Reduce, Scan 和 Sort 算子的 Triton 实现

&Block 间通信

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

Reduce, Scan 和 Sort 算子的 Triton 实现

01 Recap: Triton Lang

02 Reduce

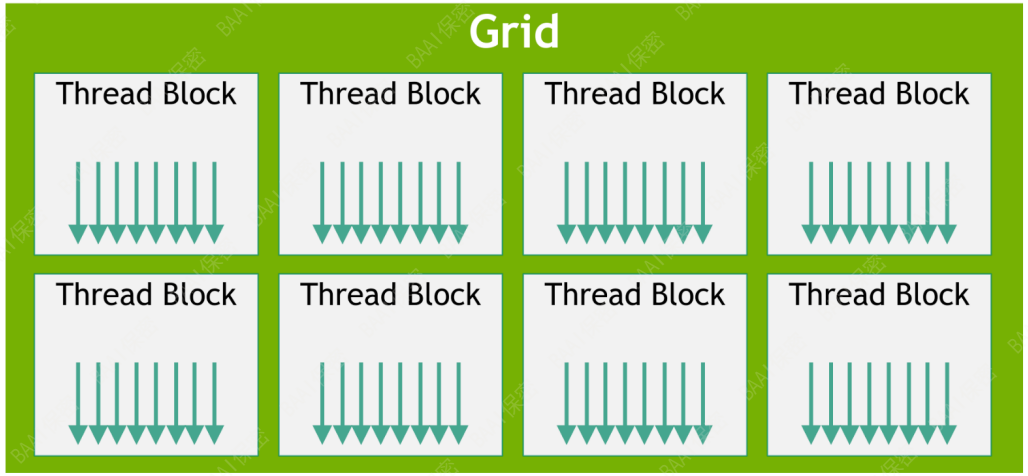
03 Scan

04 Sort

05 Message Passing between Blocks

Triton Lang 基本介绍

Triton Lang 是一门**面向 tile** 的 **Block 级**的编程语言。



面向 Tile

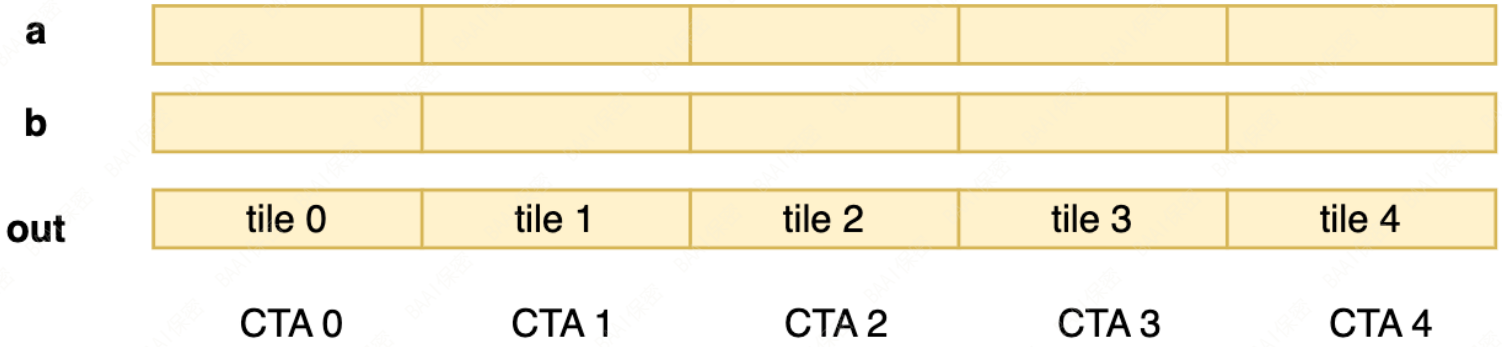
- 数据以块(tile)的形式呈现，编译器负责处理到 warp/thread 的映射；

Block 级编程：

- Triton Lang 的运算都是定义在 tile 上，由整个 block (CTA) 来执行。tile 的产生(tl.arange, tl.zeros 等), 数据加载(tl.load), 保存(tl.store), 各种定义在 tile 上的运算(pointwise, reduce, scan, gemm 等)
- Triton jit function 描述的也是一个 block 的运算，而不必细分到每个 thread.
- Triton jit function 保留了一些和 cuda 相关的细节。
  - launch 的时候需要指定 grid size(而且是 3d 的 grid，直接和 cuda kernel 的 grid 对应)
  - 需要通过 num_warps 指定 thread block 的线程数。 (32 threads/warp)

Vector Add

- 最简单的 parallel 问题
- vector 上分块，分出 5 个 tile, 然后把这些 tile 分给不同的 Block(CTA) 去计算。

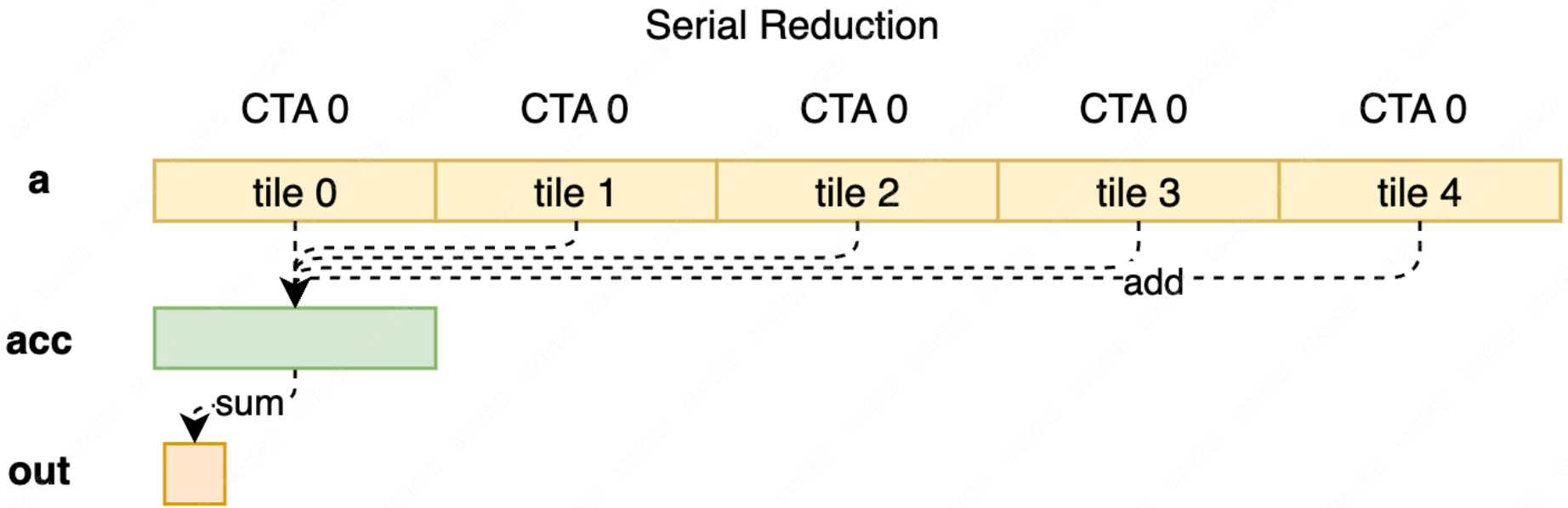



```
@triton.jit
def add_kernel(a_ptr, b_ptr, o_ptr, N, TILE_SIZE: tl.constexpr):
    pid = tl.program_id(0)
    offsets = pid * TILE_SIZE + tl.arange(0, TILE_SIZE)
    mask = offsets < N

    a = tl.load(a_ptr + offsets, mask=mask)
    b = tl.load(b_ptr + offsets, mask=mask)
    o = a + b
    tl.store(o_ptr + offsets, o, mask=mask)
```

https://github.com/iclementine/Code_for_presentation/tree/master/reduce-scan-sort-and-message-passing-between-blocks

Sequantial Reduce

- 对输入 vector 分块，共 5 个 tile, 都分配给同一个 CTA 计算。如果有足够的 batch size 来并行，这也算是一个可行的算法。但从 tile 间它其实不是一个 parallel reduction.
- CTA内部有并行，但从 CTA 级别来看，算法是 Serial 的)



北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

Reduce, Scan 和 Sort 算子的 Triton 实现

01 Recap: Triton Lang

02 Reduce

03 Scan

04 Sort

05 Message Passing between Blocks

Parallel Reduce

Serial Reduce:

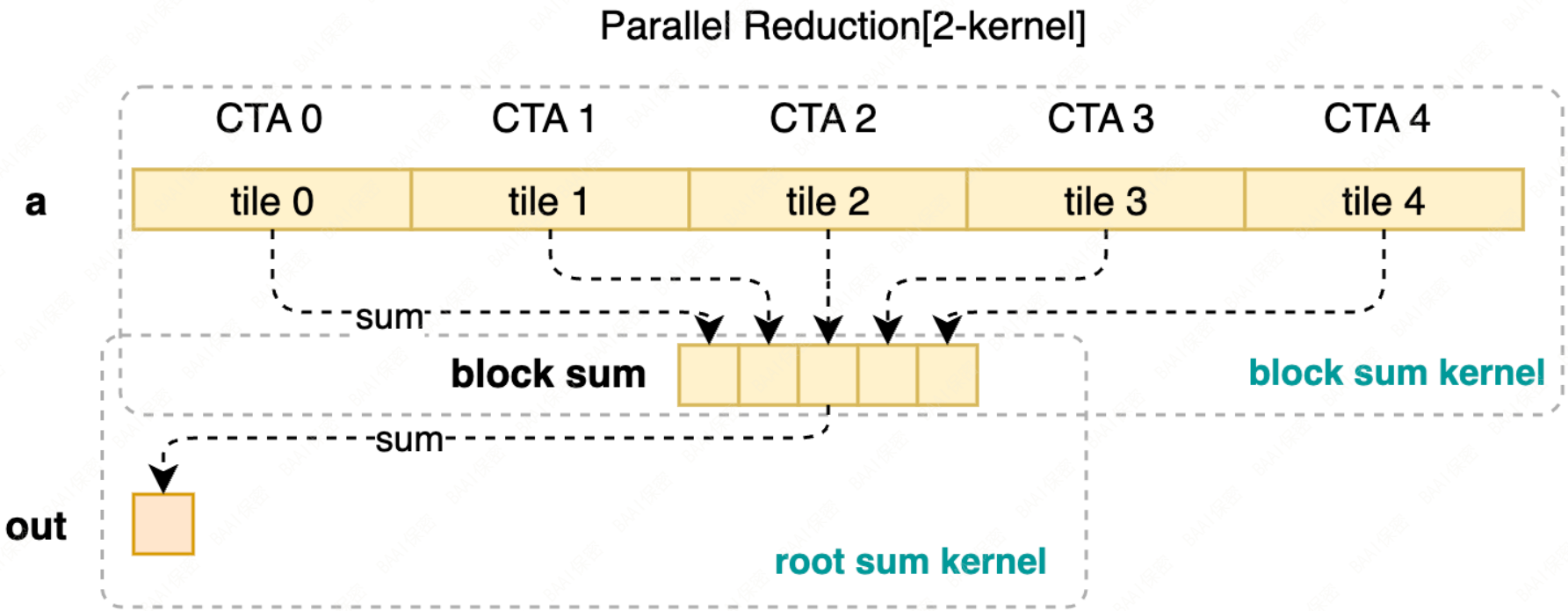
- 根据输出数据分块，然后交由不同的 CTA 计算。各个 CTA 之间无需通信，独立计算。
- Reduce 运算仍然是 sequential 的，只是按 tile sequential.

Parallel Reduce:

- 任务划分更细：需要多个 CTA 计算不同的部分，最后共同作用于一个输出 tile.
- 多个 CTA 的结果合并需要通信。因此程序分为 **CTA 内**和 **CTA 间**两部分。(Intra-Block, Inter-Block)

Parallel Reduce[2-kernel]

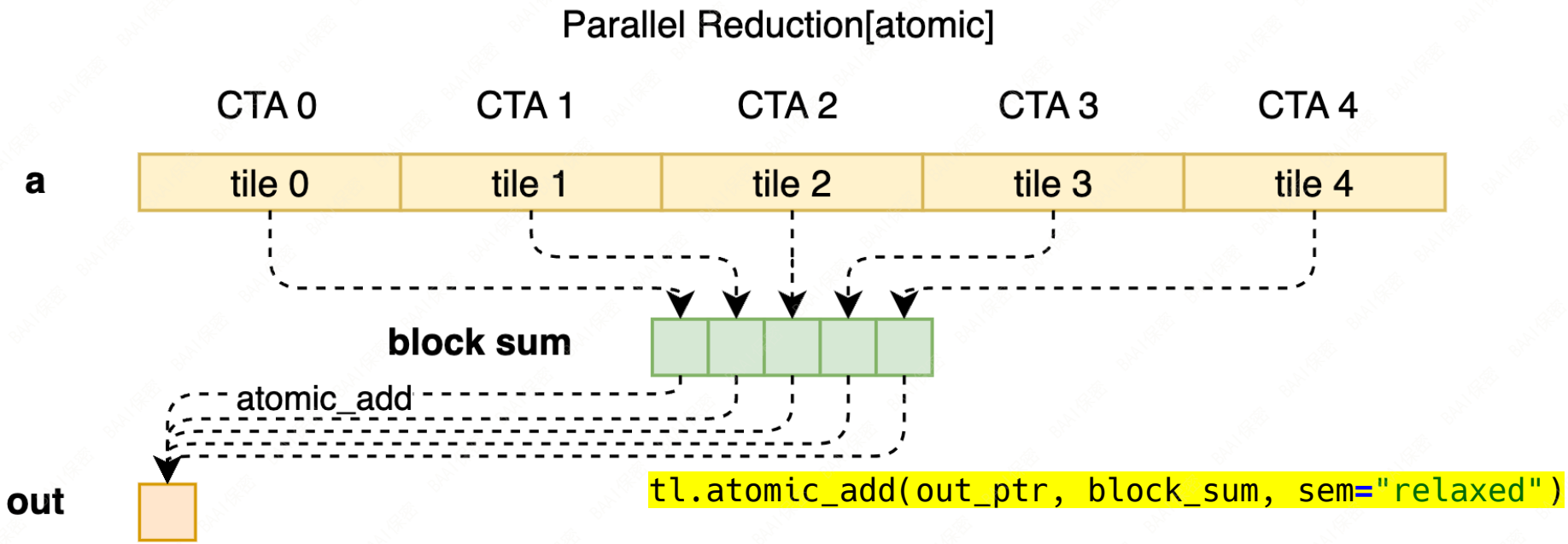
- **block sum kernel**: 对输入 vector 分块，共 5 个 tile, 分配给不同的 CTA 计算。
- **root sum kernel**: 最后启动仅包含一个 CTA 的 sum 把中间结果加起来。（两部分程序很相似）



- 序列很长，经过一次 reduce 之后还是无法用一个 CTA 处理怎么办?
  - block sum kernel 可以改成 serial reduce kernel 处理大的范围。
  - 其实也可以递归, 每次 block sum 之后，size 变成 1/tile_size.

Parallel Reduce[atomic]

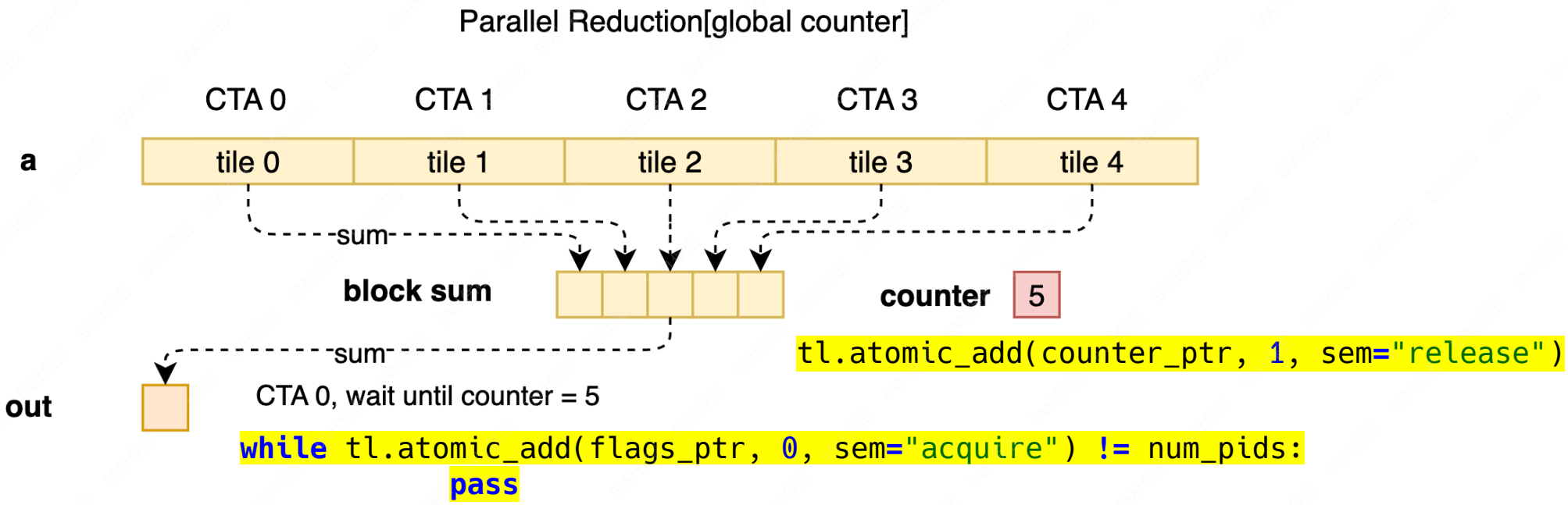
- 对输入 vector 分块，共 5 个 tile, 分配给不同的 CTA 计算。(下面的代码每个 CTA 处理连续多个 tile)
- 需要 CTA 之间需要以合并 local sum.（比如通过在 global memory 上 atomic_add)



Parallel Reduce[global counter]

对输入 vector 分块，共 5 个 tile, 分配给不同的 CTA 计算。(下面的代码每个 CTA 处理连续多个 tile)

- **block sum**: global memory, 每个 cta 一个标量。
- **counter**: global memory, 整个 grid 独有一个，用于表示已经执行完的 CTA 数。



Tips:

- 当需要多个 CTA 共同写共一个地址时，使用 atomic op 保证原子性；
- 当需要用一个 condition block 住 CTA 的执行，直到条件满足，可以用 atomic op 去 poll on a flag.
- 当 block 间合并的运算不能靠 atomic op 实现时，这种做法更具一般性。（比如 softmax)

Benchmark

- size 为 (1024 * 1024 * 1024) 的 fp32 vector

Implementation	Bandwidth（GB/s)
torch	912.494
serial-reduction	42.472
global-counter	914.919
2-kernel	914.959
atomic_add	**915.37**

- size 为 (1024 * 1024) 的 fp32 vector

Implementation	Bandwidth（GB/s)
torch	296.621
serial-reduction	40.377
global-counter	292.100
2-kernel	313.951
atomic_add	**353.384**

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

Reduce, Scan 和 Sort 算子的 Triton 实现

01 Recap: Triton Lang

02 Reduce

03 Scan

04 Sort

05 Message Passing between Blocks

Scan 问题描述

Inclusive Scan 对于长度为 n 的向量 x，计算前缀和。

```
y[i] = x[0] + x[1] + ... + x[i]
```

x	3	1	3	7	5	6	4
y	3	4	7	14	19	25	29

其他变种：

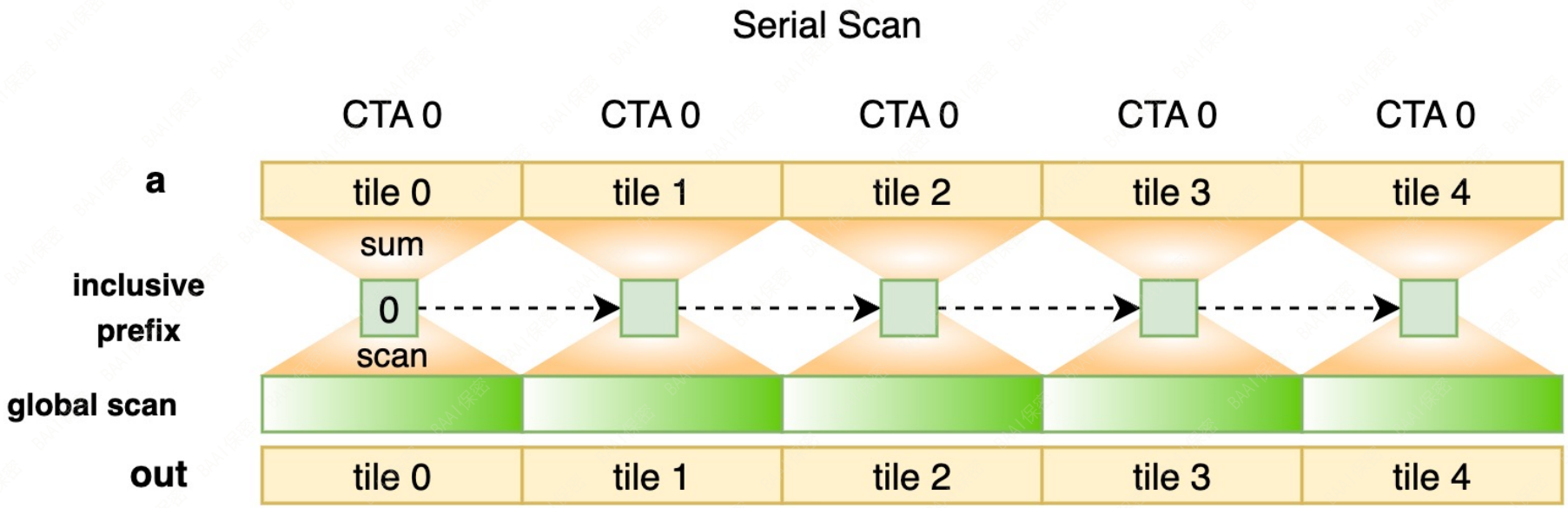
- Exclusive Scan: y[i] 为 x[0, i) 的和，而不是 x[0, i] 的和。相当于错一位。
- Initial Value: 有 initial value 和无 initial value 的区别。
- 可配置的 Binary op: binary op，比如 add, mul, max, min, ...

下面讨论 Inclusive Sum.

Sequential Scan

进入 Parallel Scan 之前，先考虑串行的实现。同一个 CTA 计算所有的 tile, 每个循环节需要计算

- local sum
- global scan = P + local_scan
- 更新 P = exclusive scan of local sum



其实 Block 内的 local_sum 和 local_scan 都可以单独计算的。但是由于计算 local_sum 的 scan 的过程是串行的，就把整个过程串行了，可以通过各种方法将其解开。

Scan-then-Propagate(Scan-Scan-Add)

- Intra Block Scan：块内 Scan, 保存 Block San 和 Block Sum.
- Root Scan：对 Block Sum 进行 Scan(可能包含递归)
- Propagate: 把 Root Scan 结果加到 Block Scan 上。

- 可能包含递归，因为中间的 Root Scan 问题的性质和初始问题相同。
- 可能包含 2n + 1 次 Kernel: n 次 Intra Block Scan + n 次 Propagate + 1 次 Root Scan
- global memory 读写量下限是 4O(n)。
- 可以把 Block Scan 改成 Serial 版，以确保只调用 3 个 Kernel.

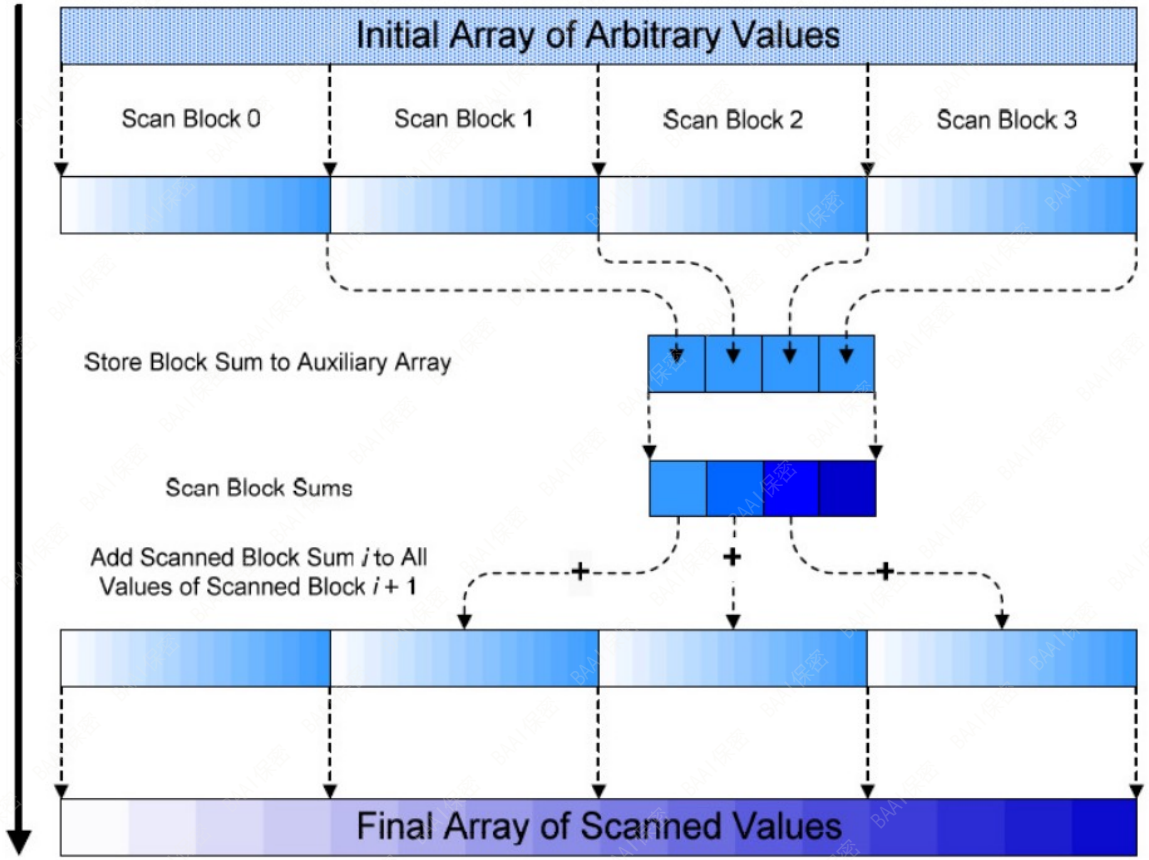
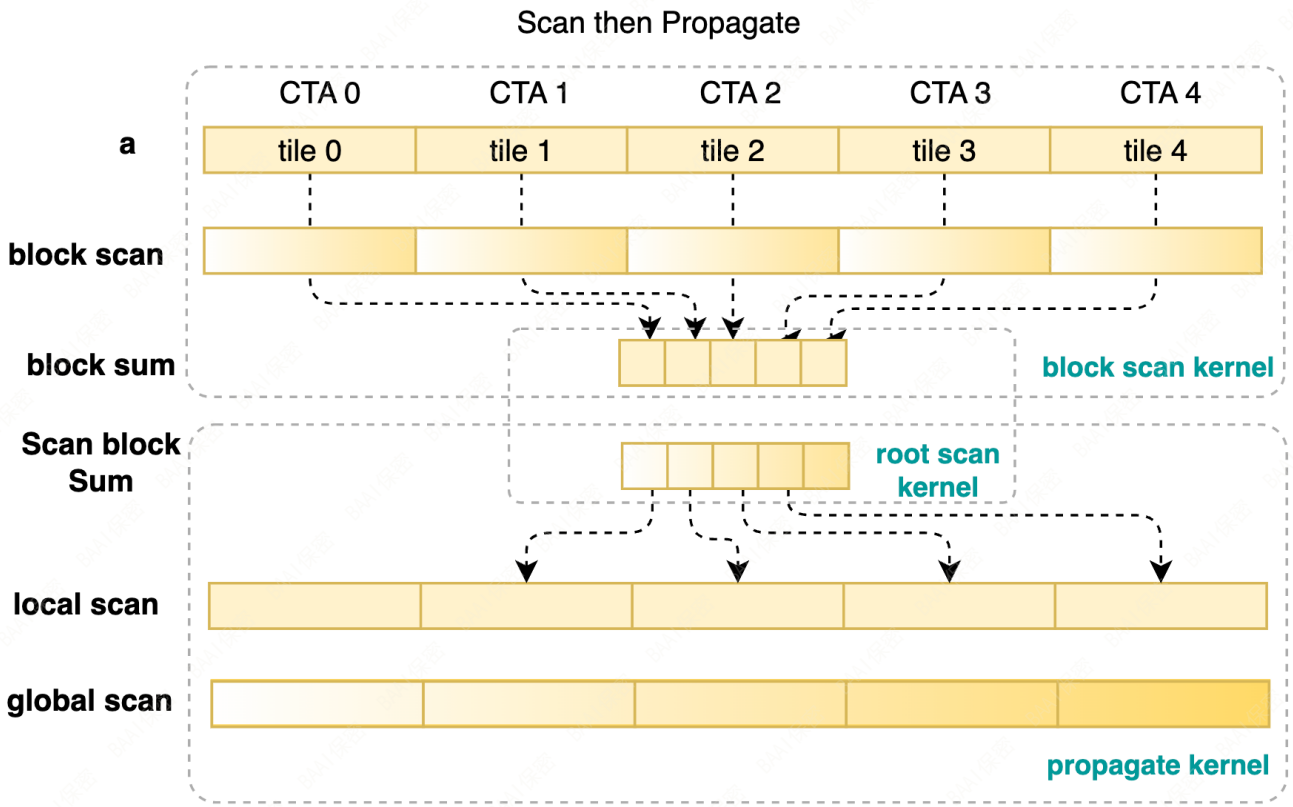
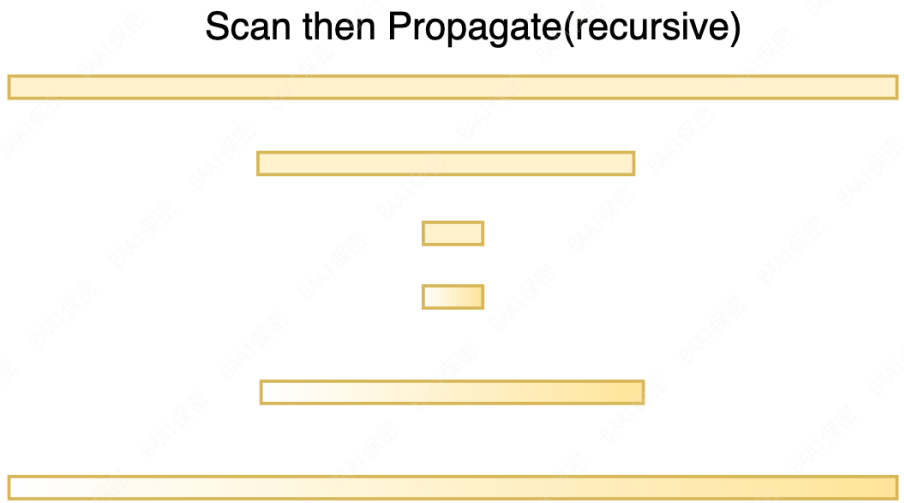


Figure 5: Algorithm for performing a sum scan on a large array of values.

Scan-then-Propagate(Scan-Scan-Add)





- 其实没有必要在第一阶段计算 Block Scan, 因为需要保存又要在第三阶段加载。可以在第三阶段计算从原数组 Block Scan，少一次 O(n) 的写操作。

Reduce-then-Scan(Sum-Scan-Scan)

三 Kernel 版 Reduce-then-Scan，不包含递归。(也可以实现为 递归版)

- Reduce: 每个 CTA 计算块内 reduce; （O(n) 读)
- Root-Scan: 对上述 Block Sum 进行 Scan(确保上一步 Block Sum 个数可以由一个 Block 处理;
- Block Scan: 第 i 个 CTA 计算块内 Scan 以及把 Block Sum 的 Scan 结果中的第 i- 1 个元素加上去。（O(n) 读，O(n) 写)

总共的读写是 3O(n).

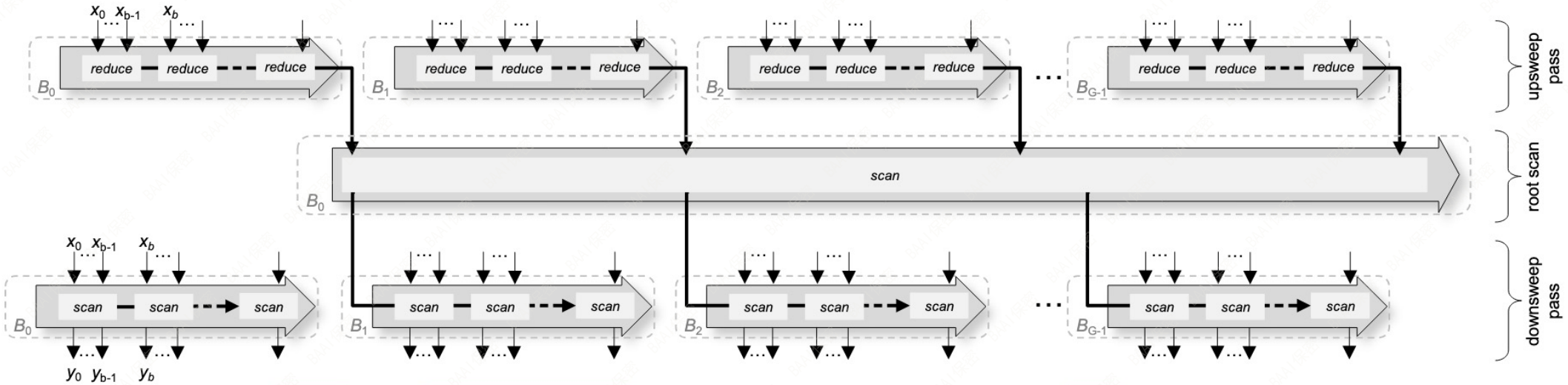
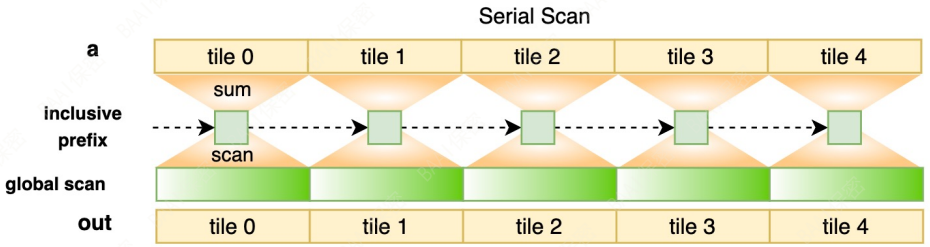
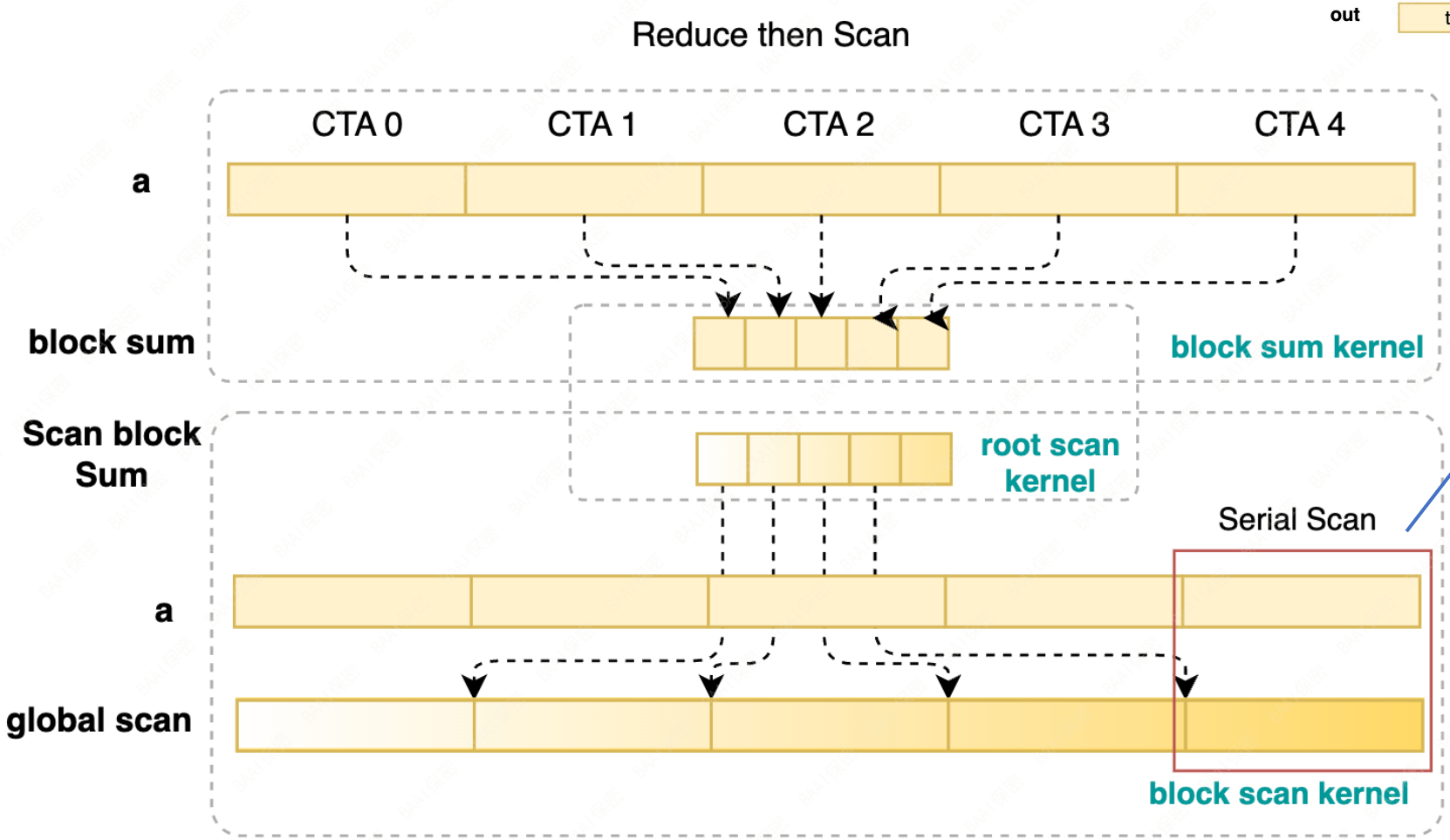


Fig. 3. Three-kernel *reduce-then-scan* parallelization among G thread blocks ($\sim 3n$ global data movement)

Reduce-then-Scan(Sum-Scan-Scan)



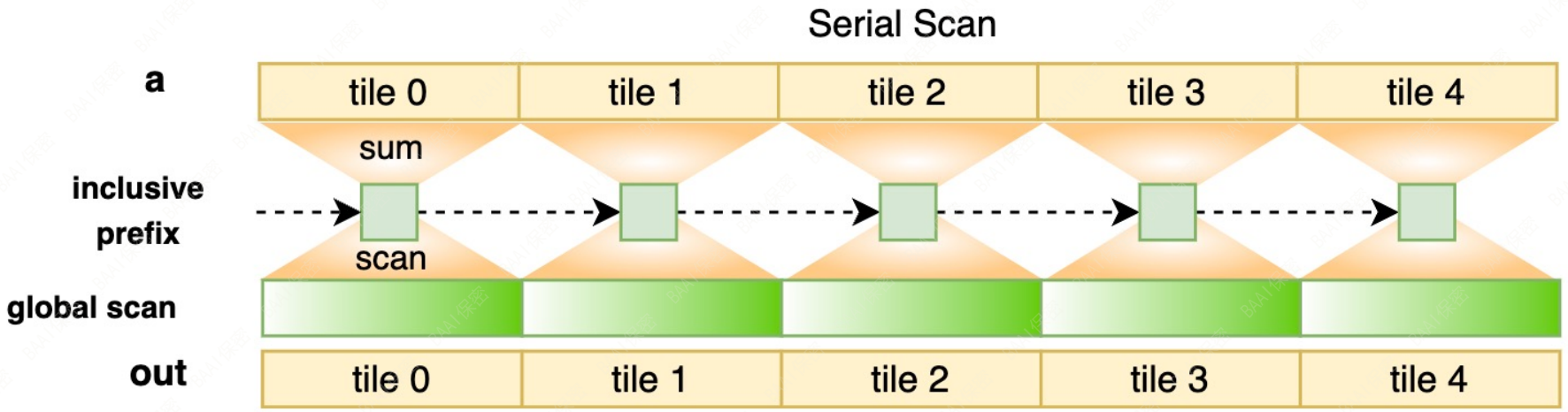


Single-Pass

- Scan-then-propagate 方案需要先计算 Block Scan 存到 Global Memory 在 propagate 阶段再 load 一次。（是否有方法让第一阶段计算出来的 Block Scan 不用写到 Global Memory，节省一次读和写）
- Reduce-then-Scan: 第一阶段不算 Block Scan, 但是第一和第二阶段都需要 Load 一次原数组。(是否可以在 Block Scan 的时候顺便把 Block sum 和 Block 间的 Scan 一起算了，节省一次单独的 Load)

- **能否用一个 kernel 完成 Parallel Scan？ single kernel**
- **能否只读 O(n) 写 O(n)? single pass?**

回顾 Serial Scan 从读写量上来说是最优的，但是它是串行的。改成并行?



Single-Pass Chained-Scan

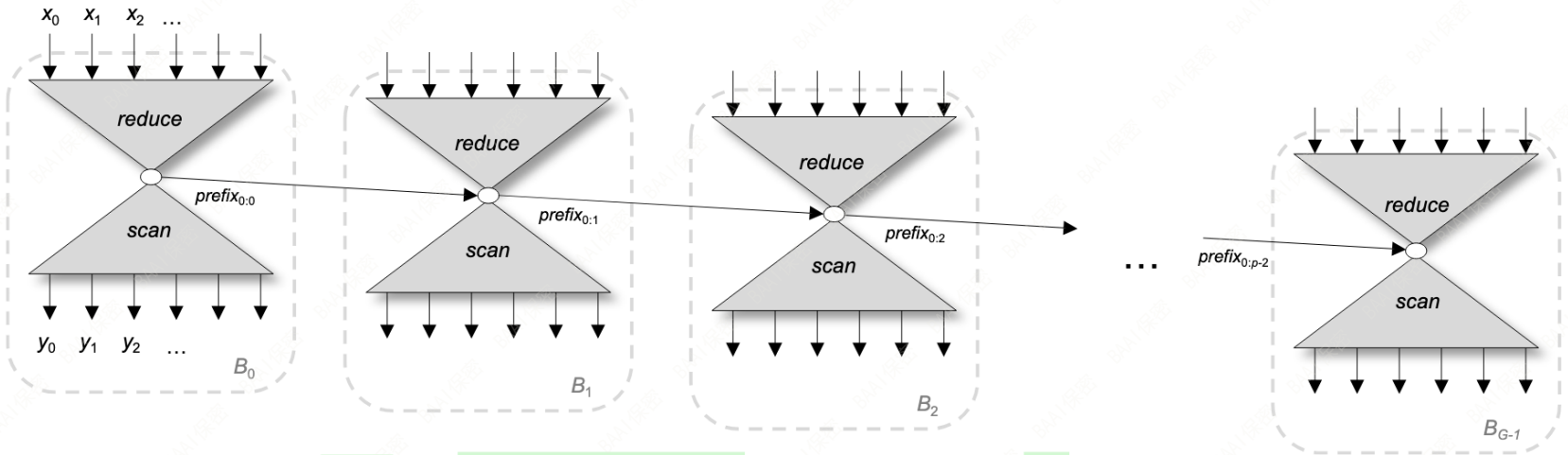
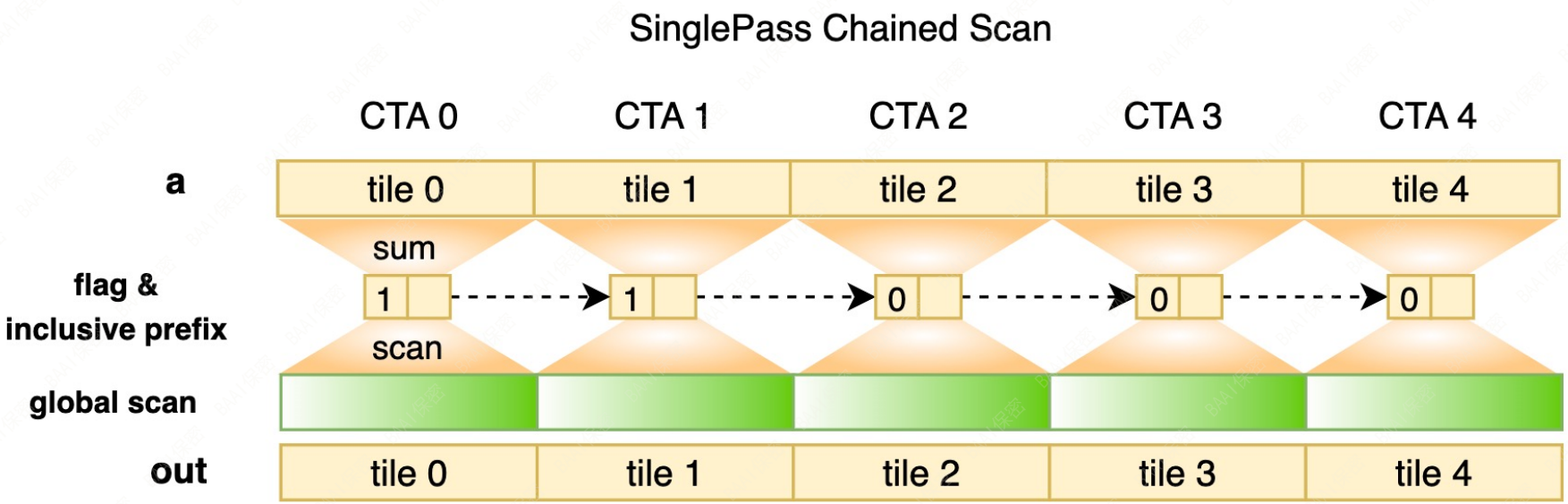


Fig. 4. Single-pass *chained-scan* prefix scan among G thread blocks (~$2n$ global data movement)

- 辅助存储：
  - prefix：每个 Block 一个标量，存储包含本 root scan 的结果。
  - flag：每个 Block 一个标量，表示 prefix 已经计算完成
- 每个 Block 计算：
  - 加载 input 数组中的一块，block_sum.
  - 阻塞 Block 的执行，直到前一个块的 prefix 计算完成。
  - prefix[i] = prefix[i-1] + block_sum
  - 以 prefix[i-1] 为 init 计算 block scan 并写入 global memory

Single-Pass Chained-Scan

- per-block 的 inclusive prefix sum 和 flag.阻塞 Block 的执行，直到前一个块的 prefix 计算完成。
- 一个块的 prefix 计算完成的标志是 flag 变成 1. 需要保证 inclusive prefix 写入 然后在写入 flag. （需要保证 compiler 不会 reorder 这两个 memory op. 而且需要保证其他 block 也看到的是一样的顺序，需要 memory barrier)
- 但在 Triton 中无法单独添加 memory barrier, 需要配合 atomic op 一起使用。



Single-Pass Chained-Scan


```
@triton.jit
def stream_scan_kernel(in_ptr, out_ptr, flag_ptr, prefix_ptr, N, TILE_N: tl.constexpr):
    # compute block sum

    # wait unitl the i-1 block complets, load the inclsive_prefix for block i-1
    if pid > 0:
        while tl.atomic_add(flag_ptr + pid - 1, 0, sem="acquire") != 1:
            pass
        prefix = tl.load(prefix_ptr + pid - 1)
    else:
        prefix = tl.zeros_like(local_sum)

    # update inclisive prefix
    updated_prefix = prefix + local_sum
    tl.store(prefix_ptr + pid, updated_prefix)
    tl.atomic_xchg(flag_ptr + pid, 1, sem="release")

    # compute the block scan with previous prefix as the init value
    global_cumsum = prefix + tl.cumsum(x, 0)
    tl.store(out_ptr + n_offsets, global_cumsum, mask=mask)
```

Single-Pass Decoupled-Lookback Scan

- Single-Pass Chained Scan 中 inclusive prefix 的更新是串行的，受限于 Block 间的信息传递速度。
- 可以增大 Block 处理的 tile size 缩短传播路径, 但这个算法不能在块内使用 Serial 的方法计算多个 tile(否则就会带来额外的一次数据加载，形成 3O(n) 的读写。
- 如何加速 Block 间信息的传递？Decoupled-LookBack. 通过增加冗余计算来将串行的计算解偶。

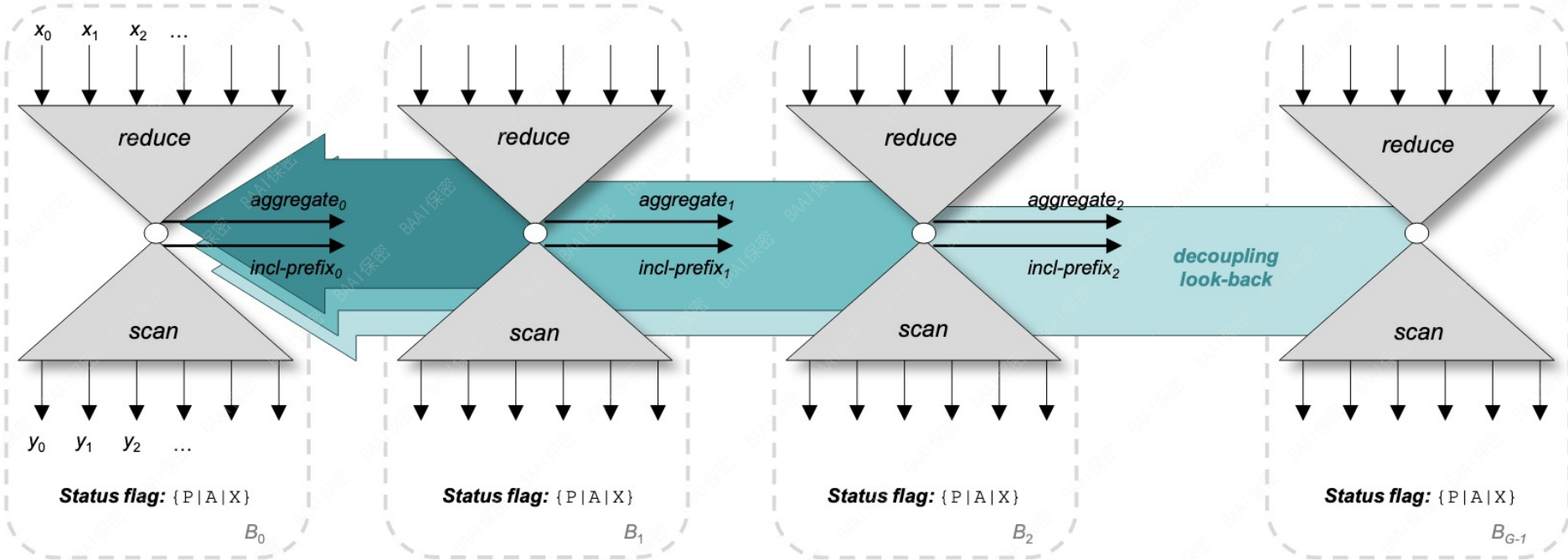


Fig. 5. Single-pass *adaptive look-back* prefix scan among G thread blocks (~$2n$ global data movement)

Single-Pass Decoupled-Lookback Scan

额外状态，每个 Block 维护一个

flag: 三种状态。

- X：block sum 未算完。
- A：block sum 算好了。Aggregate 可用。
- P：Inclusive prefix sum 算好了。Inclusive Prefix 可用。

Aggregate：本 Block 处理的 block sum.

Inclusive Prefix: 各个块的 block sum 的 Scan 结果。(inclusive prefix sum)

PartitionID	0	1	2	3	4	5	6	7
Flag	P	A	A	P	A	P	X	A
Aggregate	2	2	2	2	2	2	nil	2
InclusivePrefix	2	4	nil	8	nil	12	nil	nil

每个 Block 计算：

- 加载 input 数组中的一块，block_sum. 写入 Aggregate, 将 Flag 置为 X.
- exclusive_prefix =0
- 逐个位置回看（Lookback）：
  - 如果回看位置 flag 为 X, busy wait;
  - 如果回看位置 flag 为 A, inclusive_prefix += Aggregate, 继续回看前一个位置
  - 如果回看位置 flag 为 P，inclusive_prefix += InclusivePredix, 结束回看。
- 写入 InclusivePrefix = exclusive + block_sum，并置状态为 P.
- exclusive_prefix 为 init 计算 block scan 并写入 global memory

Single-Pass Decoupled-Lookback Scan


```
@triton.jit
def single_pass_scan_decoupled_lookback_kernel(in_ptr, out_ptr,
    flag_ptr, inclusive_prefix_ptr, aggregate_ptr, N, TILE_N: tl.constexpr):
    # compute store local_sum to aggregate
    local_sum = tl.sum(x, 0)
    tl.store(aggregate_ptr + pid_n, local_sum)
    tl.atomic_xchg(flag_ptr + pid_n, 1, sem="release")

    exclusive_prefix = 0
    # decoupled-lookback
    i = pid_n - 1
    while i >= 0:
        flag = tl.atomic_add(flag_ptr + i, 0, sem="acquire")
        while flag == 0:
            flag = tl.atomic_add(flag_ptr + i, 0, sem="acquire")
        if flag == 1:
            this_aggregate = tl.load(aggregate_ptr + i)
            exclusive_prefix += this_aggregate
            i -= 1
        else: # flag == 2
            inclusive_prefix = tl.load(inclusive_prefix_ptr + i)
            exclusive_prefix += inclusive_prefix
            i = -1
    tl.store(inclusive_prefix_ptr + pid_n, exclusive_prefix + local_sum)
    tl.atomic_xchg(flag_ptr + pid_n, 2, sem="release")

    local_cumsum = tl.cumsum(x, 0)
    tl.store(out_ptr + n_offsets, exclusive_prefix + local_cumsum, mask=mask)
```

也可以 memory barrier: 把 flag 和 状态打包成一个 Machine word, 可以确保 flag 和状态直接保持一致。

Single-Pass Decoupled-Lookback Scan

flag	value
00	0
01	block_sum
10	inclusive_prefix

```
@triton.jit
def single_pass_scan_decoupled_lookback_no_barrier_kernel(
    in_ptr, out_ptr, state_ptr, N, TILE_N: tl.constexpr):
    flag_a = tl.full((), 1, dtype=tl.uint64) << 32
    flag_p = tl.full((), 2, dtype=tl.uint64) << 32
    # compute block sum & store block_sum to aggregate
    local_sum = tl.sum(x, 0)
    tl.store(state_ptr + pid, flag_a | local_sum.to(tl.uint32, bitcast=True), cache_modifier=".cg")

    exclusive_prefix = tl.zeros_like(local_sum)
    # decoupled-lookback
    i = pid - 1
    while i >= 0:
        state = tl.load(state_ptr + i, volatile=True)
        while state == 0:
            state = tl.load(state_ptr + i, volatile=True)
        value = (state & 0xFFFFFFFF).to(tl.uint32).to(x.type.element_ty, bitcast=True)
        exclusive_prefix += value
        if state & flag_a:
            i -= 1
        else:
            i = -1
    inclusive_prefix = exclusive_prefix + local_sum
    tl.store(state_ptr + pid, flag_p | inclusive_prefix.to(tl.uint32, bitcast=True), cache_modifier=".cg")

    local_cumsum = tl.cumsum(x, 0)
    tl.store(out_ptr + n_offsets, exclusive_prefix + local_cumsum, mask=mask)
```

可以使用多个 thread 并行 lookback
但在 Triton 无法实现这个

Benchmark

- size 为 (16* 1024 * 1024) 的 int32 vector

Implementation	Bandwidth(GB/s)
[torch]	**829.57**
[reduce-then-scan]	546.133
[scan_then_propagate]	404.543
[chained_scan_global_state]	264.792
[stream_scan]	146.859
[stream_scan_no_barrier]	299.251
[single_pass_scan_decoupled_lookback]	789.59
[single_pass_scan_decoupled_lookback_no_barrier]	**799.22**

- size 为 (1024 * 1024 * 1024) 的 fp32 vector

Implementation	Bandwidth(GB/s)
[torch]	**858.477**
[reduce-then-scan]	581.997
[scan_then_propagate]	425.073
[chained_scan_global_state]	269.055
[stream_scan]	173.315
[stream_scan_no_barrier]	333.159
[single_pass_scan_decoupled_lookback]	843.505
[single_pass_scan_decoupled_lookback_no_barrier]	**846.821**

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

Reduce, Scan 和 Sort 算子的 Triton 实现

01 Recap: Triton Lang

02 Reduce

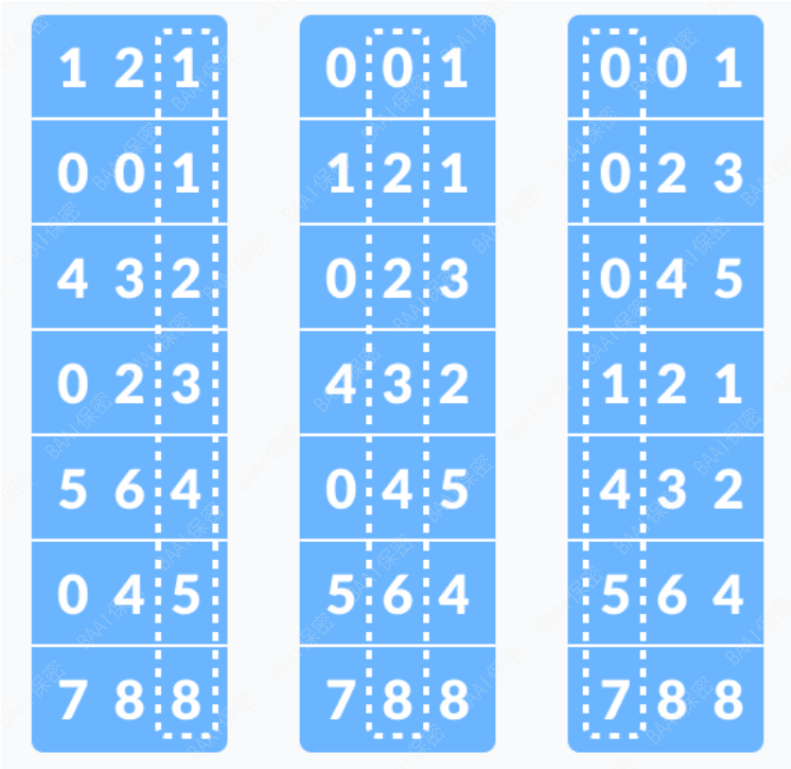
03 Scan

04 Sort

05 Message Passing between Blocks

Radix Sort

- 本次分享讲述 RadixSort 以及 Nvidia 的 OneSweep Radix Sort 在 Triton 语言中的实现。
- 输入数据 [121, 432, 564, 23, 1, 45, 788]，下图为 LSD (least-significant-digit ） Radix Sort方法每排完一位之后的结果。 https://www.programiz.com/dsa/radix-sort



Radix Sort

对于无符号整数，每次排一个数位(digit), 取所有数值的这一位作为 key, 使用 counting sort 进行排序。

正确性：

- digit 本身有序，counting sort 可以保证一位的有序性。
- 从低位往高位迭代过程中，如果高位的数值相同，低位已经排好的顺序可以保持。
- 位值制保证了高位的权重高于低位的权重。

有限性：数字的位数有限。

以某个数位提取数字，作为 key

Counting Sort:

- 首先计算 Histogram, 亦即每个 key 的出现次数。
- 计算出现在其前面的相同 key 的元素个数。
- 则在有序序列中，它的位置是所有 key 比它小的元素个数 + 其前面的相同 key 的元素个数。

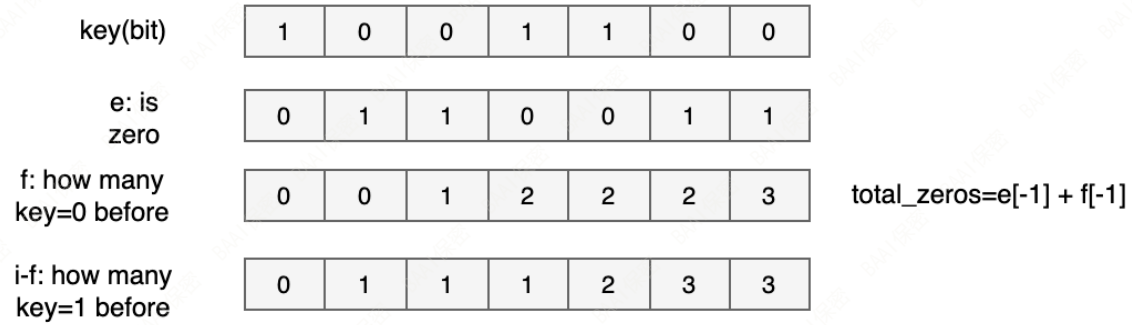
Radix Sort 中自有 Prefix Sum：

- 所有 key 比它小的元素个数。 scan on histogram
- 其前面的相同 key 的元素个数。scan on has_same_key

Radix Sort 1bit

由于二进制表示的整数只有 0 1 两个 digit, radix sort 可以更简单。下面以 numpy 代码示例。

```
def sweep(arr, i):
    N = arr.size
    index = np.arange(N, dtype=np.uint64)
    key = arr

    b = ((key >> i) & 1)
    e = 1 - b
    f = np.cumulative_sum(e, include_initial=True)[:-1]
    total_zeros = e[-1] + f[-1]
    p = np.where(b, index - f + total_zeros, f)
    arr[p] = arr

def radix_sort(arr):
    arr_copy = np.copy(arr)
    num_bits = arr.itemsize * 8
    for i in range(num_bits):
        sweep(arr_copy, i)
    return arr_copy
```



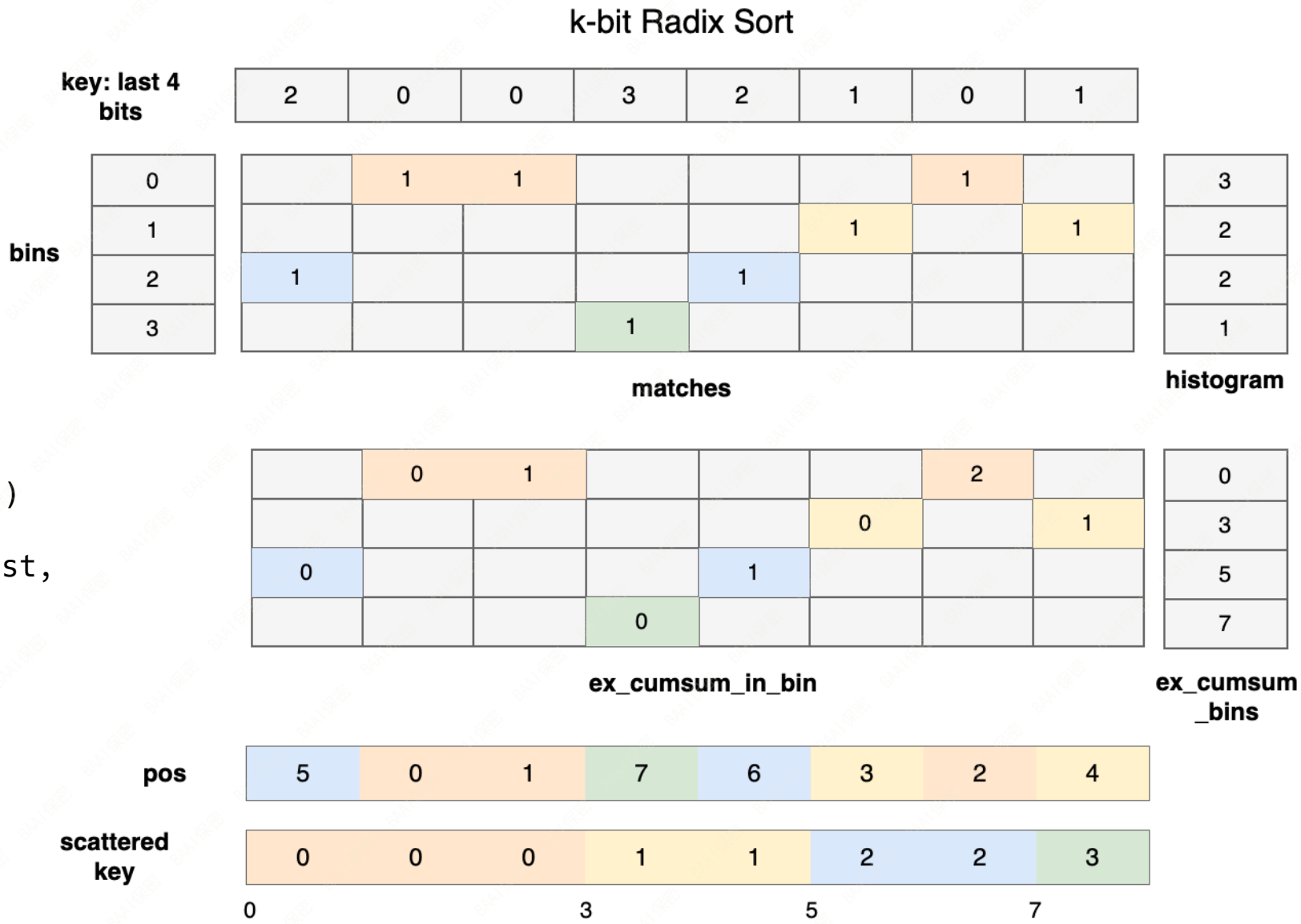
因为 radix sort 的每次 sweep 扫描 k bit, 需要排序的数字有 n bit, 那么需要 cdiv(n, k) 次扫描。每次扫描的读写量都是 2O(n). 所以可以一次排多位。

Radix Sort k-bit


```
def sweep(arr, i, k_bits):
    N = arr.size
    bfe_mask = 2 ** k_bits - 1
    key = (arr >> i) & bfe_mask
    bins = np.arange(0, 2 ** k_bits, dtype=np.uint64) #(r, )
    matches = bins[:, None] == key  #(r, N)
    ex_cumsum_in_bin = np.cumulative_sum(matches, axis=1, include_initial=True,
dtype=np.uint64) #(r, N+1)
    hist = ex_cumsum_in_bin[:, -1] # (r, ), actually a sum of matches along axis 1
    # ex_cumsum_in_bin = ex_cumsum_in_bin[:, :-1] # (r, N) this can be skipped
    ex_cumsum_bins = np.cumulative_sum(hist, axis=0, include_initial=True)
    # scatter
    index = np.arange(N, dtype=np.uint64)
    pos = ex_cumsum_bins[key] + ex_cumsum_in_bin[key, index]
    arr[pos] = arr

def radix_sort(arr, k_bits=4):
    arr_copy = np.copy(arr)
    num_bits = arr.itemsize * 8
    for i in range(0, num_bits, k_bits):
        sweep(arr_copy, i, k_bits)
    return arr_copy
```

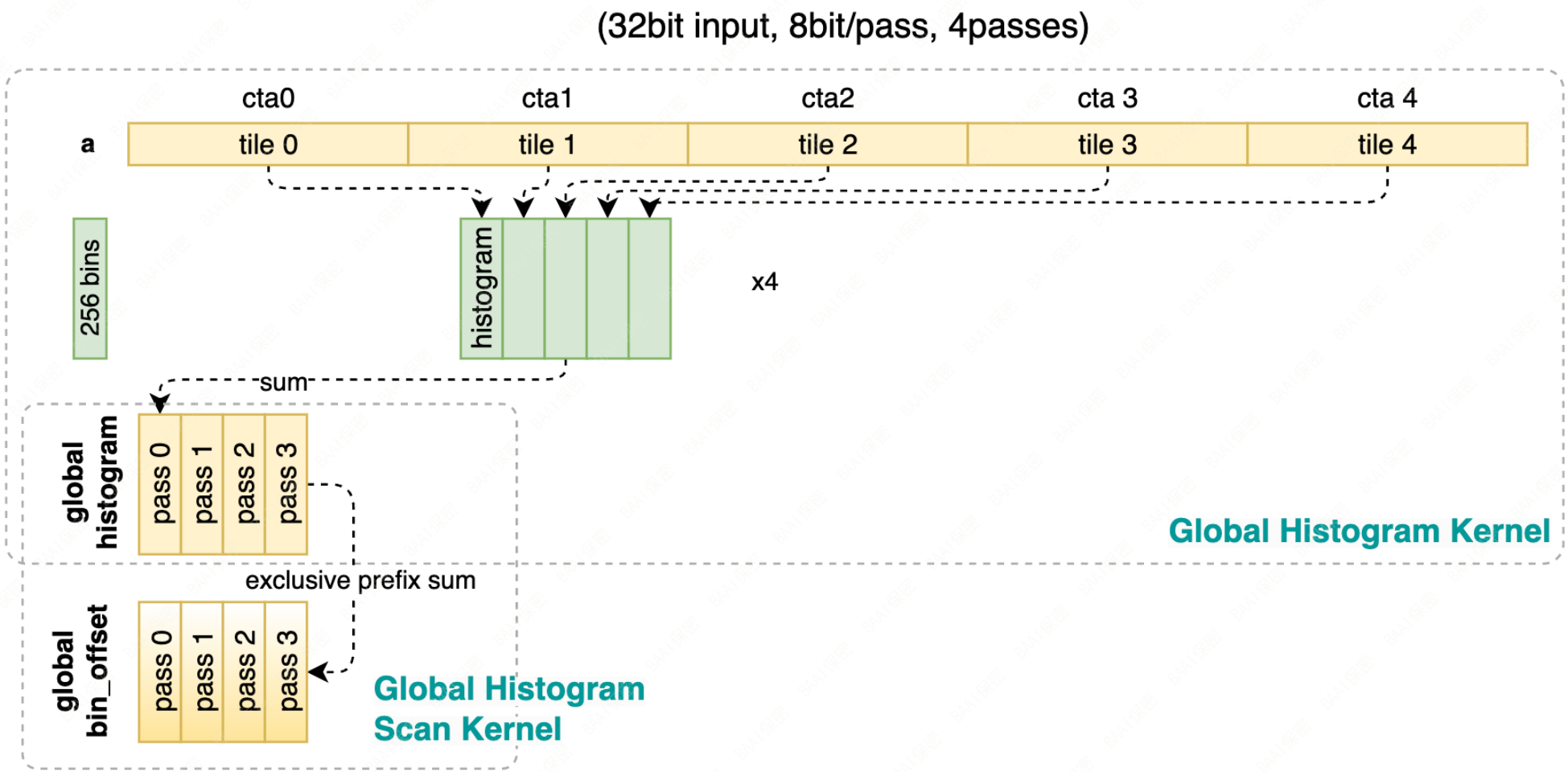
Radix Sort k-bit


```
key = (arr >> i) & bfe_mask
bins = np.arange(0, 2 ** k_bits,
dtype=np.uint64)
matches = bins[:, None] == key

ex_cumsum_in_bin =
np.cumulative_sum(matches, axis=1,
include_initial=True, dtype=np.uint64)
hist = ex_cumsum_in_bin[:, -1]
ex_cumsum_bins = np.cumulative_sum(hist,
axis=0, include_initial=True)

index = np.arange(N, dtype=np.uint64)
pos = ex_cumsum_bins[key] +
ex_cumsum_in_bin[key, index]
arr[pos] = arr
```



OneSweep Radix Sort

提前计算：对于 n bit 的输入，每个 pass 排序 k-bit, 需要 n/p 个 pass. 每个 pass 都需要一个 global_histogram on bins, 以及一个对 histogram 的 exclusive prefix sum. 但这个结果和数组的顺序无关，所以可以提前计算完成，用于各个 pass.



OneSweep Radix Sort

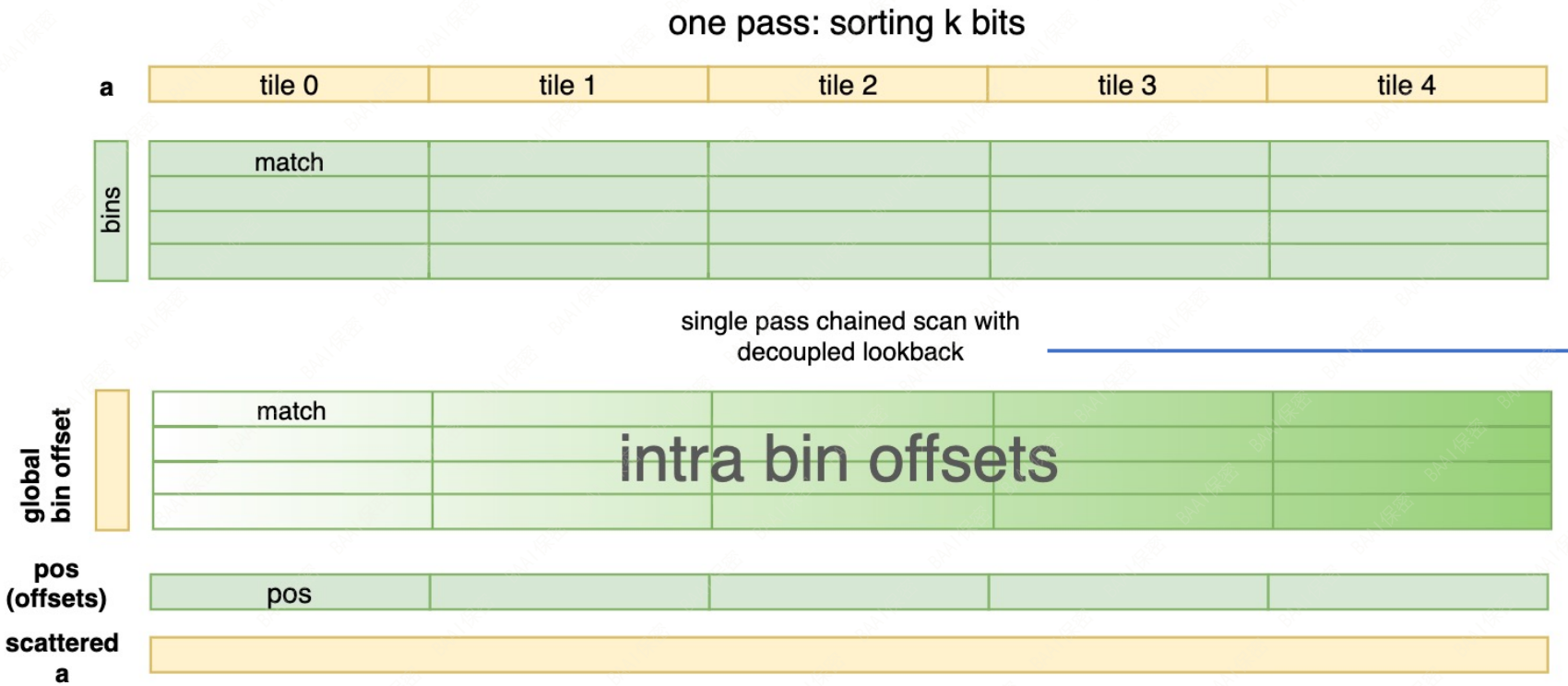
每个 pass 的过程：

- 每个 CTA 加载一块数据，（如果必要先转换为 uint），提取 kbit 上的值作为 key;
- 计算和 bins 的 match 值（0~1 mask);
- 使用 single-pass chained scan with decoupled lookback 算法计算整个序列中有多少个和自身同 key 的元素在它前面。(同一个 bin 内的 exclusive prefix sum)。注意：是同时计算 num_bins 个 prefix sum.
- 加载 global histogram 的 exclusive_prefix_sum,
- 一个元素的 scatter 位置是由两部分构成，两部分相加即是 scatter 到的位置：
  - global bin offset: 全局 key 小于它的元素个数
  - intra bin offset：同一个 bin 内的 exclusive prefix sum。
- 把输入 scatter 到上述的位置。
- （如果有关联数组，也一并加载和 scatter)

OneSweep Radix Sort

在 Triton 中有一些算法无法高效地实现，比如在 OneSweep Radix Sort 中，每一个 pass 都需要同时计算 num_bins 个 single pass chained scan with decoupled lookback. 其中的 reduce 和 scan 的部分都是可以独立计算的，只有 decoupled lookback 需要进行 Block 间的通信。

CUDA 程序中可以让 8 个 warp 的每个 thread 单独负责一个 digit 的 decoupled lookback, 256 个 thread, 刚好对应 8bit 的情况，共有 256 个 digit. 但 Triton 程序难以实现这样的精细控制。对于 CUDA 程序来说，可以只 block 一个 thread, 而 Triton 就必须 block 住整个 CTA, 只能串行计算 256 个 scan.由于上述提及的限制，目前在 Triton 中的实现性能较差。



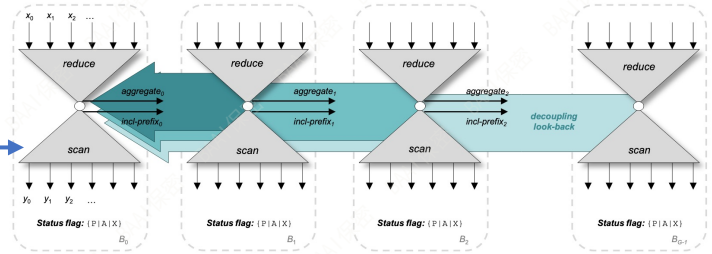


Fig. 5. Single-pass *adaptive look-back* prefix scan among G thread blocks (~$2n$ global data movement)

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

Reduce, Scan 和 Sort 算子的 Triton 实现

01 Recap: Triton Lang

02 Reduce

03 Scan

04 Sort

05 Message Passing between Blocks

Memory Barrier

GPU 的 memory operations:

- **Asynchronous**: 写并不会立即到达 global/shared
- **Reordered**: 编译器和硬件都有可能重排 memory operation。
- **Cached and staged**: 写都有可能被 cache, 因此可能对其他 thread 不可见。

因此需要使用 memory barrier 来被保证 order 和 visibility. 常见的一个 Pattern 是利用 Memory Barrier 进行 Block 间通信。由于 Triton 中无法单独使用 memory barrier, 需要配合 atomic operation 一起使用。

atomic operation 有两个参数：

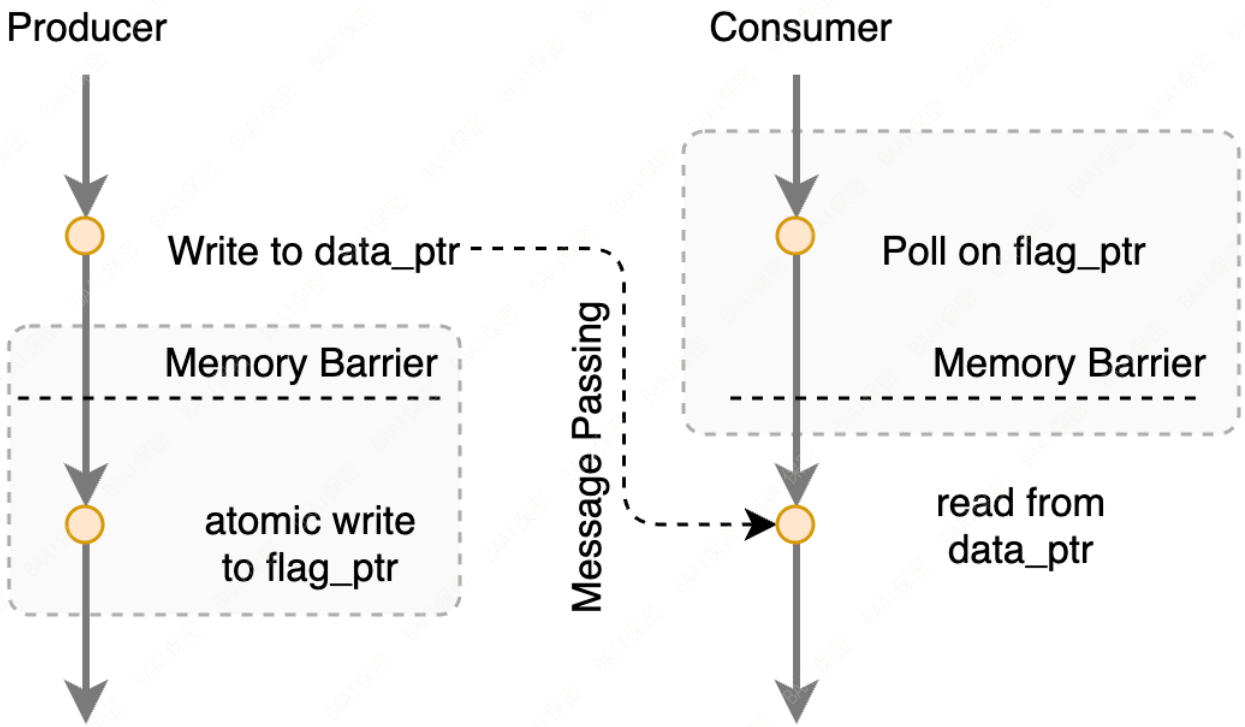
- **scope**: block, gpu, system. 如果为了跨 Block 通信，就用 gpu, 亦即 device.
- **semantic**: 和 memory barrier 有关的选项。
  - acquire: 在 atomic op 后加 memory barrier;
  - release: 在 atomic op 签加 memory barrier;
  - acq_rel: 前后都加
  - relaxed: 不加 memory barrier, 单纯保证原子性。

Memory Barrier

常用 pattern: 使用 per-block 的 flag 和用于通信的 data. data 和 flag 会被多个 CTA 读但是只会被一个 CTA 写。

需要确保 data 写完再写 flag. 以保证当其他 CTA 认为 flag 的值满足条件时，data 也是已经更新了的，而不会读到旧的数据。也需要确保读 data 的 CTA 也必须在读到 flag 满足条件之后再读 data.

```
# write data for this CTA
tl.store(data_ptr + pid, data)
tl.atomic_xchg(flag + pid, 1, sem="release")

# poll on other CTA's flag
while tl.atomic_add(
    flag_ptr + other_pid, 0, sem="acquire") == 0:
    pass
# load data writen by other CTAs
other_data = tl.load(data_ptr, other_pid)
```



CAUTION：注意防止死锁。

Atomic operations

Atomic operation 用于保证 memory operation 的原子性。

常用 pattern：使用一个 grid 的 counter 来表示已经完成任务的 CTA 数。比如，

- parallel reduce 中，可以让最后一个执行完 block sum 的 CTA 去执行第二阶段对 block sum 的求和。
- parallel reduce 中，可以让某一个 CTA 等到所有 CTA 执行完 block sum，然后执行第二阶段对 block sum 的求和。

常见 pattern: 直接用 global memory 上的数据作为 accumularor. 当不需要保证 memory order 的时候，可以使用 **relaxed** 语义。如

- 通过 atomic add 累积结果的 parallel reduction.

Volatile

防止如下的编译器优化

- Caching in registers
- Compiler reordering of *volatile* accesses

常见 Pattern:

把 data 和 flag pack 进一个 Machine word 的 state, 使用 tl.store with ".cg" cache modifier. 然后使用 tl.load with volatile 去 poll 它。

在 SinglePass chained-scan with decoupled lookback， 以及 Onesweep RadixSort 中都有使用。

参考文献

- Single-pass Parallel Prefix Scan with Decoupled Look-back
- Onesweep: A Faster Least Significant Digit Radix Sort for GPUs
- Inter-Block GPU Communication via Fast Barrier Synchronization
- Stream-K: Work-centric Parallel Decomposition for Dense Matrix-Matrix Multiplication on the GPU
- StreamScan: Fast Scan Algorithms for GPUs without Global Barrier Synchronization
- CUDA C++ Programming Guide