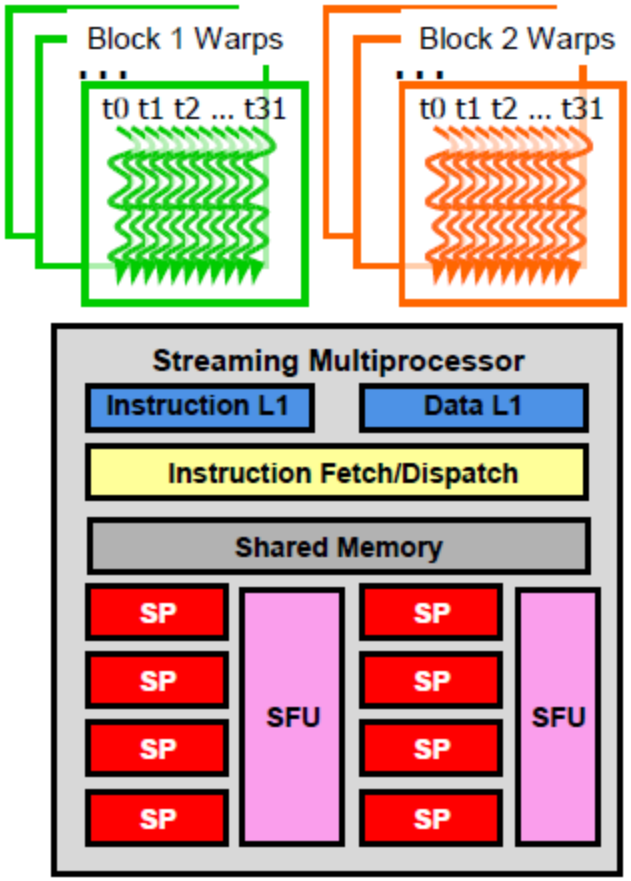


SM Architecture

- Each Thread Blocks is divided in 32-thread Warps
  - *This is an implementation decision, not*
  - *part of the CUDA programming model*

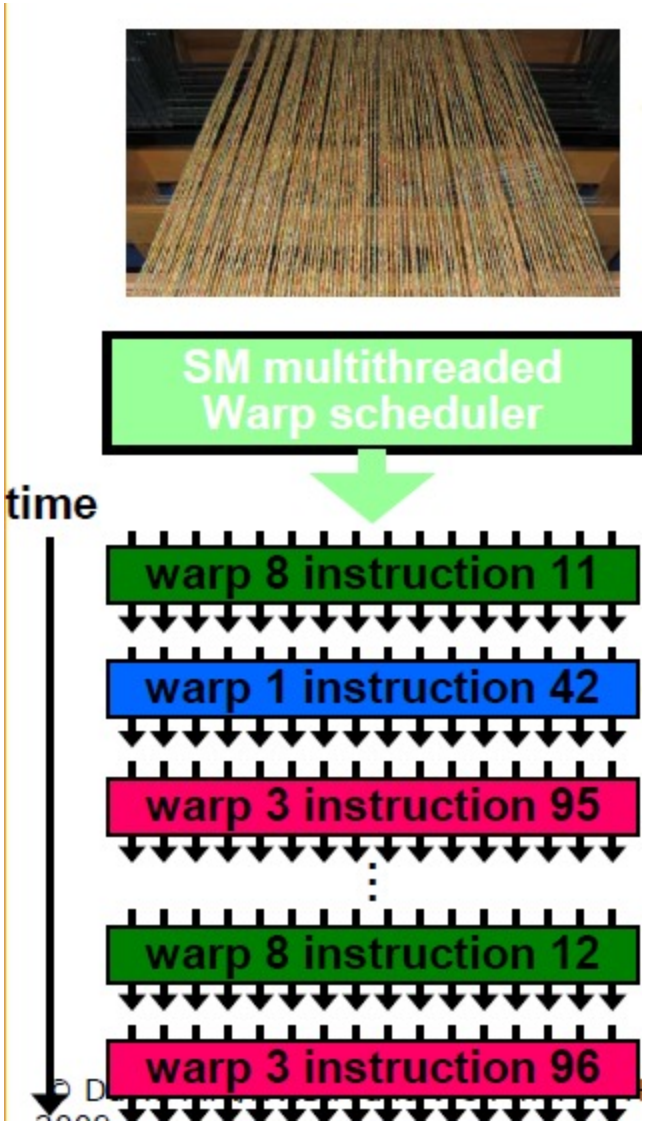
- Warps are scheduling units in SM

- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?
  - *Each Block is divided into 256/32 = 8 Warps*
  - *There are 8 * 3 = 24 Warps*
  - *At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.*

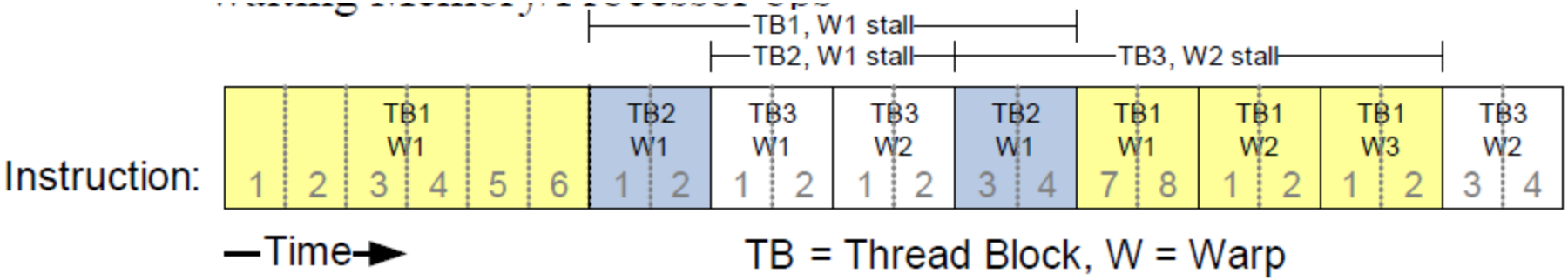


SM Architecture

- SM hardware implements zero overhead Warp scheduling
  - *Warps whose next instruction has its operands ready for consumption are eligible for execution*
  - *Eligible Warps are selected for execution on a prioritized scheduling policy*
  - *All threads in a Warp execute the same instruction when selected*



SM Architecture

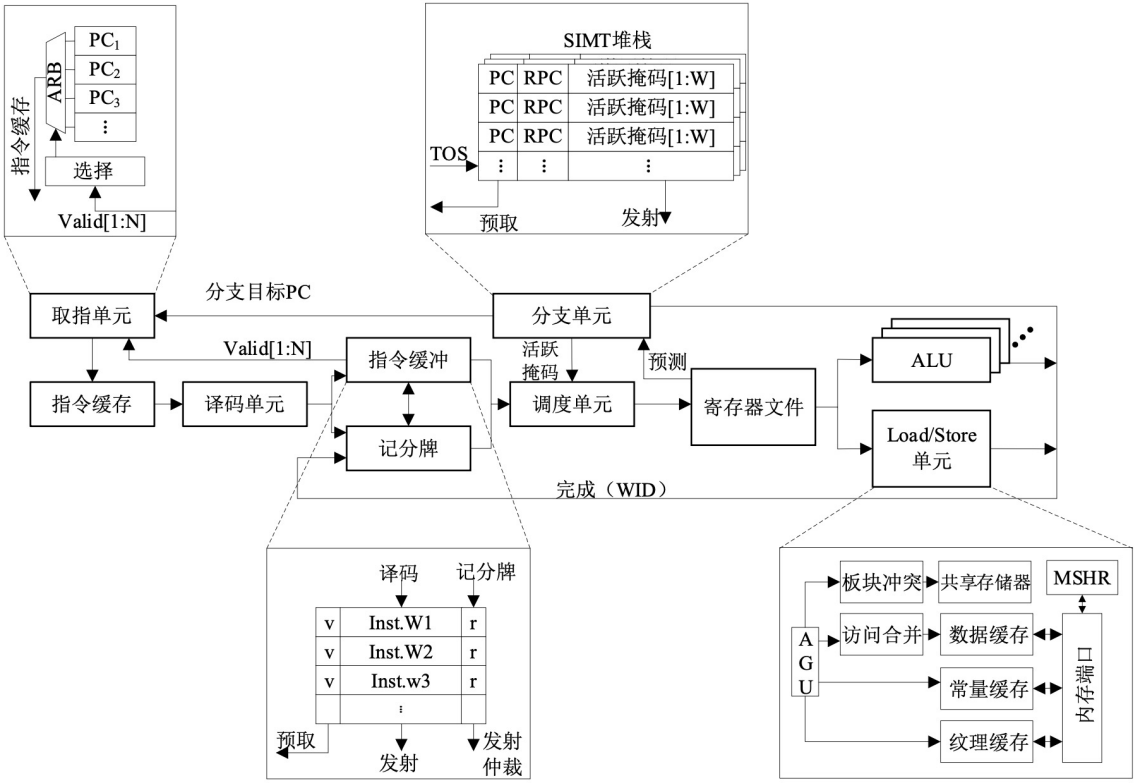


- All register operands of all instructions in the Instruction Buffer are scoreboarded
  - *Instruction becomes ready after the needed values are deposited*
  - *Prevents hazards*
  - *Cleared instructions are eligible for issue*

- Decoupled Memory/Processor pipelines
  - *Any thread can continue to issue instructions until scoreboarding prevents issue*
  - *Allows Memory/Processor ops to proceed in shadow of other waiting Memory/Processor ops*

GPGPU Pipeline

- 流水线技术利用指令级并行，GPGPU以warp为粒度执行
- 调度单元：从指令缓冲中选择一个“ready”warp发射
- 记分牌：检查指令之间的相关性、依赖性
- 分支单元：处理指令分支，指令分支会破坏SIMT的执行方式，对单个线程进行独立控制
- 寄存器文件：获取warp的源操作数



Thread Divergence

- 遇到if else，发生线程分支
- 谓词寄存器，控制每个通道是否开启/关闭

```
1  do {
2      t1 = tid*N;              // A
3      t2 = t1 + I;
4      t3 = data1[t2];
5      t4 = 0;
6      if(t3 != t4){
7          t5 = data2[t2];      // B
8          if(t5 != t4) {
9              x += 1;          // C
10         }else{
11             y += 2;          // D
12         }
13     } else {
14         z += 3;              // F
15     }
16     i++;                     // G
17 } while(i < N);
```

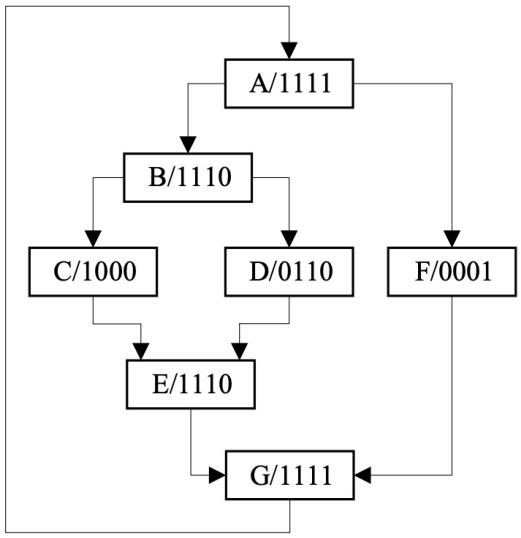
内核函数

```
1   A:  mul.lo.u32      t1, tid, N;
2       add.u32         t2, t1, i;
3       ld.global.u32   t3, [t2];
4       mov.u32         t4, 0;
5       setp.eq.u32     p1, t3, t4;
6       @p1 bra         F;
7   B:  ld.global.u32   t5, [t2];
8       setp.eq.u32     p2, t5, t4;
9       @p2 bra         D;
10  C:  add.u32         x, x, 1;
11      bra             E;
12  D:  add.u32         y, y, 2;
13  E:  bra             G;
14  F:  add.u32         z, z, 3;
15  G:  add.u32         i, i, 1;
16      setp.le.u32     p3, i, N;
17      @p3 bra         A;
```

谓词寄存器

PTX代码

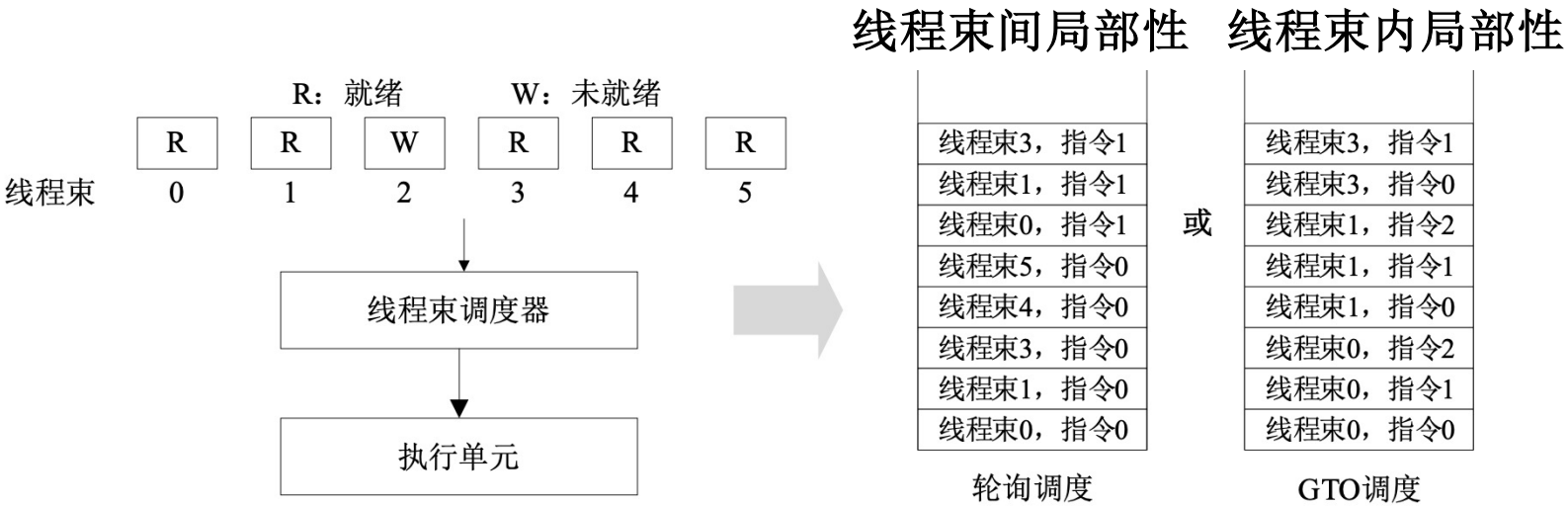
效率低！
尽量避免分支



Warp Scheduler

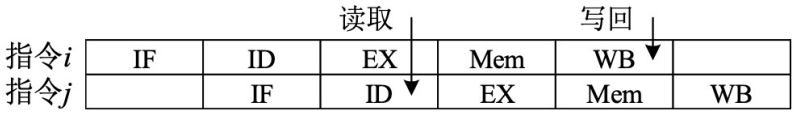
- 从就绪的warp中选择一定量的warp发送给空闲的core
  - 满足条件：下一条指令已经取到、指令的所有相关性都已解决、指令需要的执行单元可用

- 轮询：对所有warp赋予相同的优先级
- GTO（greedy-then-oldest）：允许一个warp执行完
- 改进、优化：优先级、两级调度

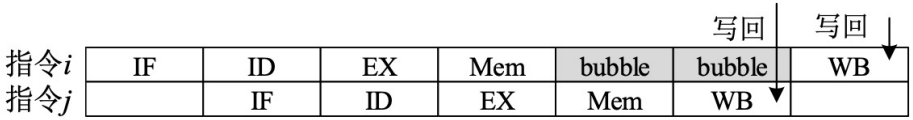


Score Board

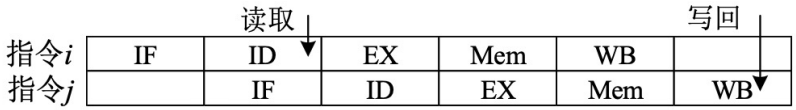
- 由于没有CPU中的乱序执行，GPGPU中不需要太过复杂的记分牌（线程过多、成本过大）
- RAW、WAW、WAR



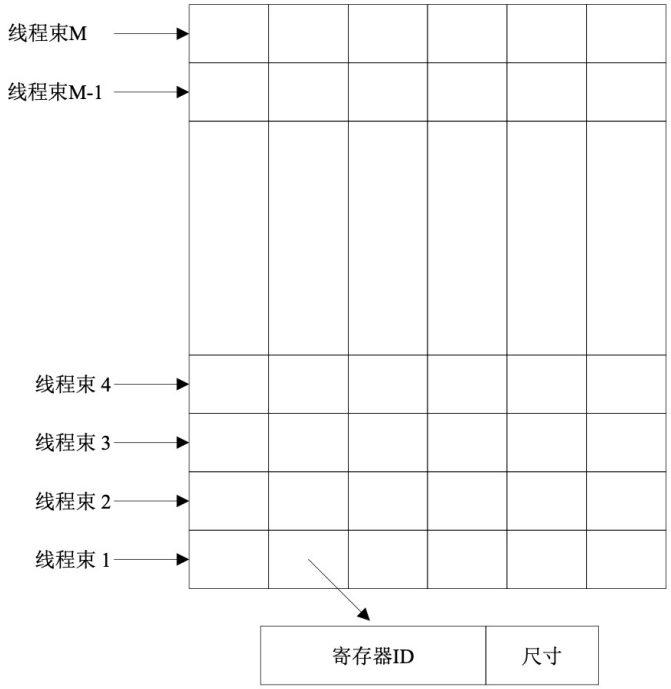
(a) 顺序流水线下的写后读（RAW）冒险（假设指令 i 的目标寄存器和指令 j 的源寄存器为相同寄存器）



(b) 顺序流水线下的写后写（WAW）冒险（假设指令 i 的目标寄存器和指令 j 的目标寄存器相同）



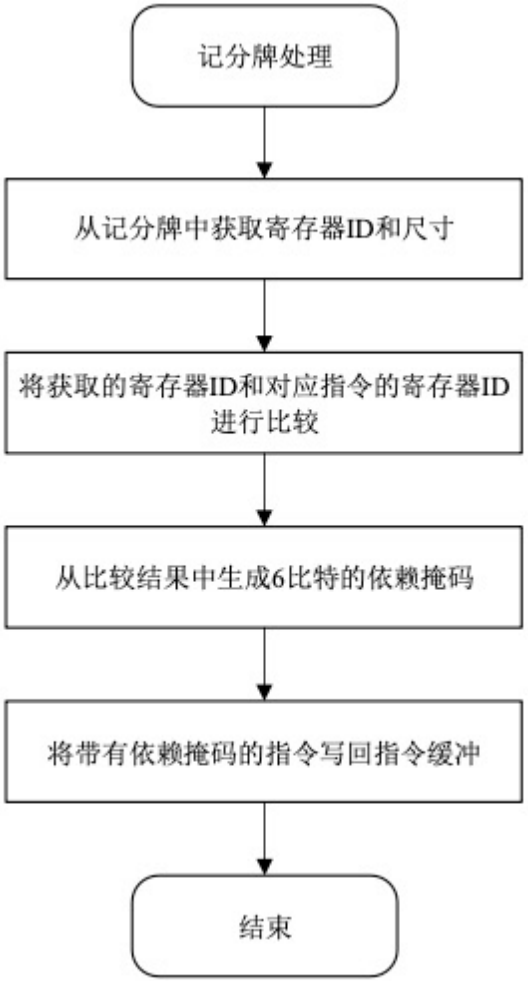
(c) 顺序流水线中不太可能发生读后写（WAR）冒险



寄存器ID：指令要写回的寄存器号
尺寸：连续寄存器号的数量
相较为每个寄存器分配1bit记录是否被写回，该方式更轻量

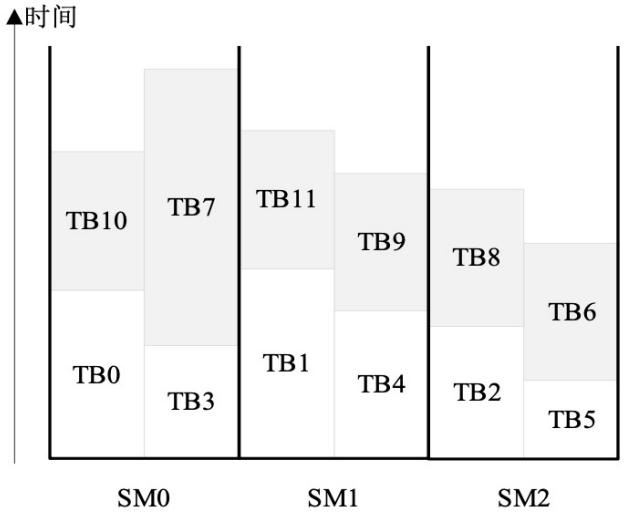
Score Board

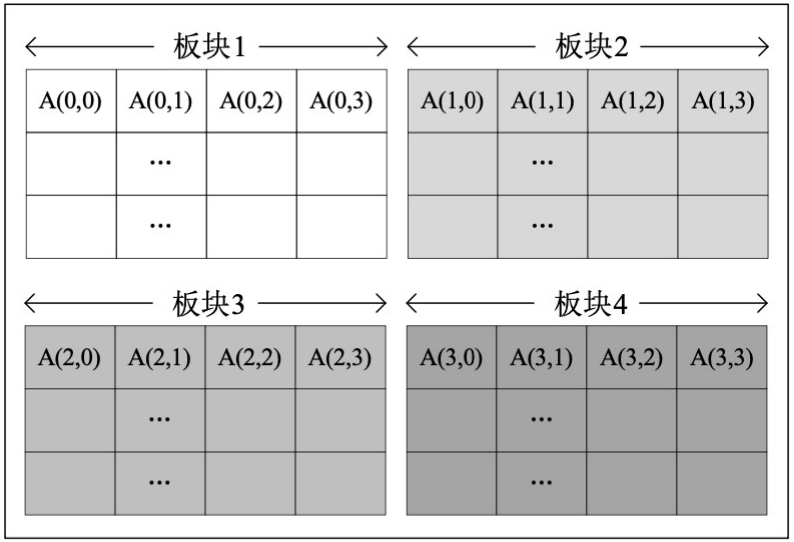
- 由于没有CPU中的乱序执行，GPGPU中不需要太过复杂的记分牌（线程过多、成本过大）
- RAW、WAW、WAR



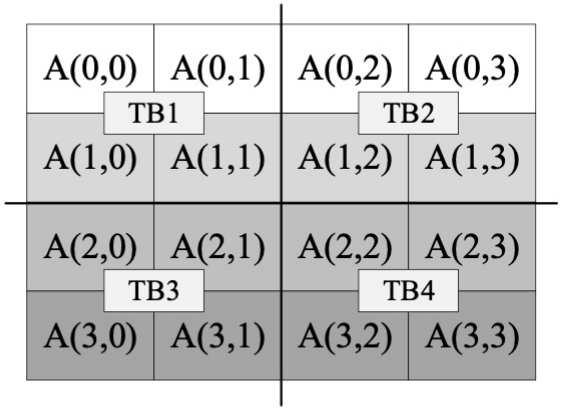
Thread Block Schedule

- 对于GPGPU的编程模型，Thread block（TB）是一个重要的层次，由一个或多个warp组成，是CUDA将任务分配给SM的基本任务单元
- 调度策略影响SM的性能，workload balance
- 轮询
  - 破坏TB之间的空间局部性
  - 增加DRAM板块访问冲突





(a) DRAM数据布局（行优先）

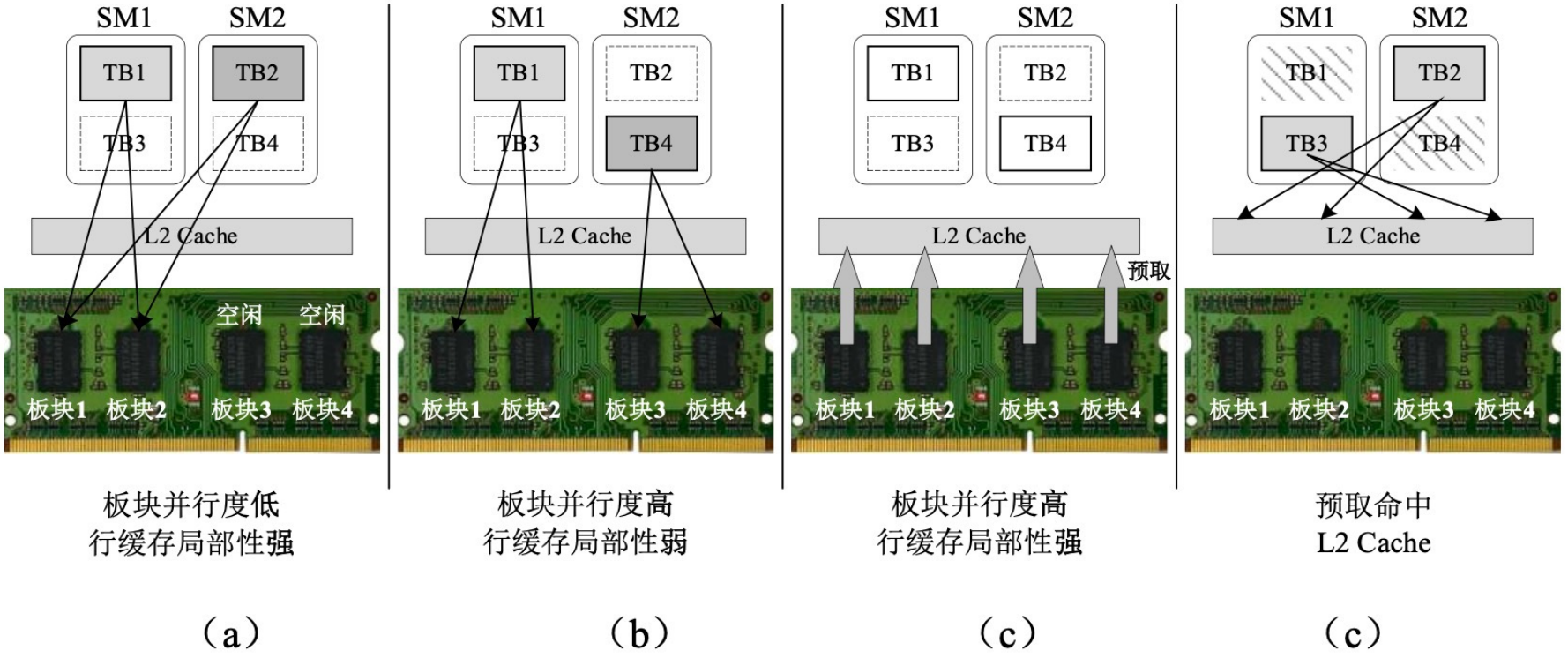


(b) 矩阵与线程块数据布局

Thread Block Schedule

- 考虑Cache、DRAM数据分布特点，结合“prefetch”进行调度

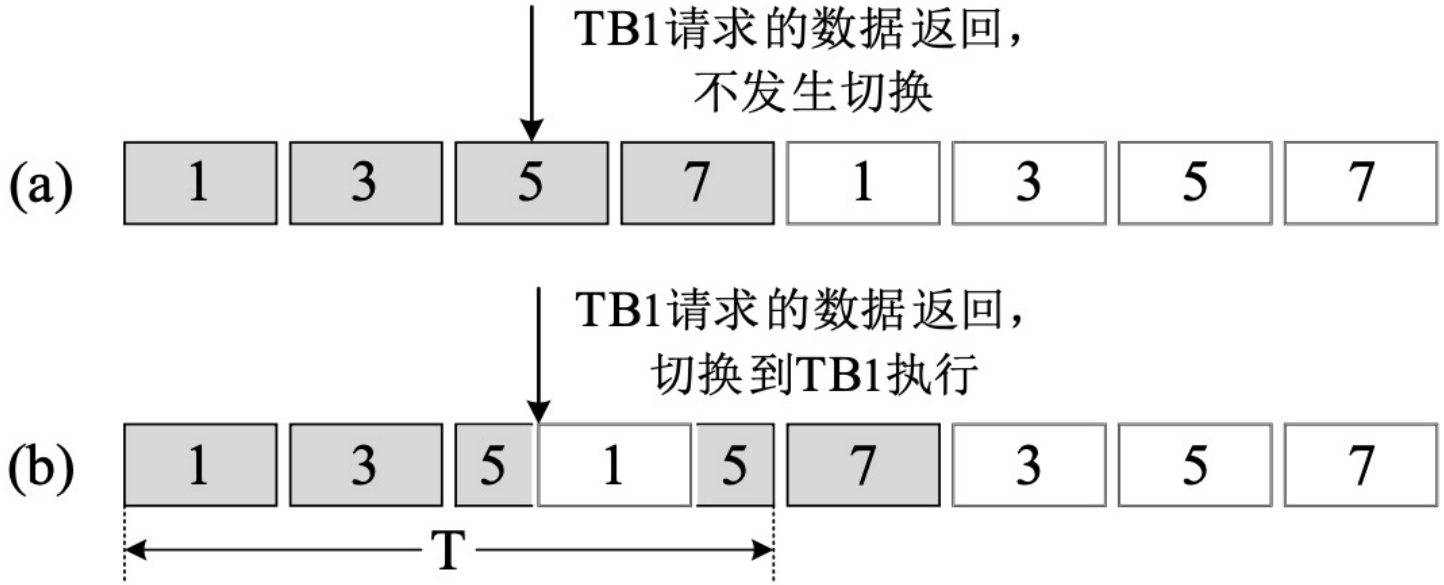
破坏线程块的局部性



Thread Block Schedule

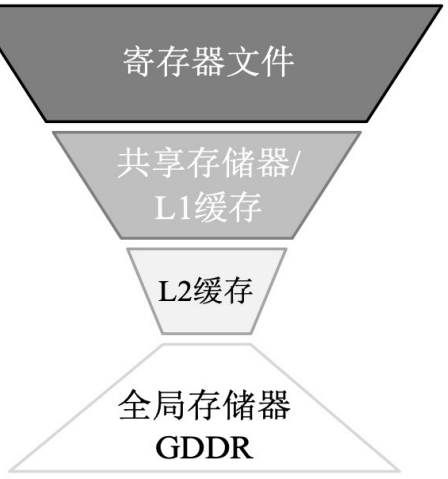
- GPGPU 中 L1 数据缓存的容量往往只有几十或几百 KB，远远无法满足大量线程块并行执行所产生的数据缓存需求，容易导致缓存冲突、抖动和缺失现象
  - **考虑到 L1 数据缓存容量十分有限，TB1 之前加载到缓存的数据很可能被后续执行的线程块替换掉**

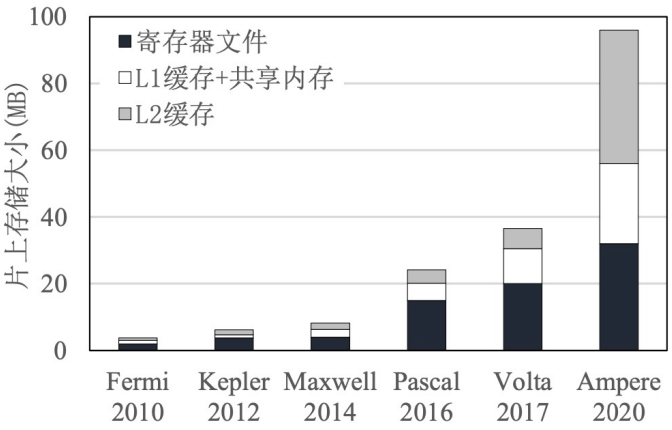
- 感知时间局部性的抢占式调度：允许再次进入就绪态的TB抢占



GPGPU Memory Hierarchy

- CPU：金字塔
- GPU：倒金字塔
  - 寄存器文件、L1 cache/共享存储器，位于SM内部
    - A100中L1 cache容量高达10MB
  - L2 cache是SM间共享



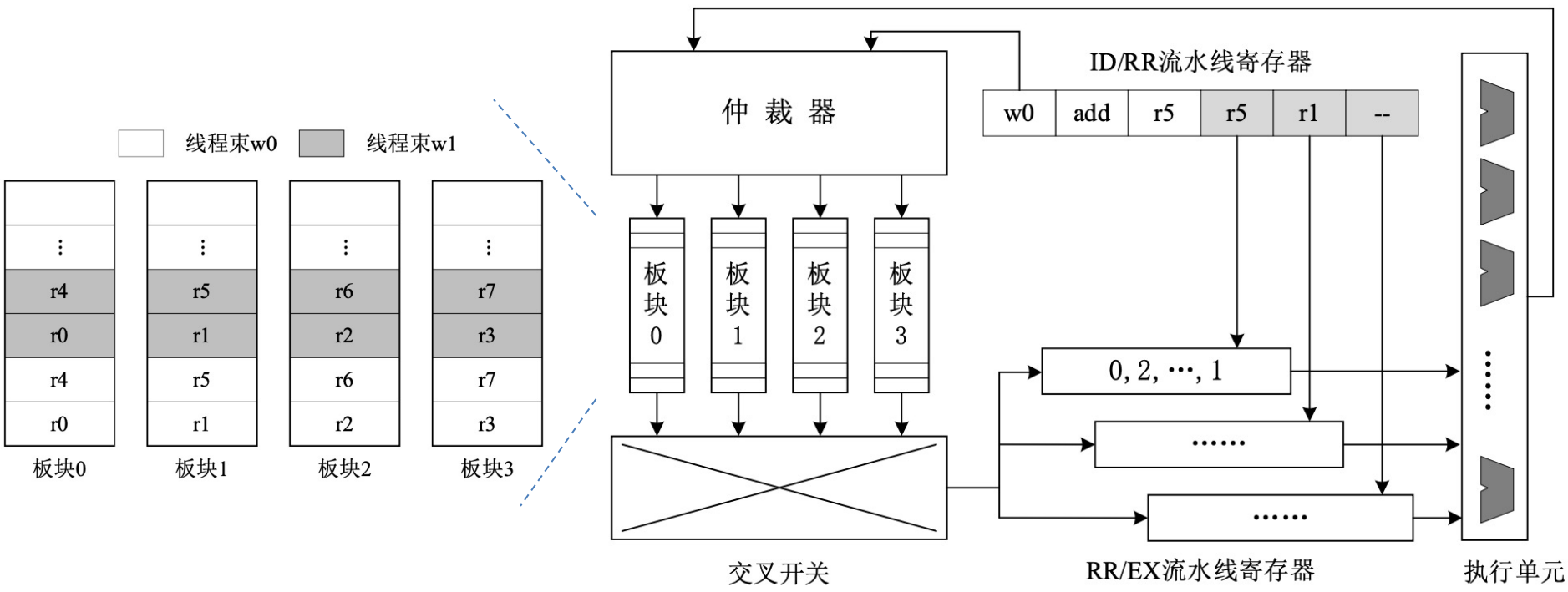


CPU存储结构

GPGPU存储结构

Register File

- Register file寄存器文件：SRAM，考虑到GPU的主频较低（1.2GHz）
  - 节约面积
- 多板块单端口SRAM设计



Bank Conflict

- 由于多板块单端口的RF设计，可能存在bank conflict
- _n表示所属bank

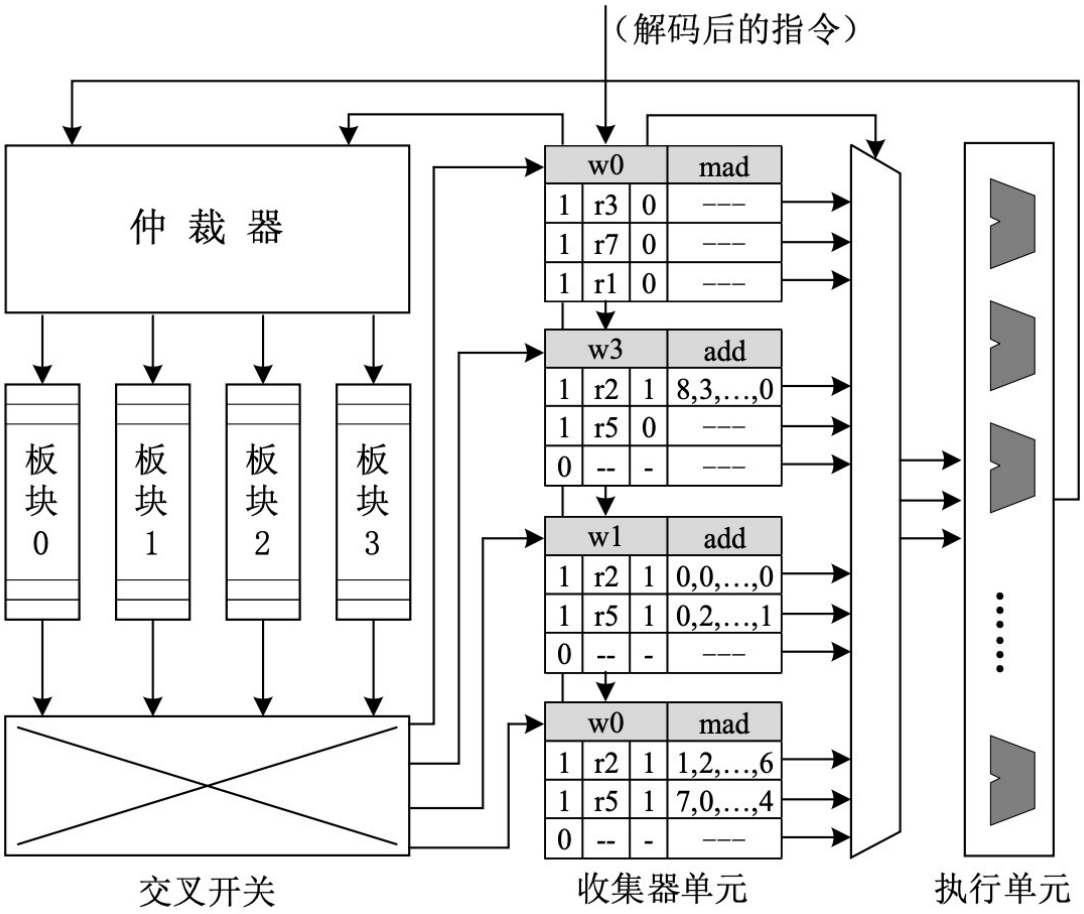
```
i1:  mad   r2_2, r5_1, r4_0, r6_2
i2:  add   r5_1, r5_1, r1_1
```

周期	线程束	指令
0	w0	i1: mad r2_2, r5_1, r4_0, r6_2
1	w1	i2: add r5_1, r5_1, r1_1
3	w2	i2: add r5_1, r5_1, r1_1
6	w3	i2: add r5_1, r5_1, r1_1

周期		1	2	3	4	5	6	7	8	9	10	11
板块	0	w0 i1:r4										
	1	w0 i1:r5	w1 i2:r1	w1 i2:r5	w2 i2:r1	w1 i2:r5	w2 i2:r5	w3 i2:r1	w2 i2:r5	w3 i2:r5		w3 i2:r5
	2	w0 i1:r6		w0 i1:r2								
	3											
操作		w0读操作数	w0执行 w1读操作数	w0写回 w1读操作数	w1执行 w2读操作数	w1写回	w2读操作数	w2执行 w3读操作数	w2写回	w3读操作数	w3执行	w3写回

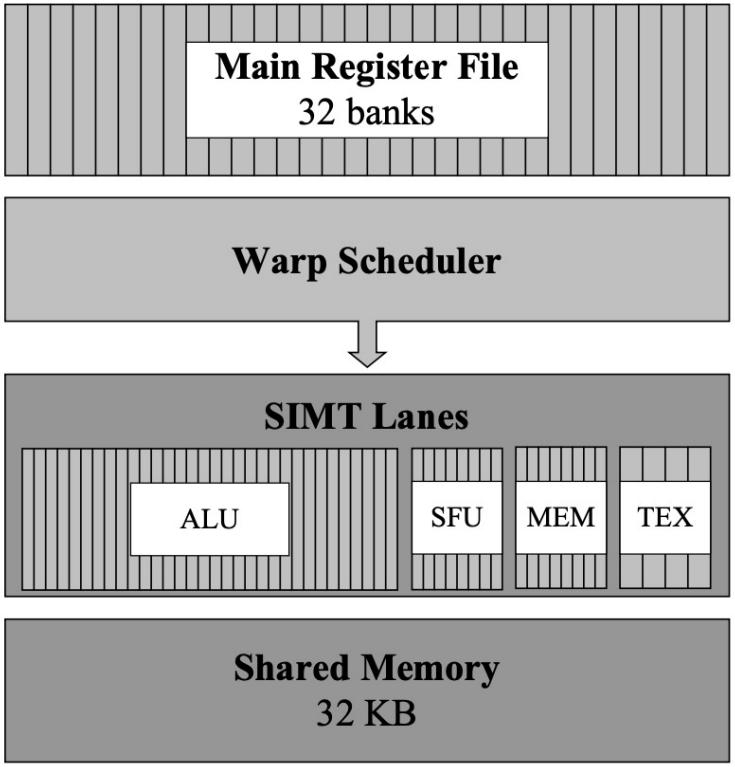
Operand Collector

- 通过允许尽可能多的指令同时访问操作数，利用多板块并行性提高来访问效率
  - 防止大量warp被bank conflict阻塞
- 为每个进入寄存器读取阶段的warp分配一个收集器单元

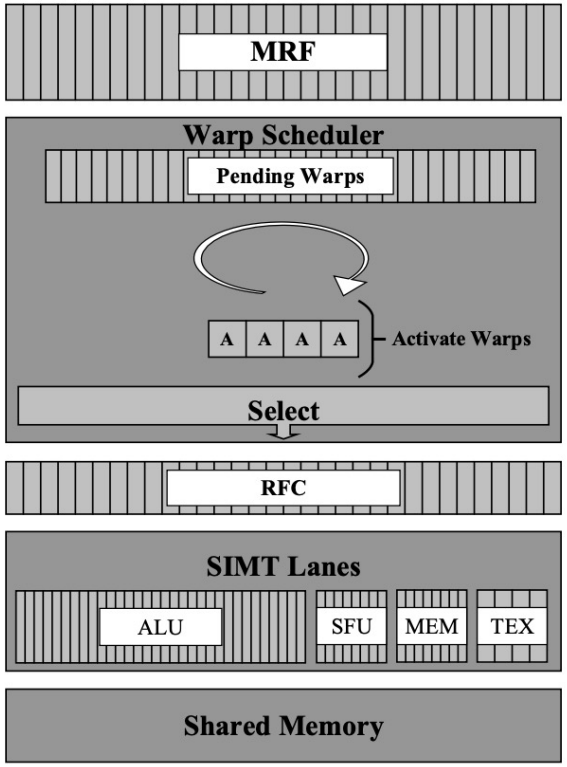


Optimization for RF

- 增加前置寄存器文件缓存设计
  - 部分数据生命期短、复用率低，写入大容量RF开销大
  - Register file cache



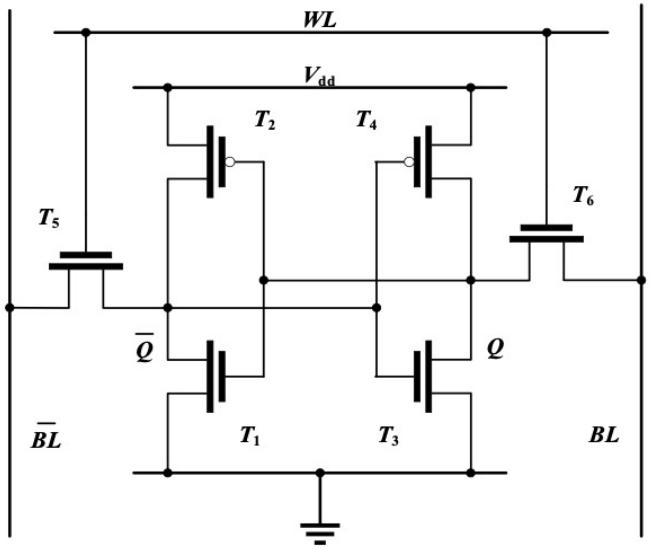
（a）原有可编程多处理器架构

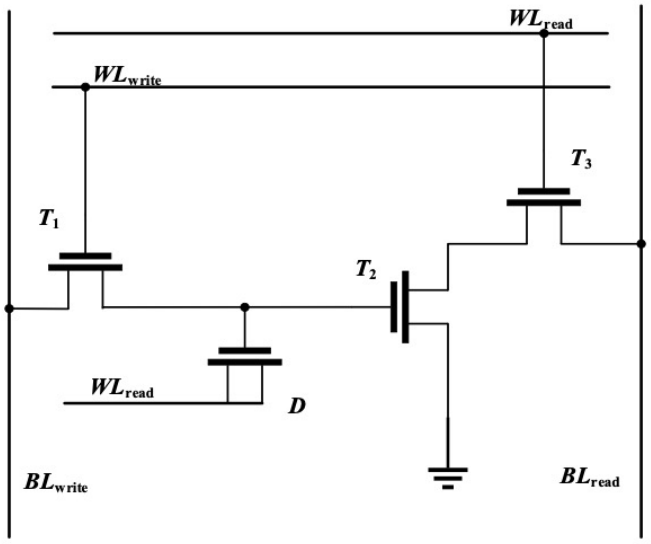


（b）修改后的架构

Optimization for RF

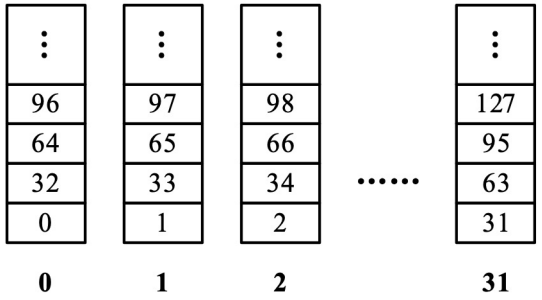
- 基于嵌入式动态存储器的RF设计
  - 嵌入式动态存储器eDRAM（栅级电容）相较SRAM（晶体管）功耗更低，轻量
  - 但数据保留时间短，引入“刷新”操作
  - 当发生bank conflict时，其他bank可能是空闲的，对其他bank进行“刷新”，隐藏刷新的延迟



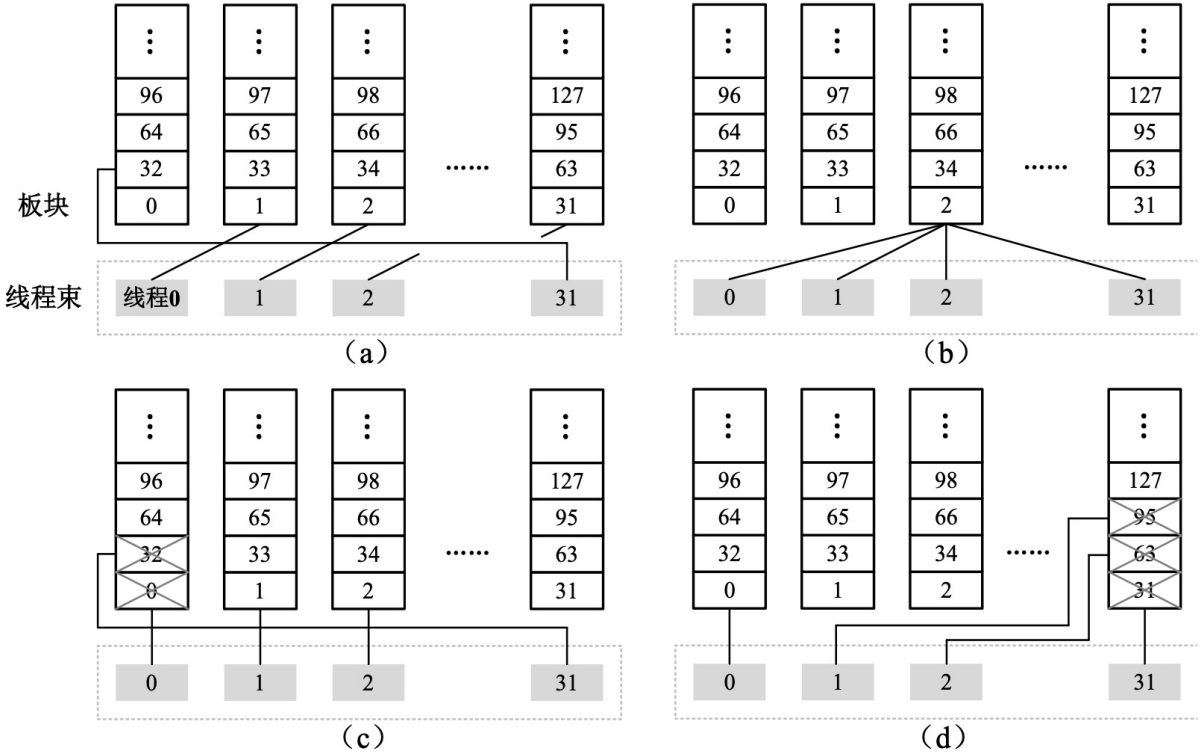


Shared Memory

- 分bank，与L1 cache同级
  - 一个warp访问32个bank
  - 一个warp访问同一bank的同一行
  - 一个warp访问同一bank的不同行，bank conflict！
- Shared memory绕过tag检查，其地址由编程人员指定

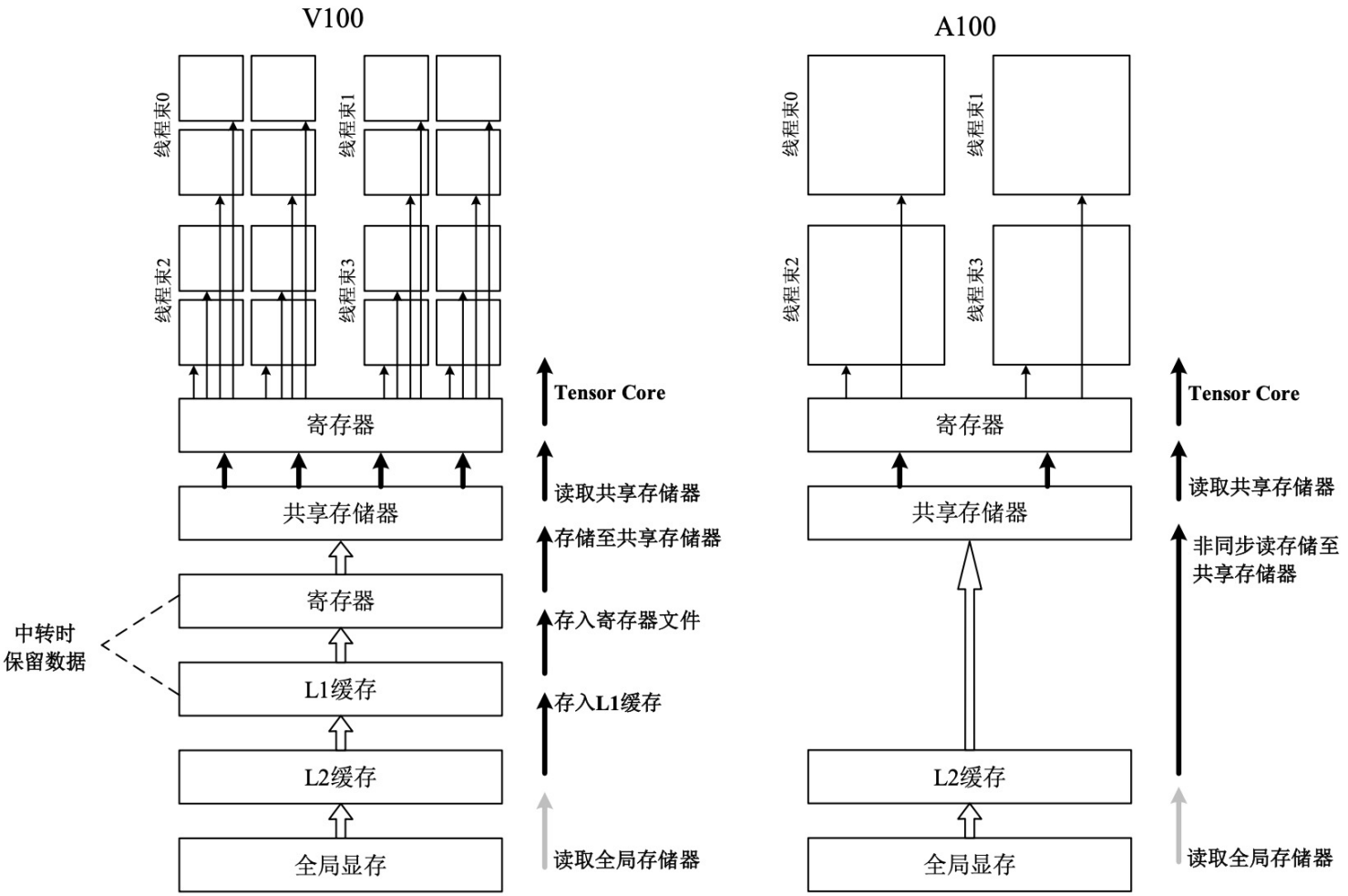


共享存储器板块

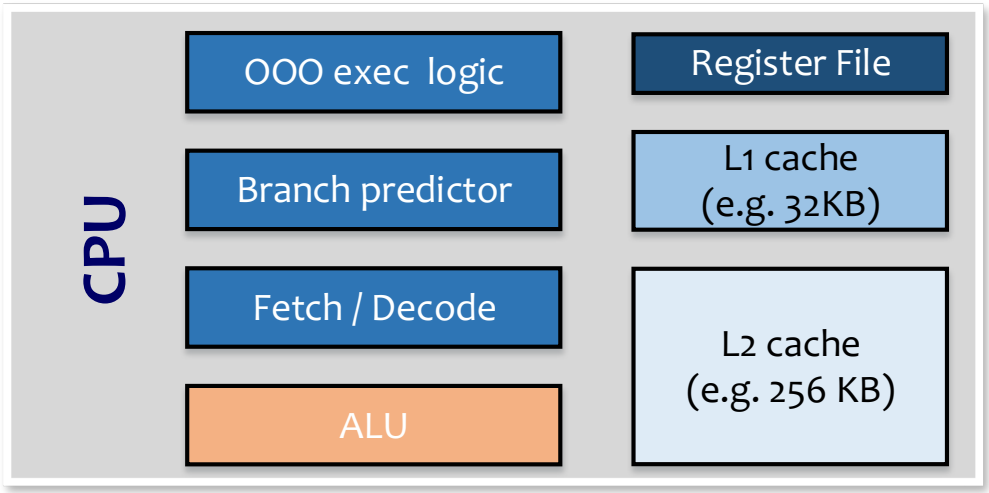


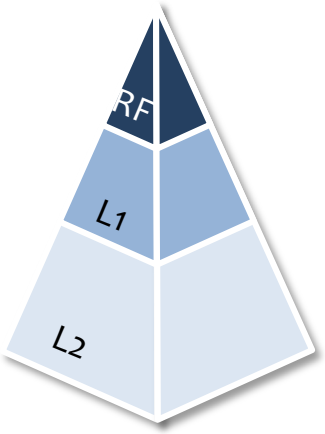
Dataflow

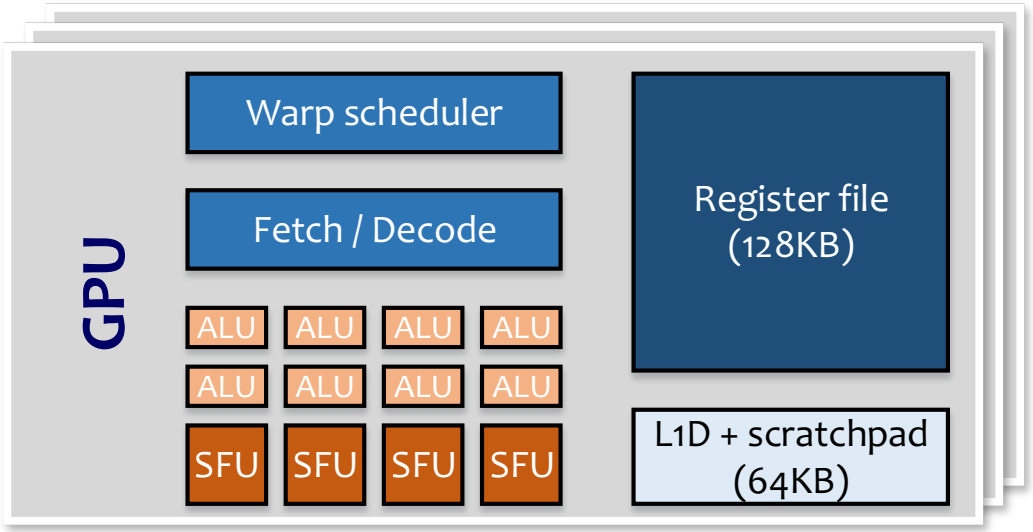
- 前几代GPU访问shared memory需要跨越多个层级
- Ampere架构重新设计了全局存储器到shared memory的数据通路

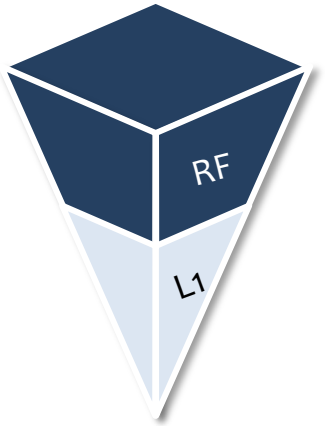


CPU & GPU Architecture Outline









NVIDIA Volta