ReArch Group Meeting 2023.09.21

GPU Warp Scheduling and Control Code

胡洧铭

答辩提纲

一 背景

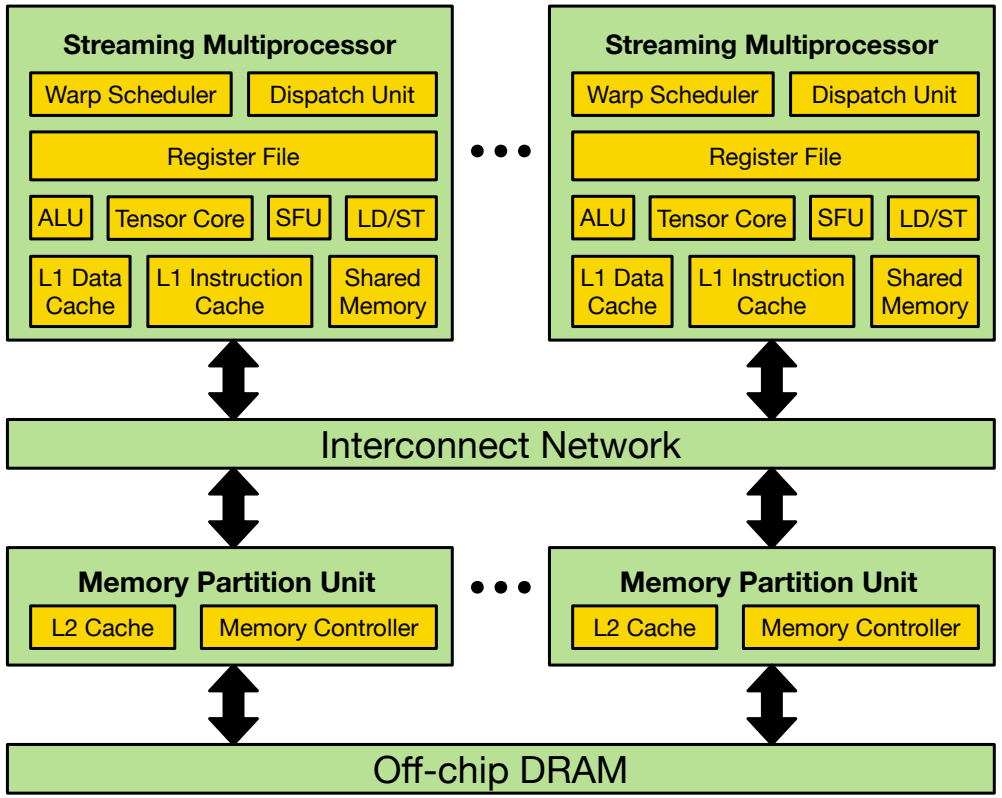
二 自适应线程束调度算法

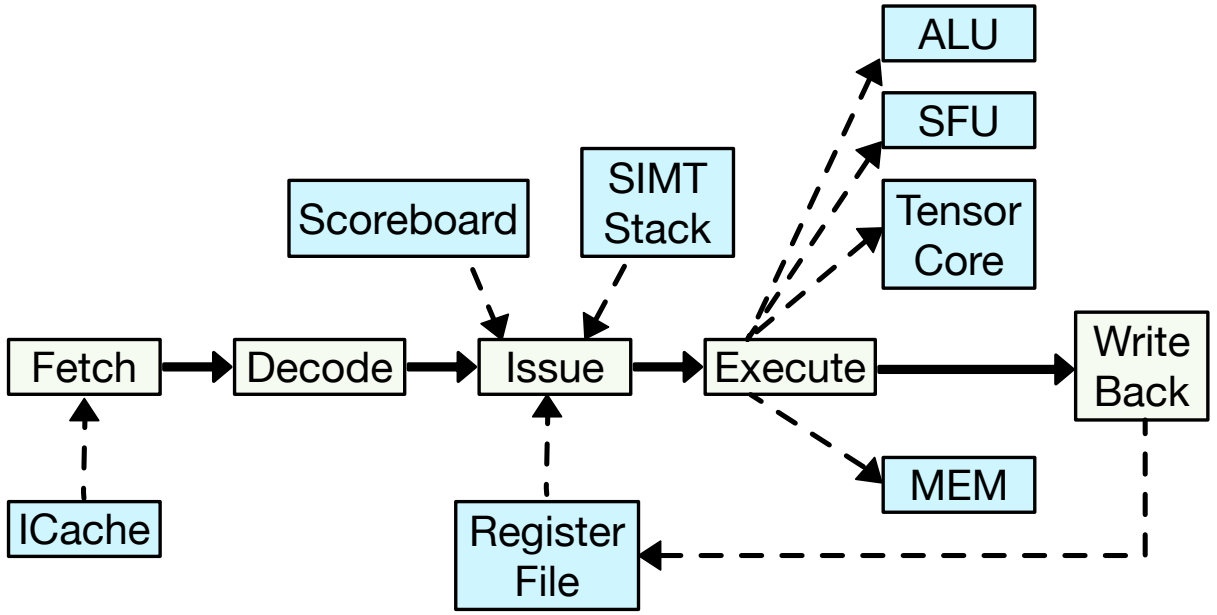
三 基于指令控制码的调度

四 论文总结

1.1 背景

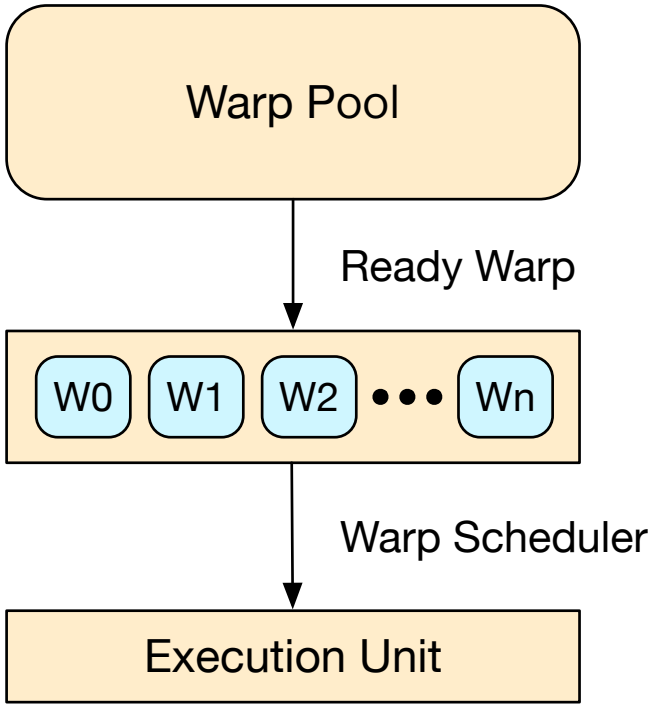
- 基本的 GPU 微架构和流水线
  - 流多处理器：计算单元、寄存器堆、线程束调度器、L1 Cache
  - 片上互联网络、L2 Cache、内存控制器、DRAM

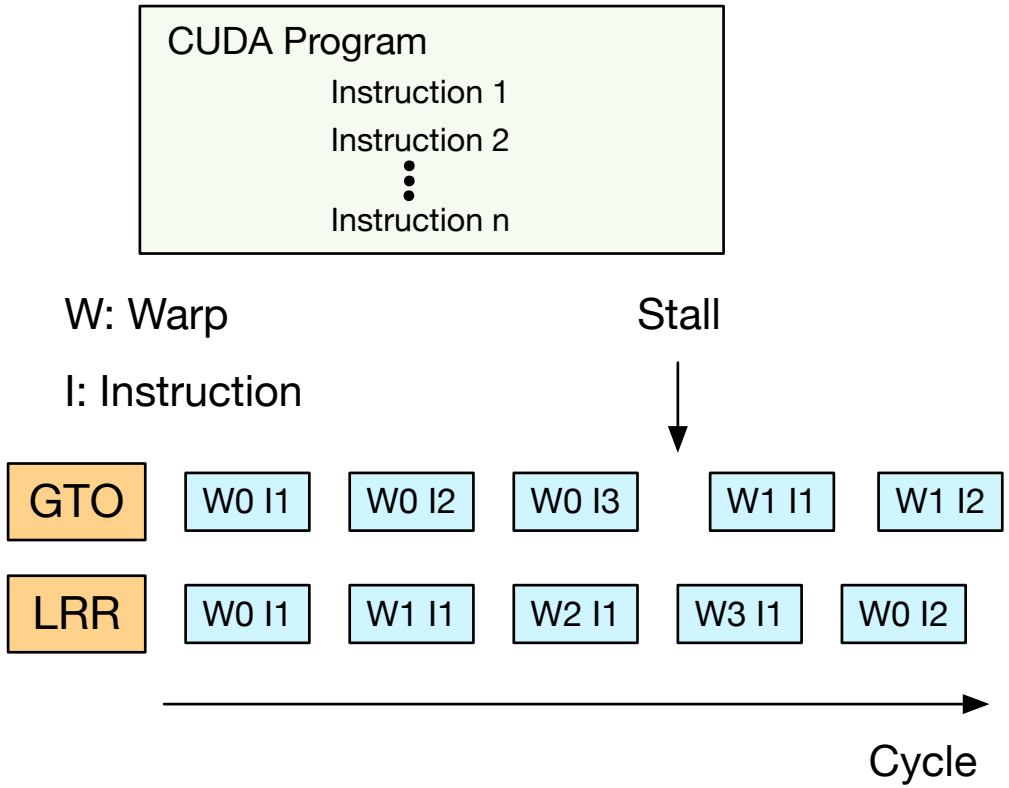




1.1 背景

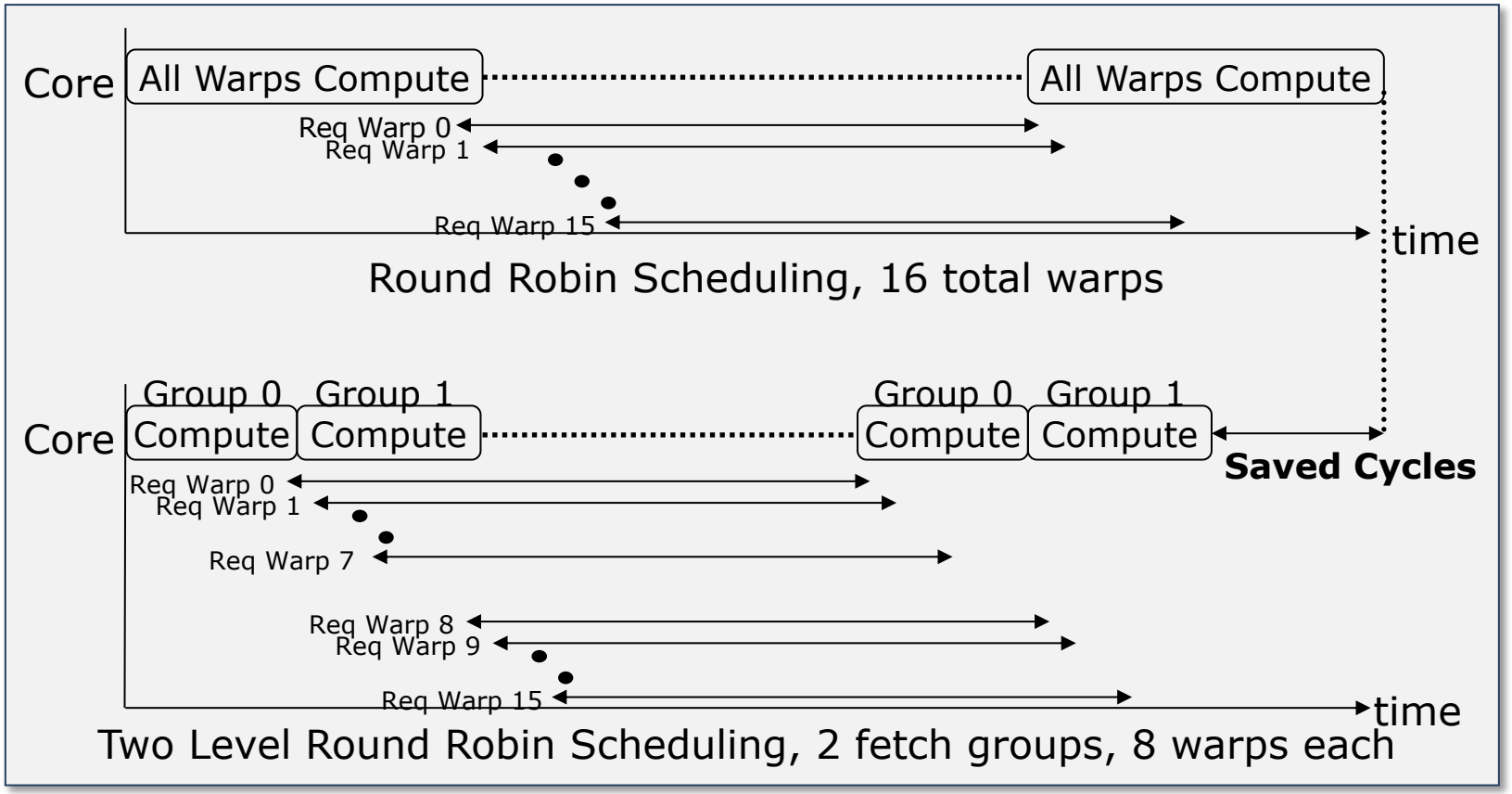
- 两种硬件设计上简单的调度算法
  - 松散-轮询调度算法，Loosely Round Robin (LRR)
  - 贪心-最老线程束调度算法，Greedy-Then-Oldest (GTO)





1.2 一些相关的研究

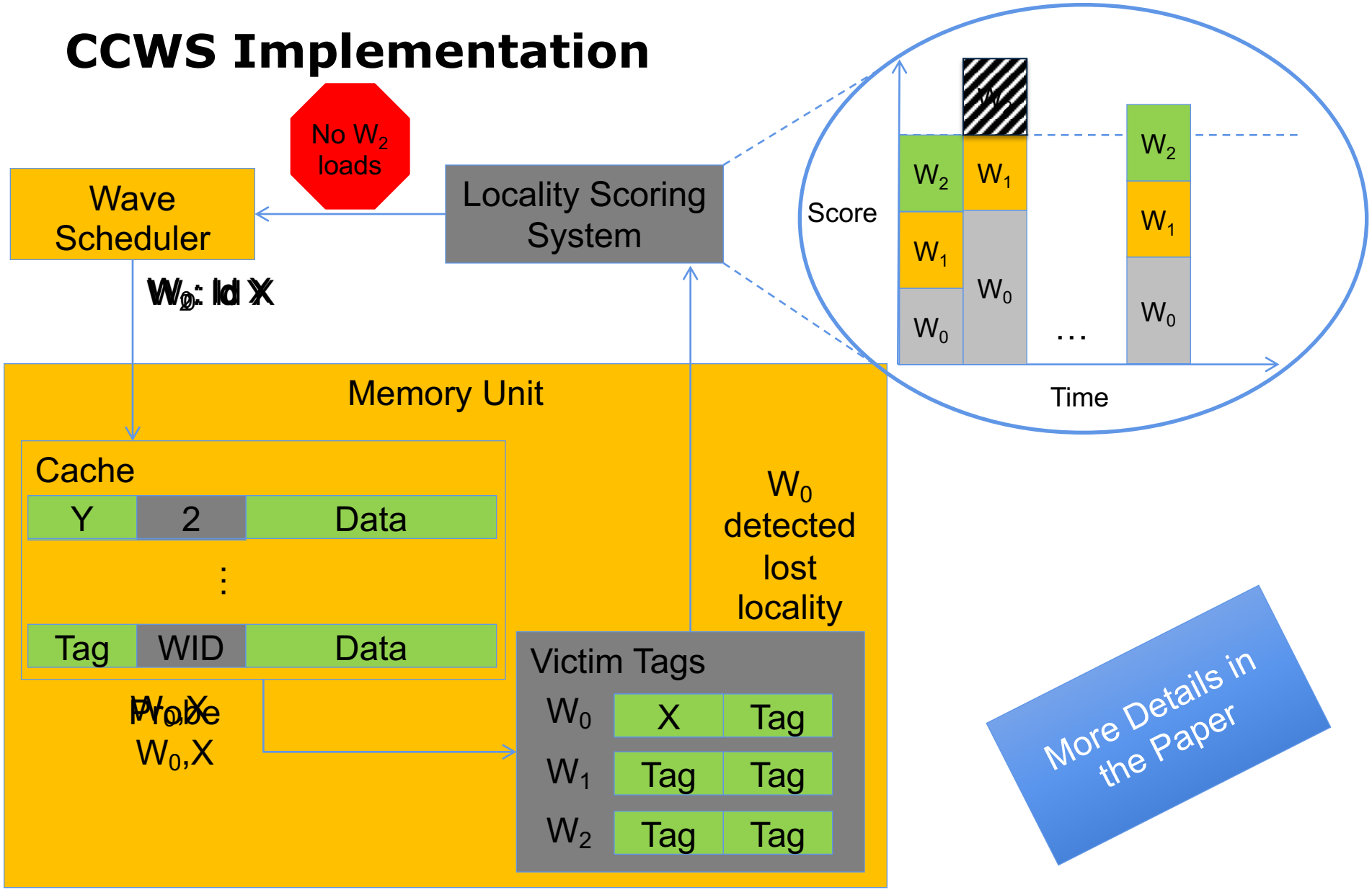
- Two-level, Improving GPU performance via large warps and two-level warp scheduling, MICRO, 2011
  - Motivation: 现有的 round robin 遇到 long latency 操作可能同时 stall
  - Key idea: 多个 warp 组成 fetch group，fetch group stall 时切换



1.2 一些相关的研究

- CCWS, MICRO, 2012
  - Motivation: 现有的 RR 调度算法无法利用 intra-warp cache locality
  - Key idea: 利用 victim tag，检测 warp locality 是否损失，来决定 warp scheduling 的优先级

CCWS Implementation



1.2 一些相关的研究

- iPAWS, HPCA, 2016
  - Motivation: 一种调度算法无法满足多个应用的 workload
  - Key idea: 动态地选择 warp scheduling policy

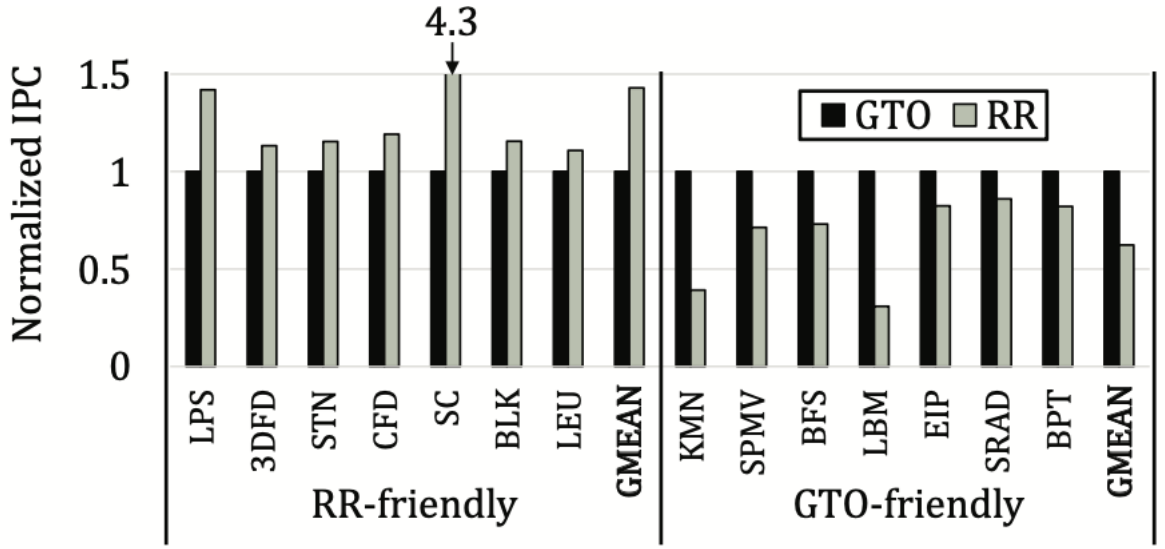


Figure 1: Performance comparison of a greedy scheduler (GTO) and a fair, round-robin (RR) scheduler across different GPGPU workloads.

1.3 两个方面

- 微架构：基于缓存局部性的自适应线程束调度策略
  - 缓存局部性的分类及量化方法
  - CUDA kernel 粒度的性能分析
  - 自适应线程束调度策略
- 汇编层面：能否实际影响到 GPU 的线程束调度
  - 指令控制码对线程束调度的影响
  - 指令控制码重排以优化矩阵乘法性能

答辩提纲

一 研究背景及意义

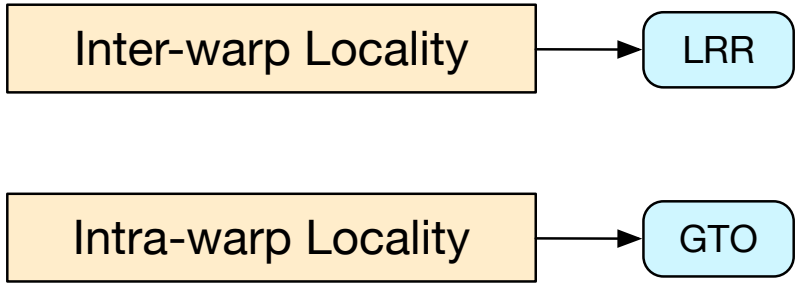
二 **自适应线程束调度算法**

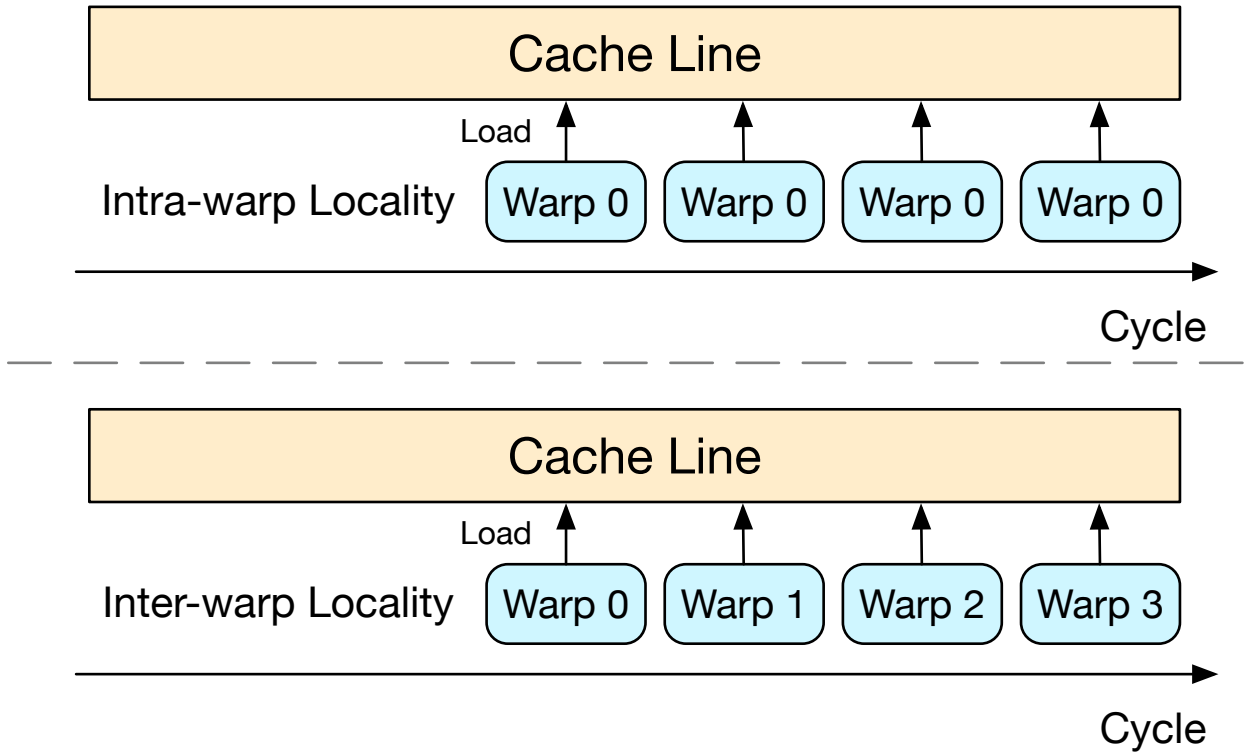
三 基于指令控制码的调度

四 论文总结

2.1 缓存局部性

- 线程束间局部性，Inter-warp locality
- 线程束内局部性，Intra-warp locality

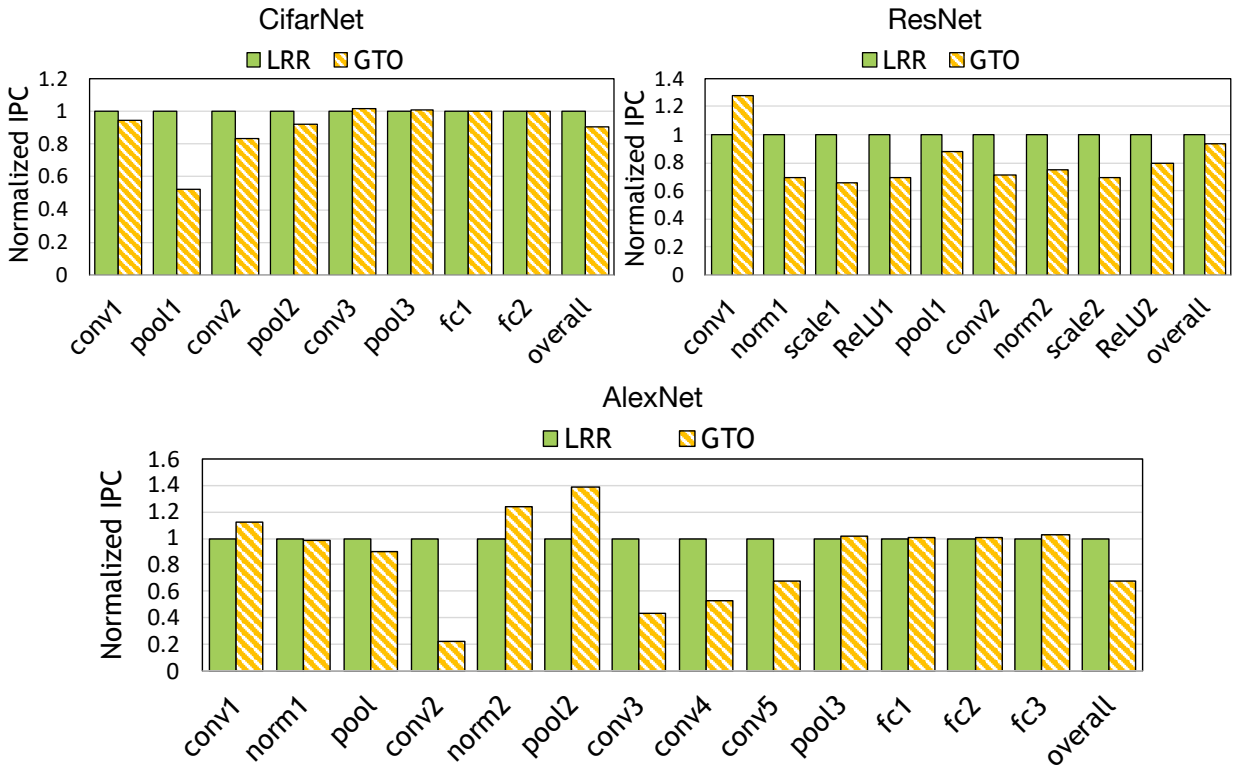




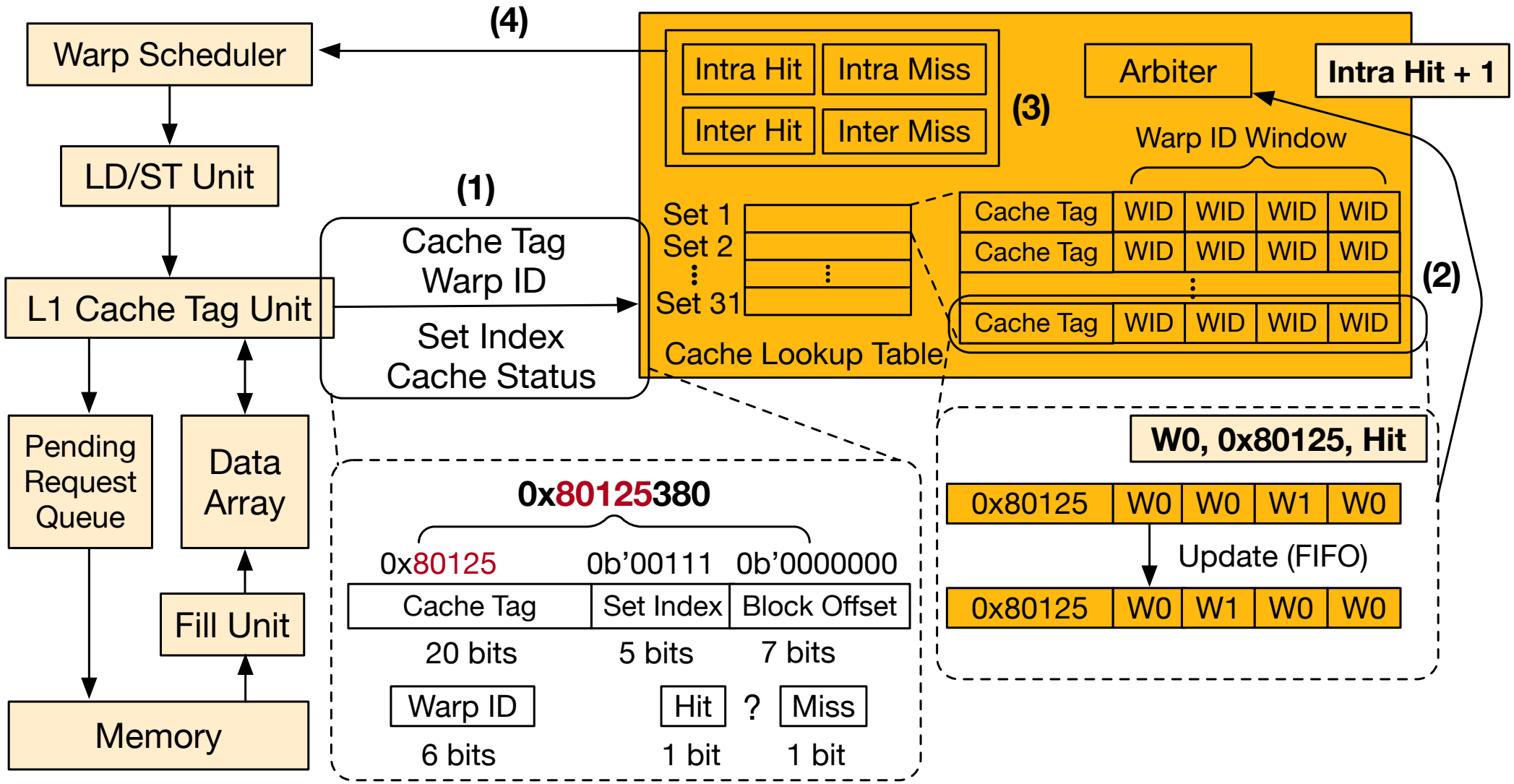
2.2 研究动机

- 以 CUDA kernel 粒度进行分析，LRR 和 GTO 存在的性能差异
  - Benchmark: CifarNet, ResNet, AlexNet

- LRR 和 GTO 策略在硬件实现上相对简单，并且能够匹配两种不同类型的缓存局部性

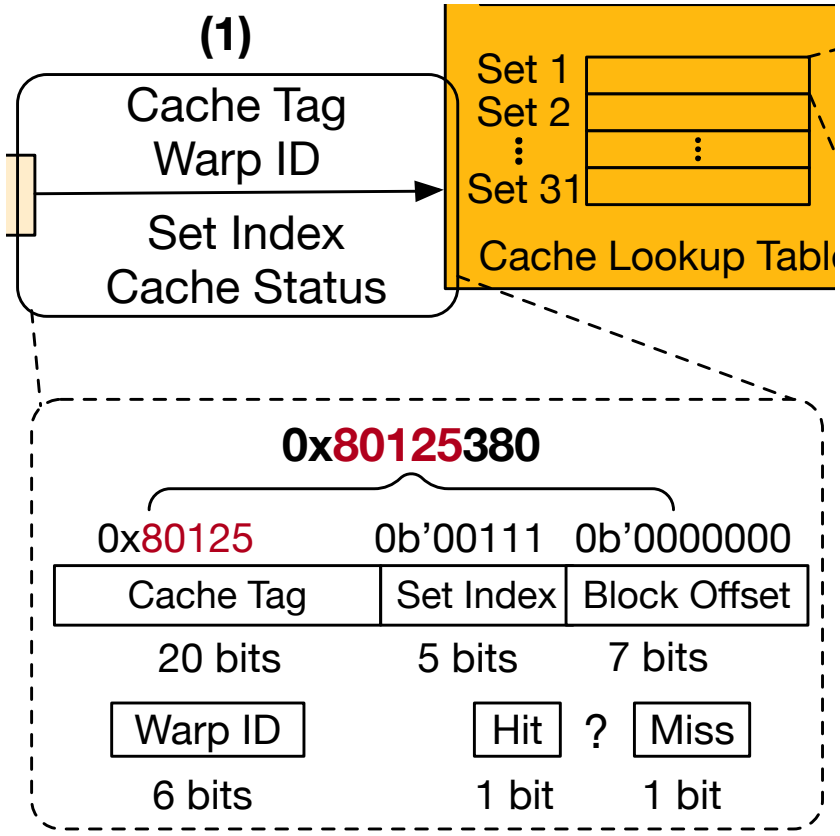


2.2 缓存局部性查找表


Warp Scheduler
LD/ST Unit
L1 Cache Tag Unit
Pending Request Queue
Data Array
Fill Unit
Memory
(4)
Intra Hit
Intra Miss
Inter Hit
Inter Miss
(3)
Arbiter
Intra Hit + 1
Warp ID Window
(1)
Cache Tag
Warp ID
Set Index
Cache Status
Set 1
Set 2
Set 31
Cache Lookup Table
Cache Tag
WID
(2)
W0, 0x80125, Hit
0x80125
W0
W1
Update (FIFO)
0x80125380
0x80125
0b'00111
0b'0000000
Cache Tag
Set Index
Block Offset
20 bits
5 bits
7 bits
Warp ID
Hit
?
Miss
6 bits
1 bit
1 bit

2.2 缓存局部性查找表

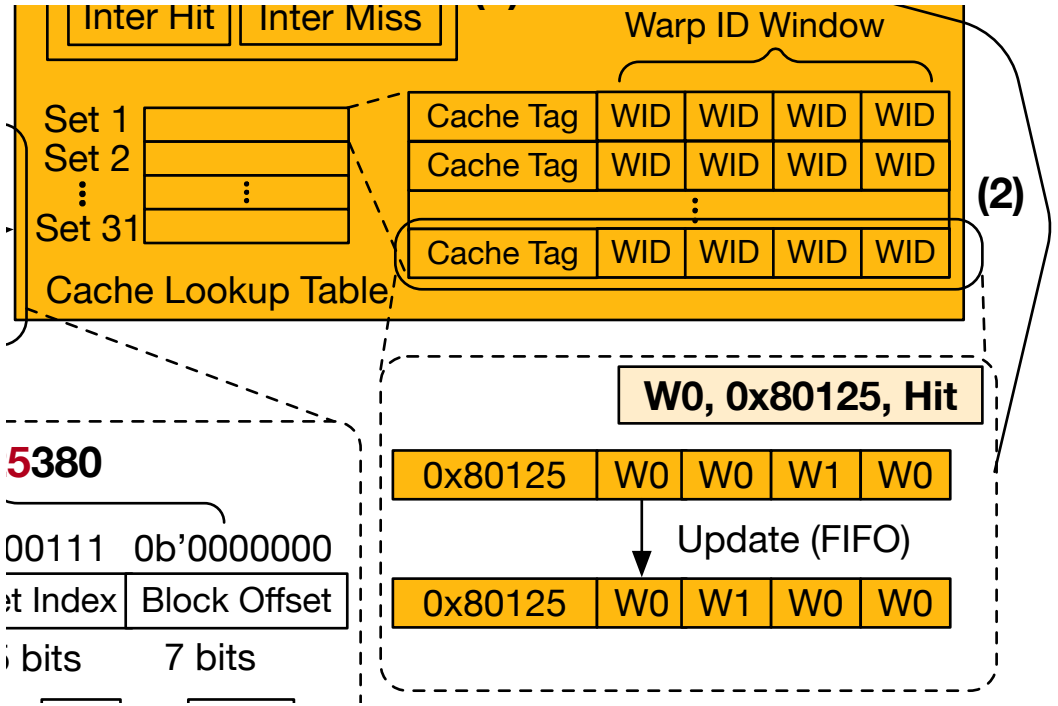
（1）将访存指令的信息送到 Cache 查找表，用于评估缓存局部性类型



- Cache 配置
  - NVIDIA Fermi, 16KB L1 Data Cache
  - 32 sets, 4-way, 128B per cache line

2.2 缓存局部性查找表

（2）确认查找表中是否有匹配的缓存标签，更新滑动窗口的信息



- 滑动窗口中记录过去 4 个访问此 cache line 的线程束 ID
- 根据滑动窗口内的信息，为缓存局部性类型投票

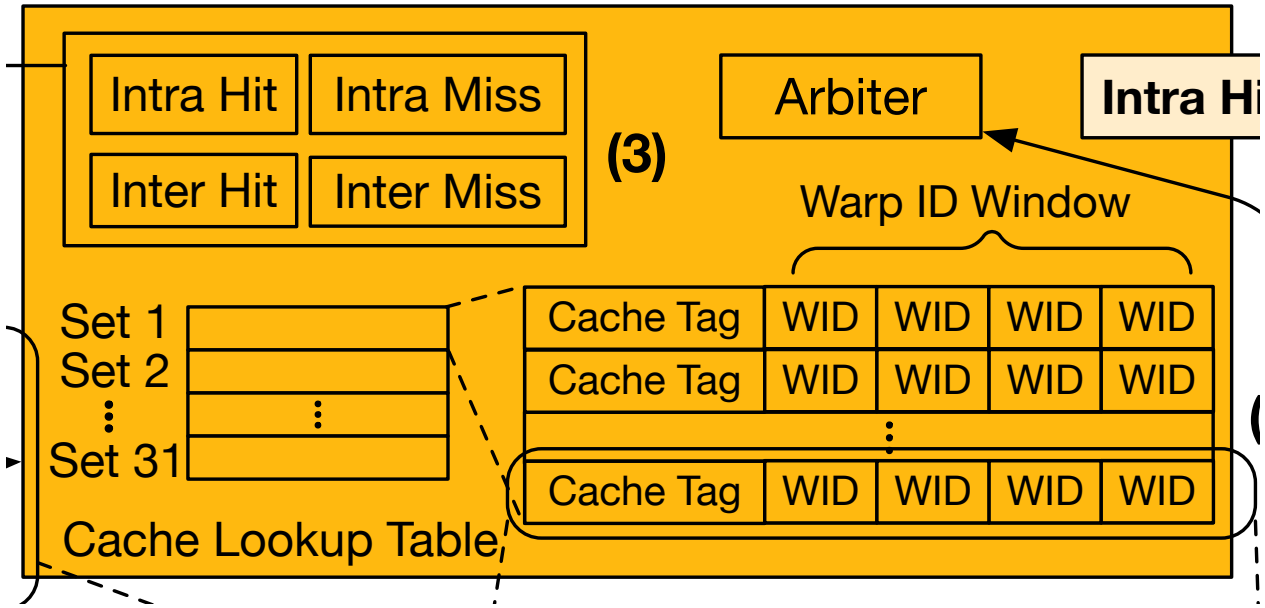
Algorithm 1 统计缓存局部性的计数器

```
temp_intra ← 0
temp_inter ← 0
/* 统计缓存局部性的计数器 */
for i = 1 to 4 do
    if warp_id = Table[set][tag][i] then
        temp_intra ← temp_intra + 1
    else if abs(warp_id - Table[set][tag][i]) ⩽ 2 then
        temp_inter ← temp_inter + 1
    end if
```

2.2 缓存局部性查找表

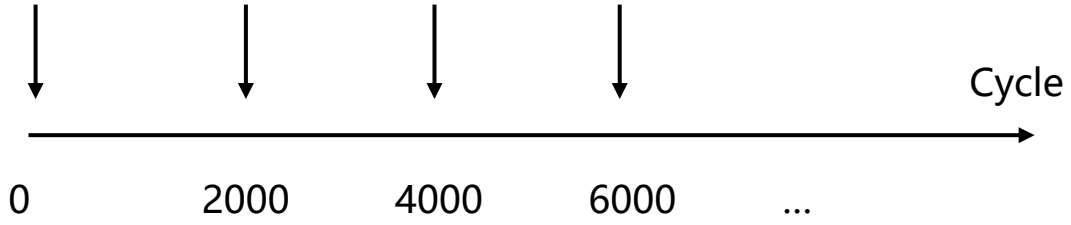
（3）根据缓存局部性类型和是否 Cache 命中，更新计数器(Counter)

- Cache 配置
  - NVIDIA Fermi, 16KB L1 Data Cache
  - 32 sets, 4-way, 128B per cache line

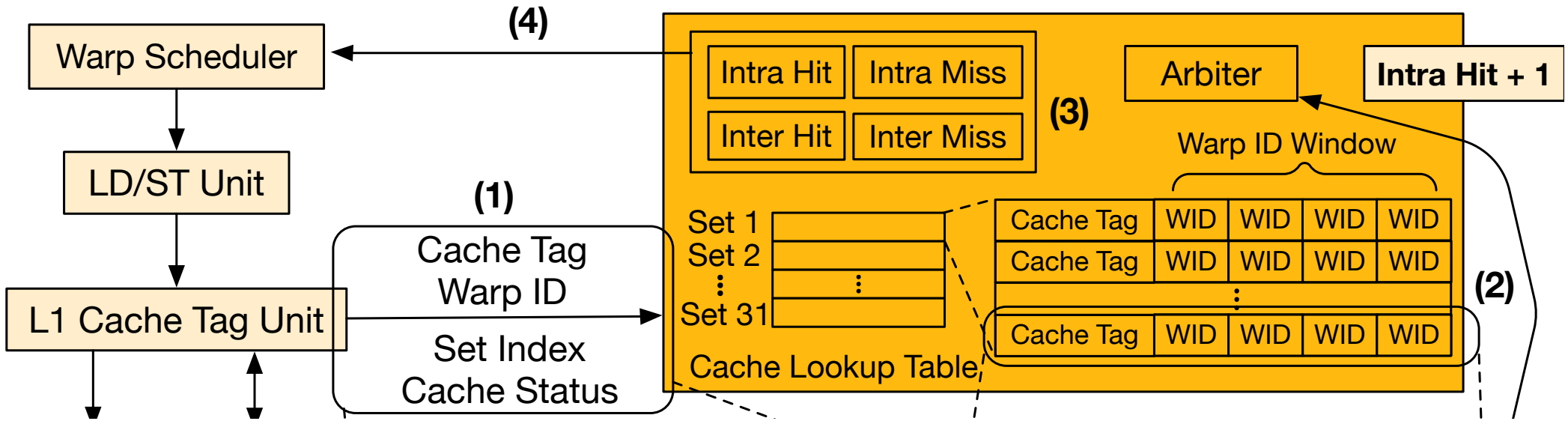


2.2 缓存局部性查找表

（4）每过一定的周期，根据计数器的信息来调整线程束调度策略



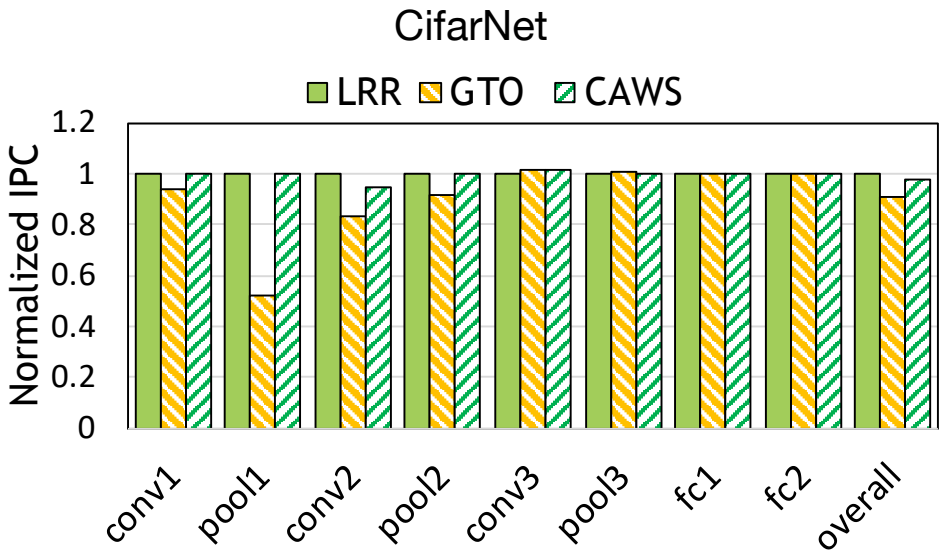
- Cache 配置
  - NVIDIA Fermi, 16KB L1 Data Cache
  - 32 sets, 4-way, 128B per cache line

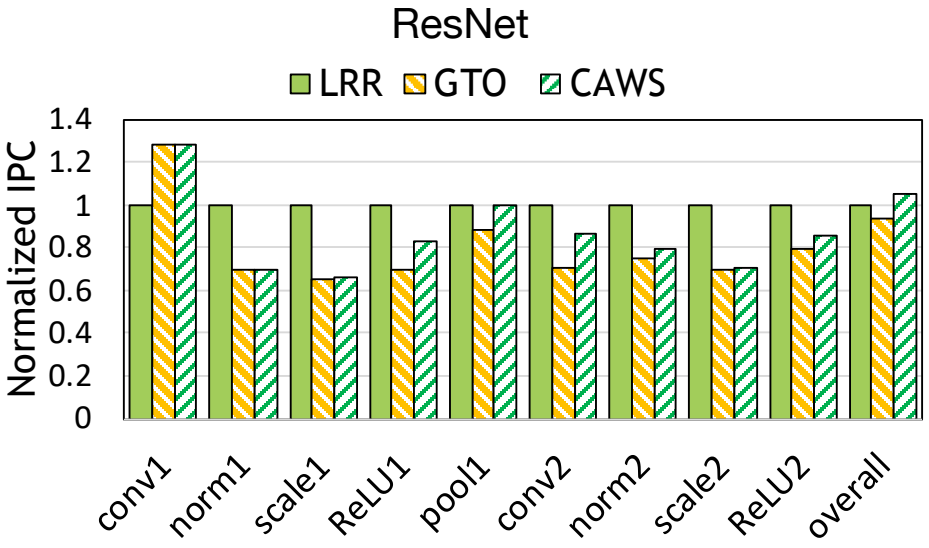


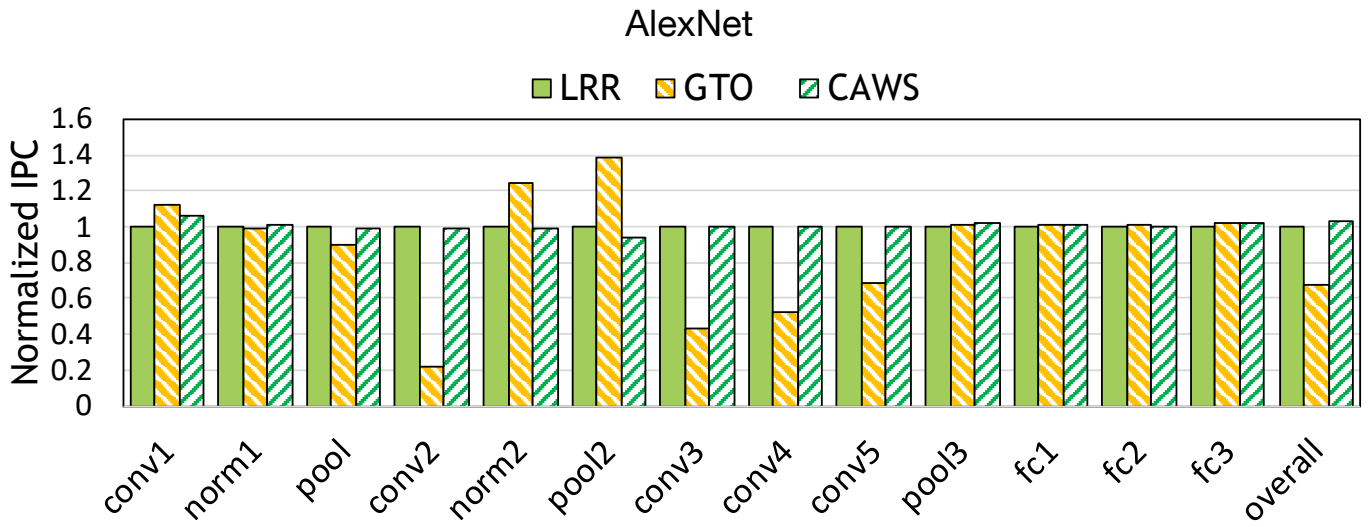
2.3 实验结果

- 实验环境
  - GPGPU-Sim v3.2.0
  - Benchmark Suit: Rodinia, PolyBench, ISPASS, Tango
  - 性能指标：每周期执行的指令数量（Instruction per cycle, IPC）

2.3 实验结果





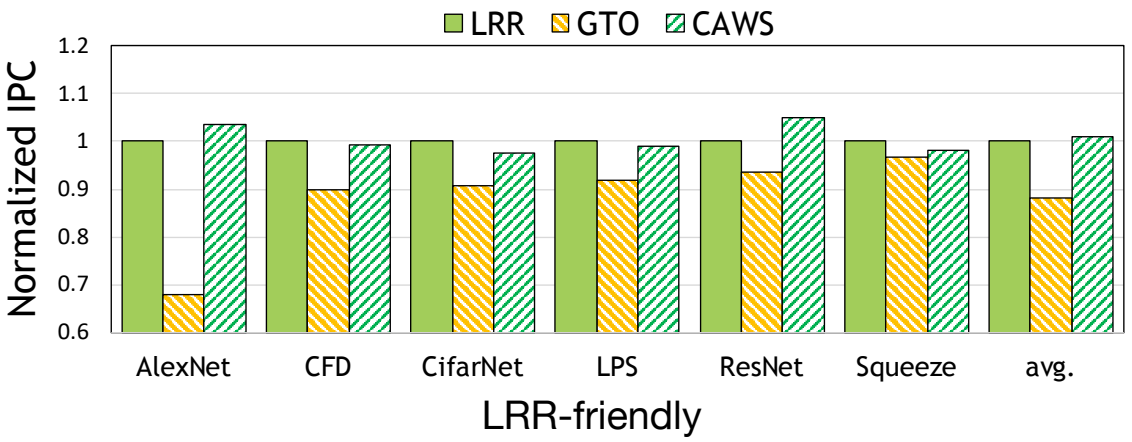


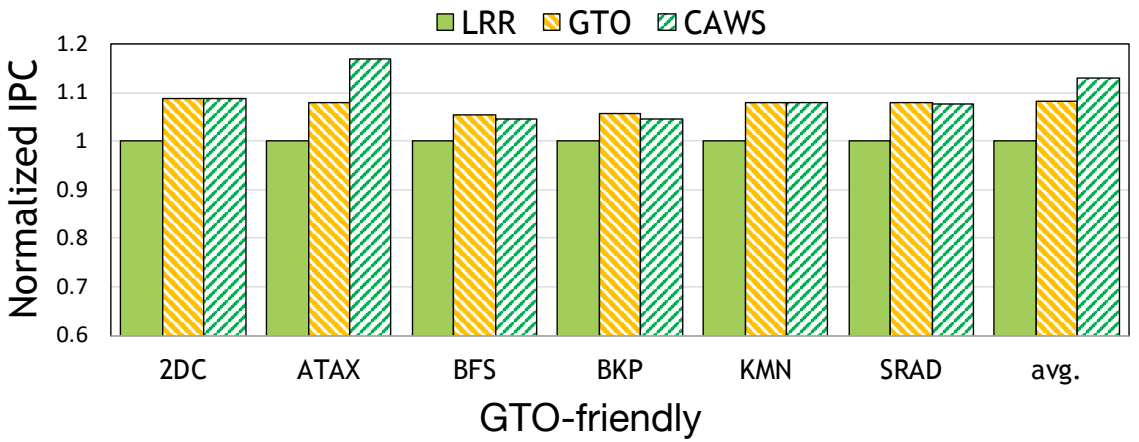
- ResNet，AlexNet，CAWS 好于 LRR 和 GTO
- CifarNet，CAWS接近 LRR 的性能

2.3 实验结果

- Benchmark 分类
  - LRR-friendly：使用 LRR 调度策略获得更好的性能
  - GTO-friendly：使用 GTO 调度策略获得更好的性能

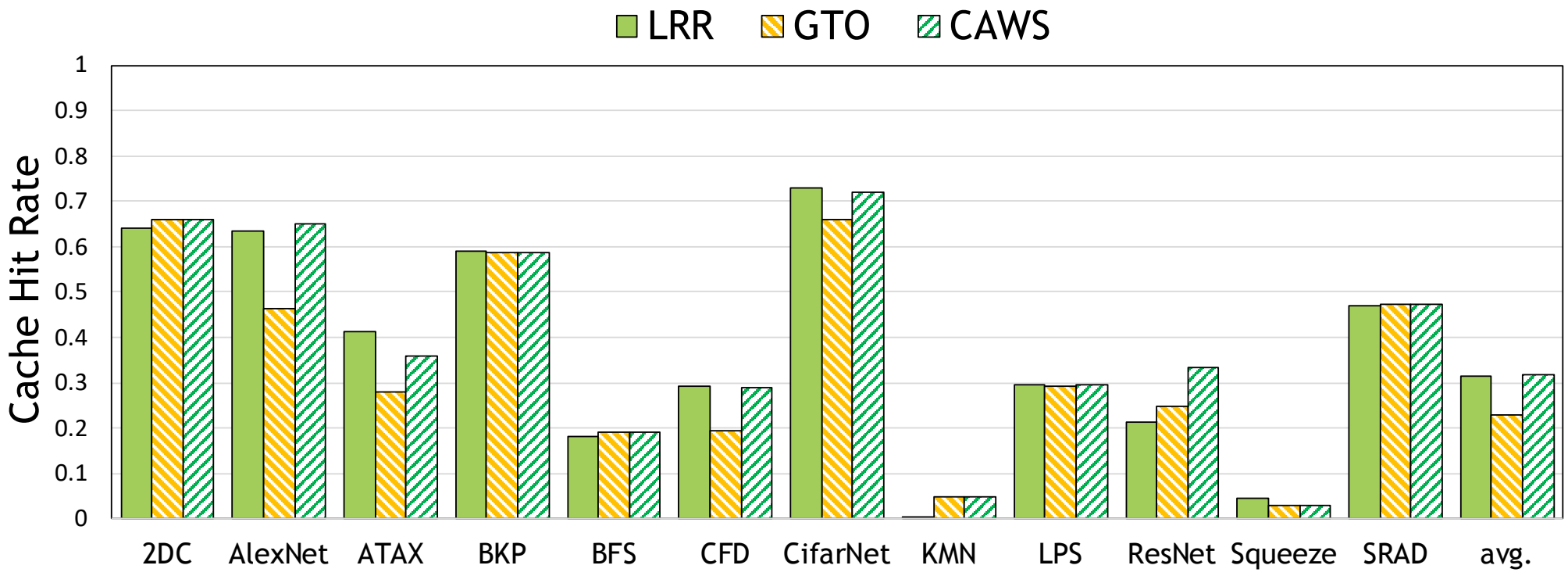
- 整体性能分析
  - 在大多数应用中，CAWS 能够达到更好的调度策略的性能





2.3 实验结果

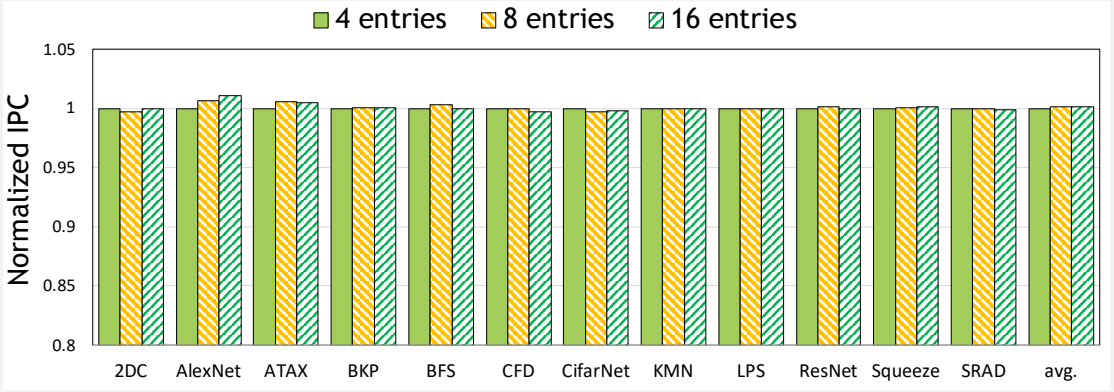
- Cache 命中率分析
  - 在大多数应用中，CAWS 能够达到更好的调度策略的 cache 命中率
  - 本质上，CAWS 是优化了 cache 命中来提升性能



2.4 开销分析

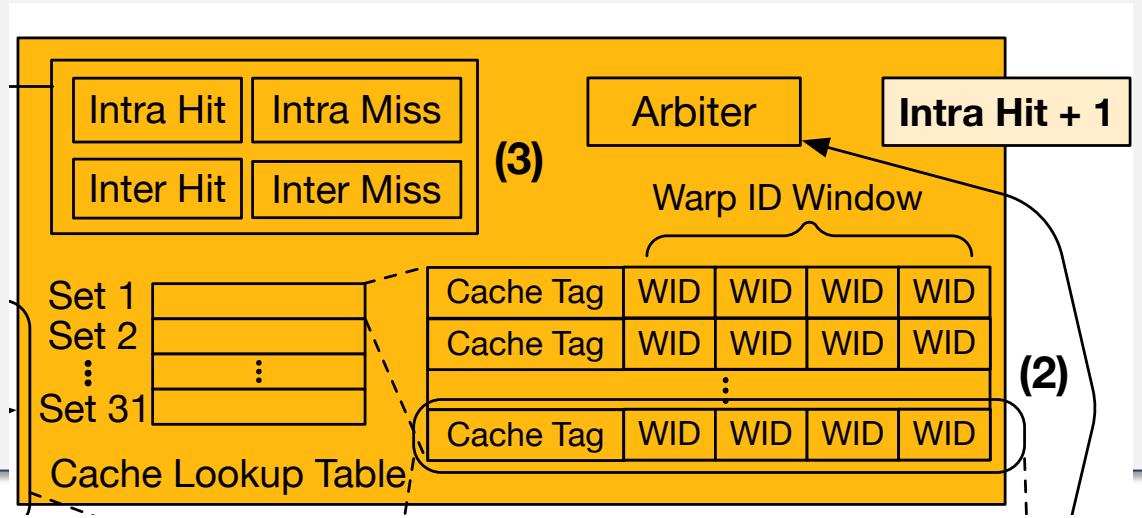
查找表 entry 数量的影响

- 总的来说，增加表格条目对性能的影响不大
- 4-8 个表格条目足够



硬件开销分析

- 查找表
  - 每个表格条目：20 (cache tag) + 4 x 6 (线程束 ID) = 44 bits
  - 32 cache sets, 每个 cache set 包含8个表格条目
  - 总计：32 x 8 x 44 = 11264 bits = 1408 Bytes
- 仲裁器、计数器等硬件



2.5 小结

- 总结
  - 不同应用工作负载不同，单一的线程束调度策略无法满足需求
  - LRR 和 GTO 两种调度策略，分别能够匹配两种缓存局部性的类型
  - CAWS 能够找到匹配缓存局部性类型的调度算法

- 不足
  - 缓存局部性查找表的硬件开销大 -> 片上一张 1.4KB 的表
  - 实验中使用的 GPU 架构比较老 -> Fermi
  - 没有和当前比较前沿的调度算法进行对比 -> Baseline LRR, GTO
  - 没有做能耗评估

答辩提纲

一 研究背景及意义

二 自适应线程束调度算法

三 基于指令控制码的调度

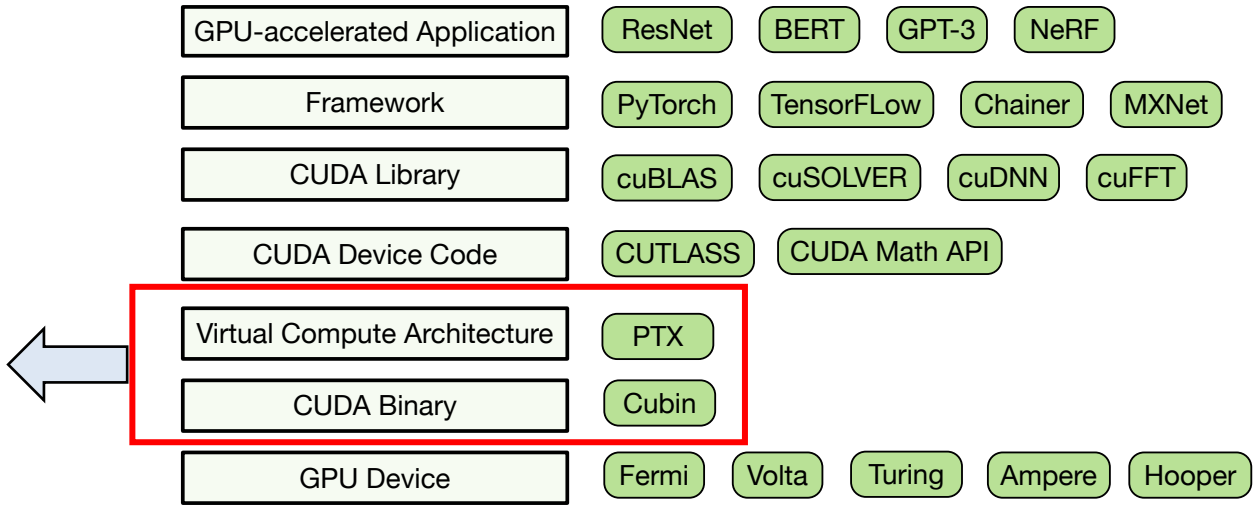
四 论文总结

3.1 背景

- 指令控制码 (control code)
  - 从 NVIDIA Kepler (2012) 架构开始，汇编代码中包含指令控制码
  - 编译器协助硬件处理数据依赖、指令调度，节省硬件资源
  - 从 NVIDIA Volta (2017) 架构开始，每条汇编指令后都会有指令控制码，相当于指令长度从 64 bits -> 128 bits

Listing 1.1　SASS 汇编代码

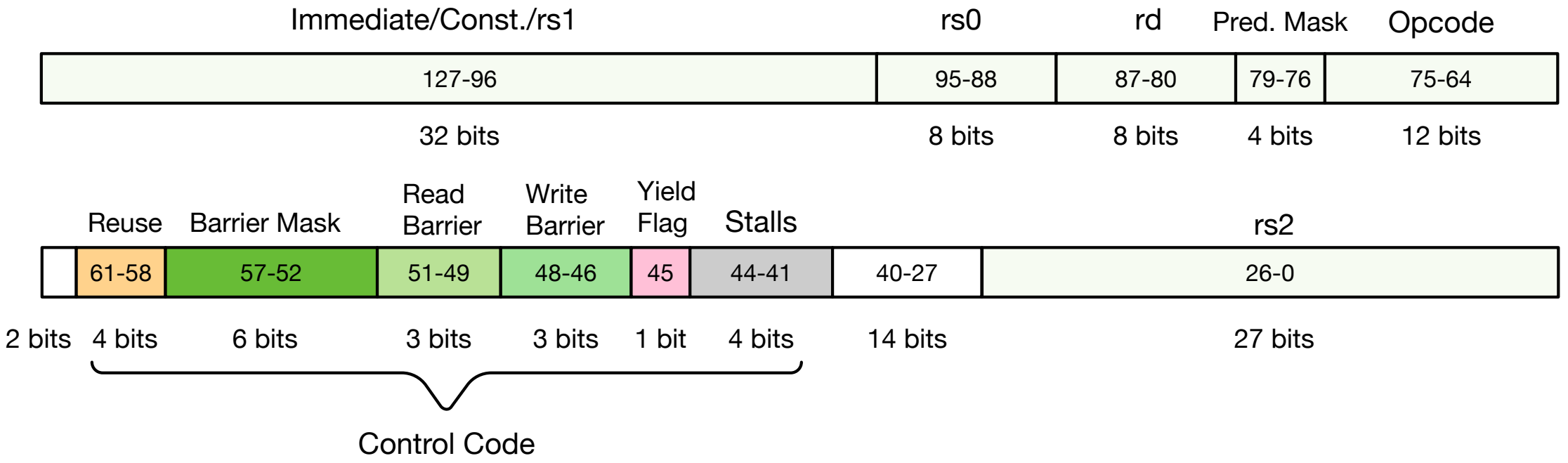
```
/*17d0*/ CS2R R40, SRZ;                  /* 0x0000000000287805 */
                                         /* 0x000fe2000001ff00 */
/*17e0*/ ULDC.64 UR8, c[0x0][0x188];     /* 0x0000620000087ab9 */
                                         /* 0x000fe20000000a00 */
/*17f0*/ LDS.U.128 R12, [R46];           /* 0x000000002e0c7984 */
                                         /* 0x000e620000001c00 */
```



3.1 背景

Optimizing batched winograd convolution on GPUs, PPoPP'20
Control code: https://github.com/NervanaSystems/maxas/wiki/Control-Codes, Scott
Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking

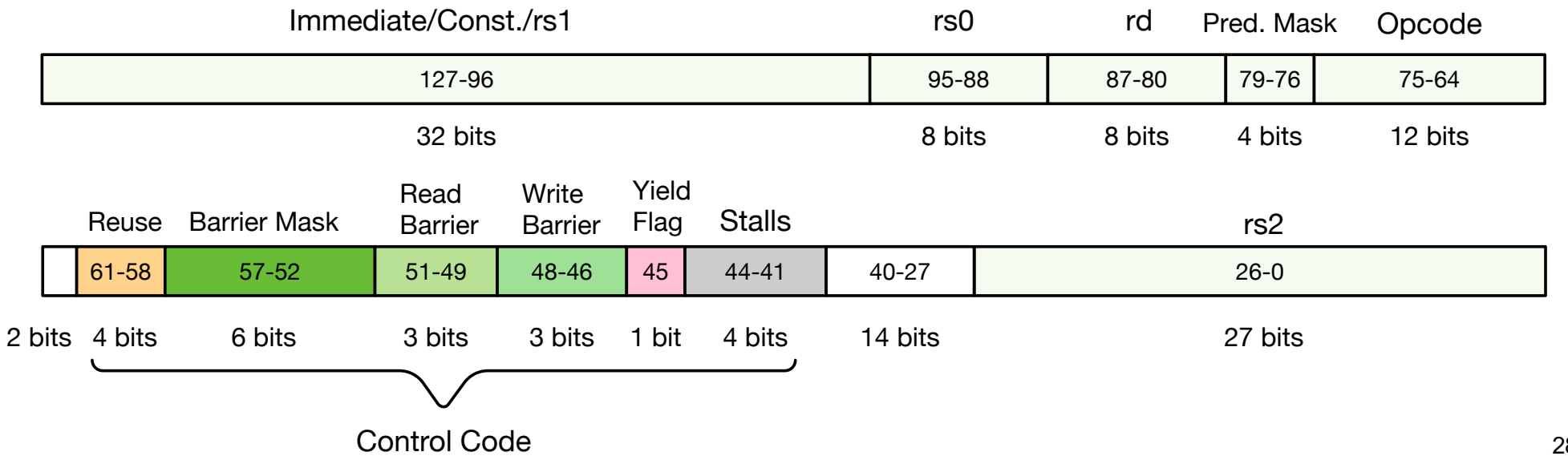
- 指令控制码作用解析
  - Yield flag hint
  - 设为1，当前线程束执行指令后，倾向于让出调度权
  - 设为 0，当前线程束指令指令后，倾向于继续执行下一条指令



3.1 研究背景

Optimizing batched winograd convolution on GPUs, PPoPP'20
Control code: https://github.com/NervanaSystems/maxas/wiki/Control-Codes, Scott
Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking

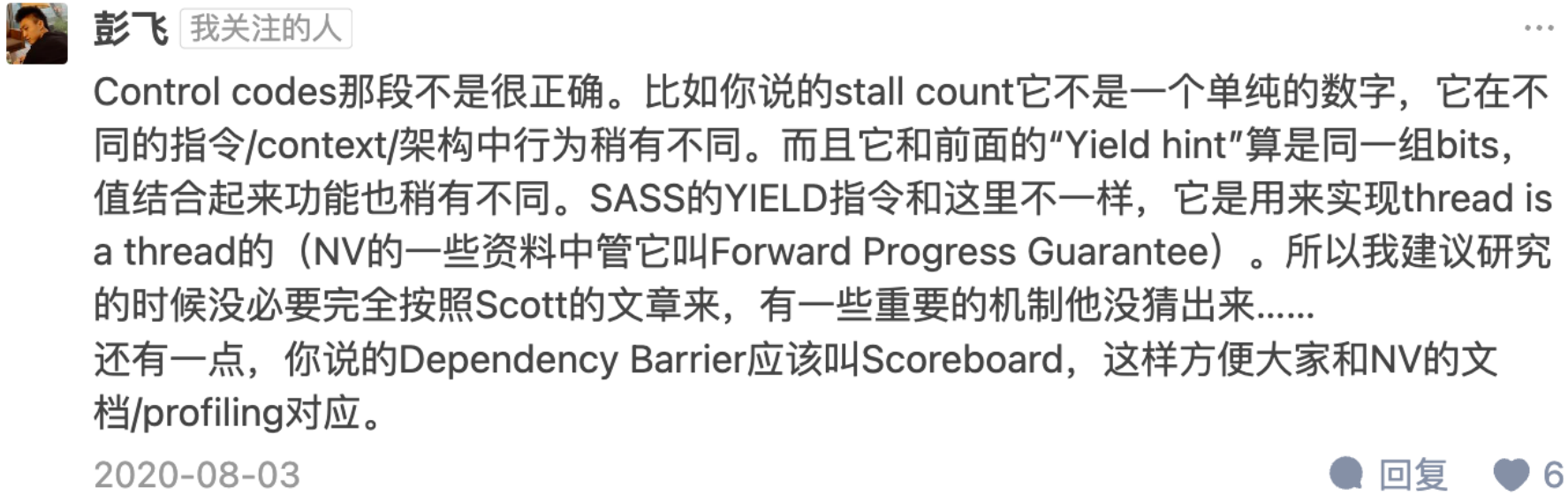
- 指令控制码作用解析
  - Reuse: 如果当前指令某个 slot 的 register 还会被下一个指令的同一个 slot 读取，那就可以reuse当前指令读取到的register内容。减少 register bank conflict
  - Stalls / stall count: 指令的 latency
  - Read barrier / write barrier / barrier mask: 处理 dependency



3.1 研究背景

Optimizing batched winograd convolution on GPUs, PPoPP'20
Control code: https://github.com/NervanaSystems/maxas/wiki/Control-Codes, Scott
Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking

- 指令控制码作用解析
  - https://zhuanlan.zhihu.com/p/166180054



彭飞 我关注的人 ···

Control codes那段不是很正确。比如你说的stall count它不是一个单纯的数字，它在不同的指令/context/架构中行为稍有不同。而且它和前面的"Yield hint"算是同一组bits，值结合起来功能也稍有不同。SASS的YIELD指令和这里不一样，它是用来实现thread is a thread的（NV的一些资料中管它叫Forward Progress Guarantee）。所以我建议研究的时候没必要完全按照Scott的文章来，有一些重要的机制他没猜出来……
还有一点，你说的Dependency Barrier应该叫Scoreboard，这样方便大家和NV的文档/profiling对应。

2020-08-03 回复 6

3.2 一些相关研究

- GEMM 加速
  - UGEMM (Wu etc., ISCA, 2020)
  - SIMD^2 (Zhang etc., ISCA, 2022)
  - Flexible Performant GEMM Kernels on GPUs (Thomas etc., TPDS, 2022)

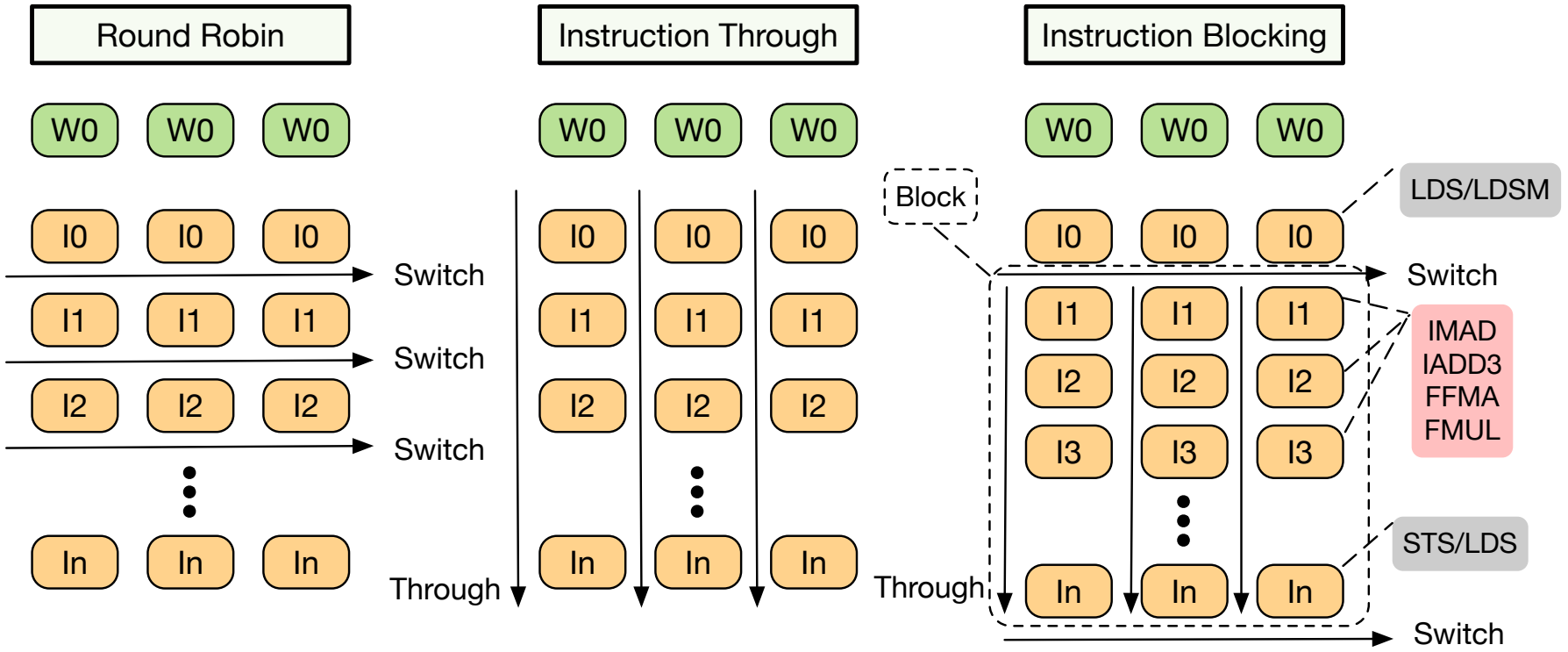
- Tensor core/systolic array
  - Tensorox (Ho etc., TPDS, 2022)
  - Dual-side Sparse Tensor Core (Wang etc., ISCA, 2021)
  - Balancing Efficiency and Flexibility for DNN Acceleration via Temporal GPU-Systolic Array Integration (Guo etc., DAC, 2020)
  - Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply (Yan etc., IPDPS, 2020)

- 汇编指令优化
  - Optimizing batched winograd convolution on GPUs (Yan etc., PPoPP, 2020)

大多数都是算法或者架构层面的加速，instruction scheduling 是否有效?

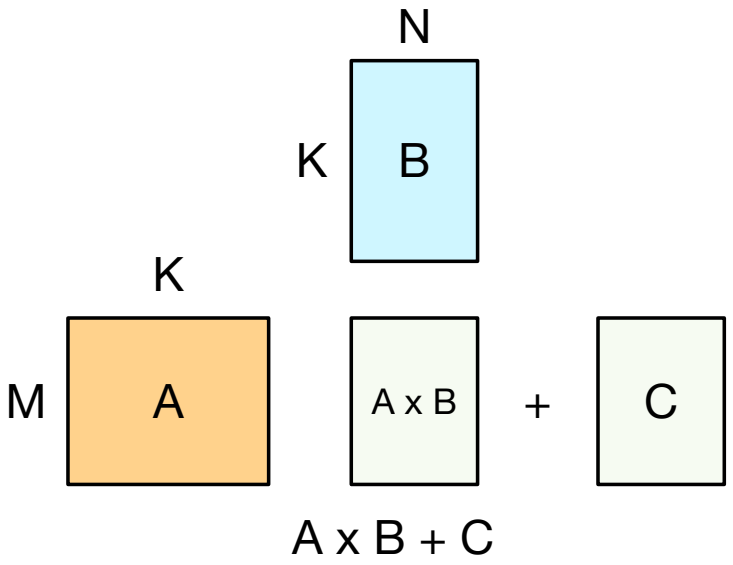
3.3 指令调度优化

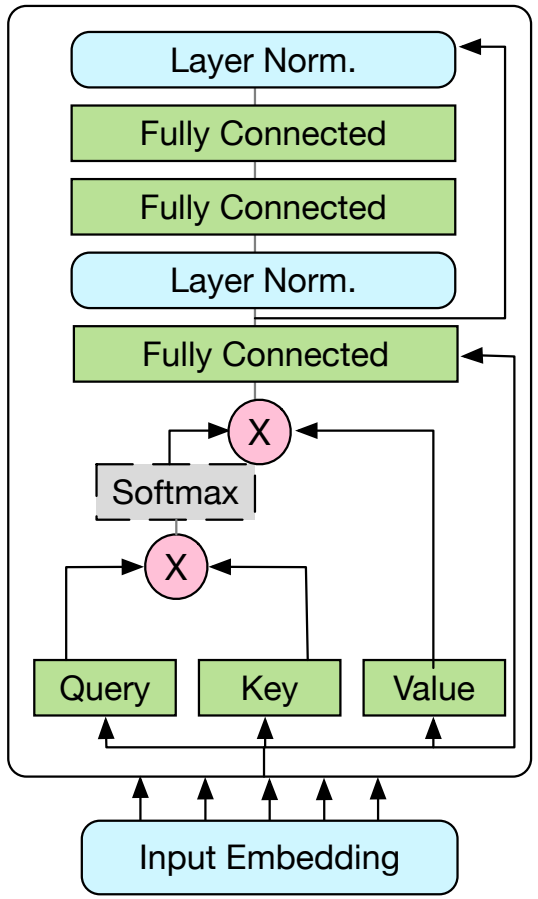
- NAÏVE, 由编译器决定；RR (Round-Robin)；IT (Instruction Through)
- IB (Instruction Blocking)
  - 考虑到 GEMM 的计算特性，当指令从共享内存加载数据时让出调度权
  - 进行乘加运算的时候连续执行指令



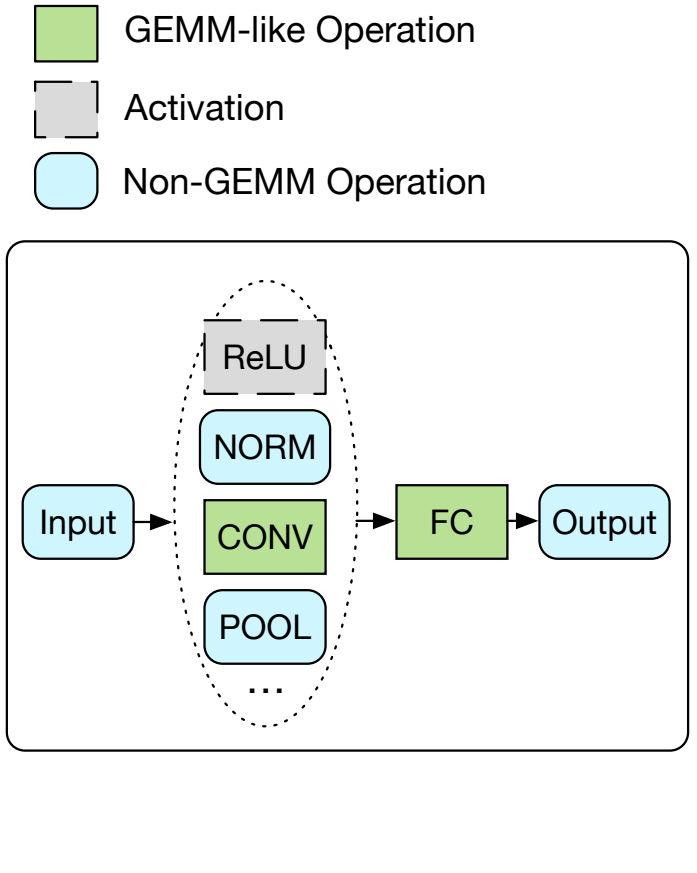
3.4 实验对象

- 测试对象：GEMM（通用矩阵乘法）
  - Transformer、CNN 模型的重要算子
  - 计算密集型和访存密集型





Transformer-based Model Layer



CNN-based Model Layer

3.4 实验结果

实验环境

- ❑ CUTLASS v2.11，矩阵乘法计算库
- ❑ CuAssembler，汇编器
- ❑ NVIDIA RTX2080 Ti

https://github.com/cloudcores/CuAssembler

硬件的峰值计算性能

数据类型	计算单元	峰值计算性能/TFLOPS
FP32	CUDA Core	14.2
FP16	CUDA Core	28.5
FP16 with FP32 Accumulator	Tensor Core	56.9
FP16 with FP16 Accumulator	Tensor Core	113.8
INT8	Tensor Core	227.7
INT4	Tensor Core	455.4

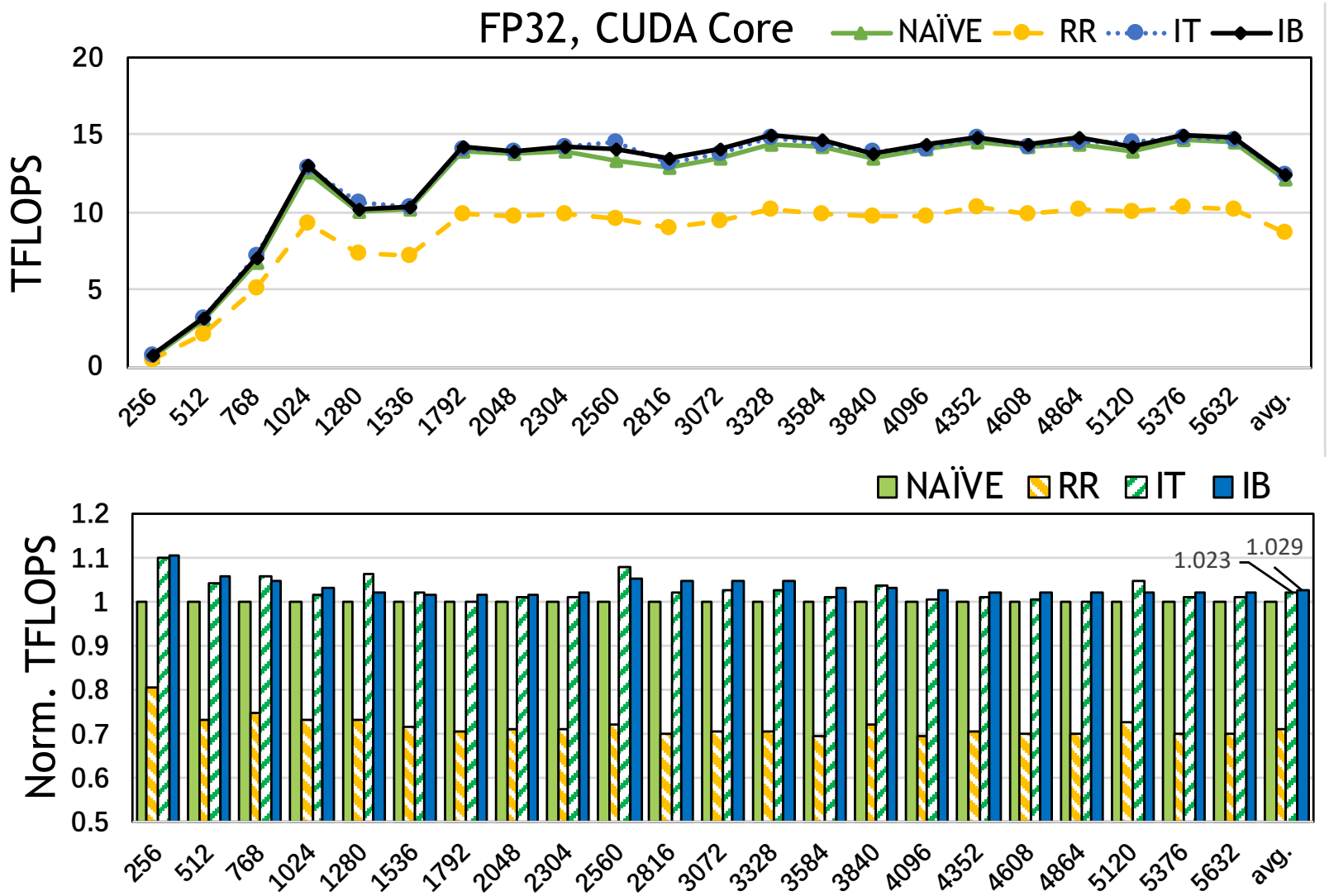
3.4 实验结果

- FP32 GEMM
- RR 存在明显的性能下降
- 其他三种方式区别不大

硬件的峰值计算性能

数据类型	计算单元	峰值计算性能/TFLOPS
FP32	CUDA Core	14.2
FP16	CUDA Core	28.5
FP16 with FP32 Accumulator	Tensor Core	56.9
FP16 with FP16 Accumulator	Tensor Core	113.8
INT8	Tensor Core	227.7
INT4	Tensor Core	455.4

3.4 实验结果


FP32, CUDA Core
NAÏVE
RR
IT
IB
TFLOPS
20
15
10
5
0
256
512
768
1024
1280
1536
1792
2048
2304
2560
2816
3072
3328
3584
3840
4096
4352
4608
4864
5120
5376
5632
avg.
NAÏVE
RR
IT
IB
Norm. TFLOPS
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
1.023
1.029

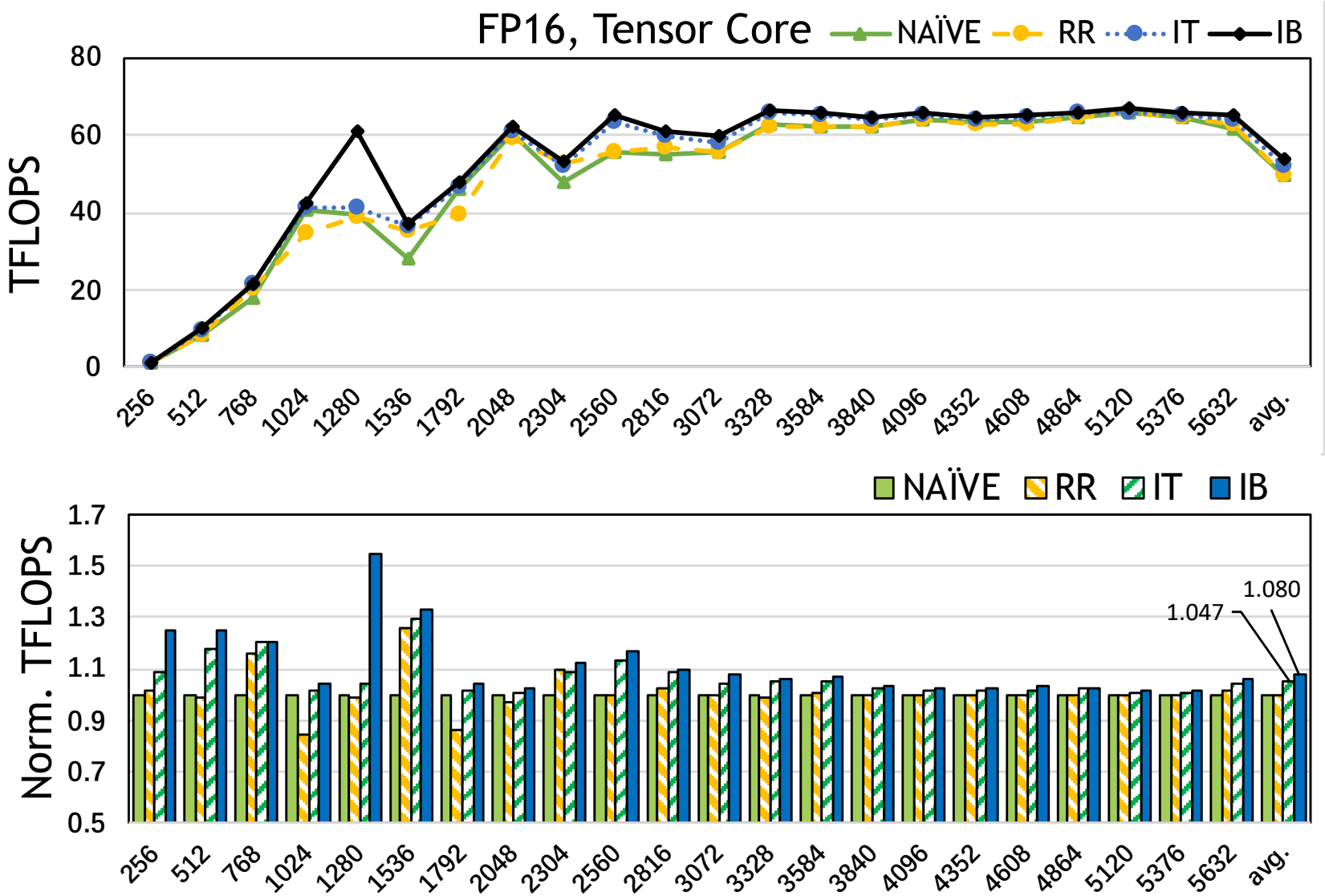
3.4 实验结果

- FP16 GEMM
- IB 相对 NAIVE 平均提升了 8% 的性能，IT 相比 NAIVE 平均提升了 4.7% 的性能
- 这部分提升主要来源于矩阵尺寸 256-3328 的 GEMM。

硬件的峰值计算性能

数据类型	计算单元	峰值计算性能/TFLOPS
FP32	CUDA Core	14.2
FP16	CUDA Core	28.5
FP16 with FP32 Accumulator	Tensor Core	56.9
FP16 with FP16 Accumulator	Tensor Core	113.8
INT8	Tensor Core	227.7
INT4	Tensor Core	455.4

3.4 实验结果


FP16, Tensor Core
NAÏVE
RR
IT
IB
TFLOPS
80
60
40
20
0
256
512
768
1024
1280
1536
1792
2048
2304
2560
2816
3072
3328
3584
3840
4096
4352
4608
4864
5120
5376
5632
avg.
NAÏVE
RR
IT
IB
Norm. TFLOPS
1.7
1.5
1.3
1.1
0.9
0.7
0.5
1.047
1.080

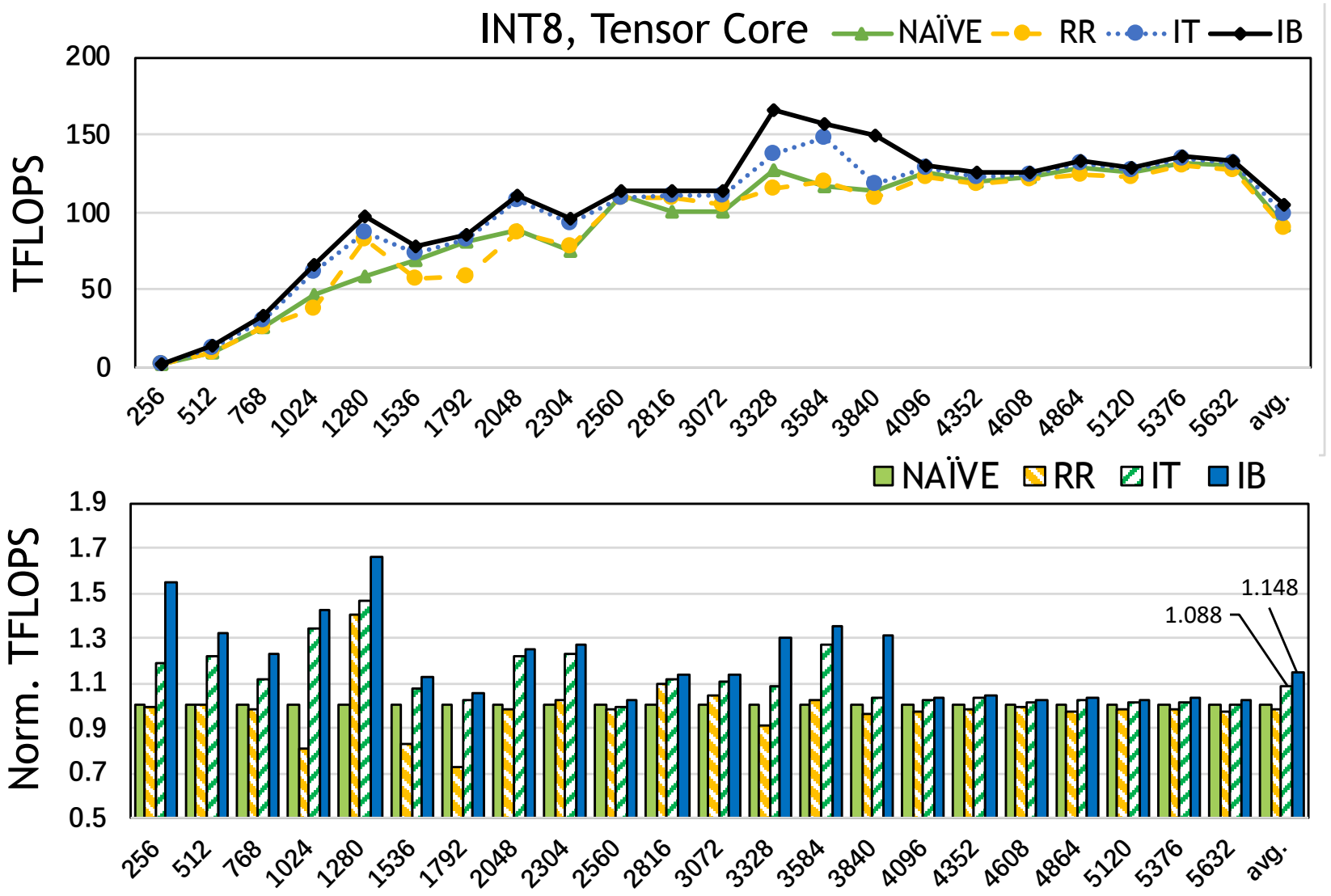
3.4 实验结果

- INT8 GEMM
- IB 和 IT 的性能比较接近，但是在矩阵尺寸为 1280、 3328 和 3840 时 IB 有比较明显的优势。
- 相比 NAÏVE，IB 平均提升 14.8% 的性能

硬件的峰值计算性能

数据类型	计算单元	峰值计算性能/TFLOPS
FP32	CUDA Core	14.2
FP16	CUDA Core	28.5
FP16 with FP32 Accumulator	Tensor Core	56.9
FP16 with FP16 Accumulator	Tensor Core	113.8
INT8	Tensor Core	227.7
INT4	Tensor Core	455.4

3.4 实验结果


INT8, Tensor Core
NAÏVE
RR
IT
IB
TFLOPS
200
150
100
50
0
256
512
768
1024
1280
1536
1792
2048
2304
2560
2816
3072
3328
3584
3840
4096
4352
4608
4864
5120
5376
5632
avg.
NAÏVE
RR
IT
IB
Norm. TFLOPS
1.9
1.7
1.5
1.3
1.1
0.9
0.7
0.5
1.148
1.088

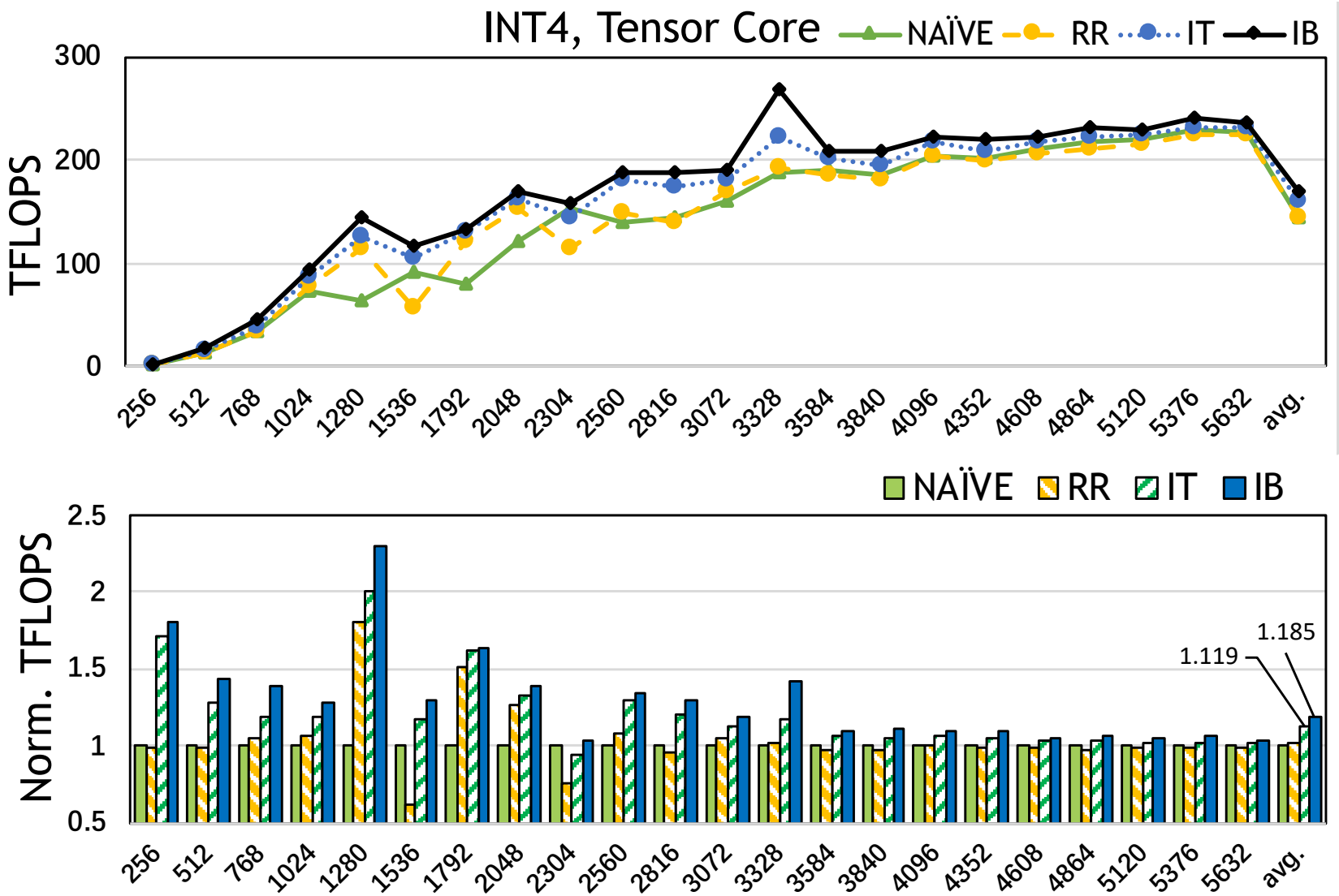
3.4 实验结果

- INT4 GEMM
- 相比 NAÏVE，IB 平均提升 18.5% 的性能
- 性能优势主要来源于矩阵尺寸 256-3072 之间
- 难以达到算法优化层面能够达到的最好性能，这给汇编指令层面的优化带来了机会。

硬件的峰值计算性能

数据类型	计算单元	峰值计算性能/TFLOPS
FP32	CUDA Core	14.2
FP16	CUDA Core	28.5
FP16 with FP32 Accumulator	Tensor Core	56.9
FP16 with FP16 Accumulator	Tensor Core	113.8
INT8	Tensor Core	227.7
INT4	Tensor Core	455.4

3.4 实验结果


INT4, Tensor Core
NAÏVE
RR
IT
IB
TFLOPS
300
200
100
0
256
512
768
1024
1280
1536
1792
2048
2304
2560
2816
3072
3328
3584
3840
4096
4352
4608
4864
5120
5376
5632
avg.
NAÏVE
RR
IT
IB
Norm. TFLOPS
2.5
2
1.5
1
0.5
1.119
1.185

3.5 小结

- 总结
  - 指令控制码能够影响线程束的调度，从而影响矩阵乘法/应用的性能

- 不足
  - Control code 的作用只是一种推测，并非 NV 的官方说明
  - 指令控制码重排的算法比较简单
  - 性能的提升不够显著 -> 几乎没有提升
  - 测试对象比较单一（only GEMM）

答辩提纲

一 研究背景及意义

二 自适应线程束调度算法

三 基于指令控制码的调度

四 整体总结

4.1 整体的总结

- Limitation
  - Warp scheduling 可能不是 GPU 性能的瓶颈
  - Control code 的作用都是基于 micro-bench 的分析，不是官方的文档
  - 一些场景可能随着 GPU 架构的迭代失效
  - NV 的 warp scheduler 是一个黑盒，难以得知其调度的细节

- 一些可能可做的拓展，解决真正的瓶颈
  - 上层应用带来的瓶颈：大模型 memory bound；多任务/kernel-fusion 场景下的资源利用率 / occupancy；GPU 不友好的 irregular application
  - 架构变化带来的瓶颈：异构架构