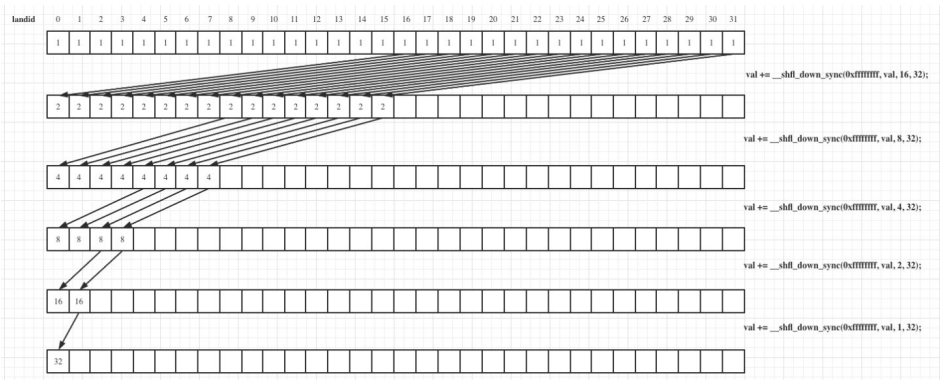
CUDA编程入门之Warp-Level Primitives



Introduction

NVIDIA GPUs 和 CUDA 编程模型采用一种称为 SIMT （单指令，多线程）的执行模型，其中一个重要的概念称为线程束(warp) 需要先了解下，才能深入理解本篇博客介绍的 warp-level 原语。

Warp 是 SM(Streaming Multiprocessor) 的基本执行单元，一个 warp 包含 32 个并行 thread，这 32 个 thread 遵循 SIMT 模式，也就是说所有 thread 会执行同一条指令，但每个 thread 会访问各自的数据。许多 CUDA 程序通过显示的利用 warp-level 编程尽可能频繁地一起执行相同的指令序列，从而最大限度地提高性能。在这个博客中，作者向我们展示了如何使用 CUDA 9 中引入的 warp-level primitives(原语)，使您的编程安全有效。

Warp-level Primitives

CUDA 9 引入了三类 warp-level 原语:

1. Synchronized data exchange: 在 warp 中的线程之间交换数据。

- Warp Vote Functions: __all_sync , __any_sync , __uni_sync, __ballot_sync
- Warp Reduce Functions: __shfl_sync , __shfl_up_sync , __shfl_down_sync , __shfl_xor_sync
- Warp Match Functions: __match_any_sync , __match_all_sync

2. Active mask query: 返回一个 32 位掩码，指示 warp 中的哪些线程在当前执行线程中处于活动状态。

    - __activemask

3. Thread synchronization: 同步 warp 中的线程，并提供内存隔离(memory fence)。

    - __syncwarp

Synchronized Data Exchange

__all_sync

```
int __all_sync(unsigned mask, int predicate);
```

表示如果 warp 中的任何线程传入了非零 predicate，则返回非零（即当且仅当所有线程的 predicate 非零时返回 1，否则返回 0）

参考例句：

```
__global__ void vote_all(int *a, int *b, int n)
 {
     int tid = threadIdx.x;
     if (tid > n)
         return;
     int temp = a[tid];
     b[tid] = __all_sync(0xffffffff, temp > 48);
 }
```

__any_sync

```
int __any_sync(unsigned mask, int predicate);
```

表示如果 warp 中的任何线程传递了非零 predicate，则返回非零（即当且仅当至少有一个线程的 predicate 非零时返回 1，否则返回 0）

参考例句：

```
__global__ void vote_any(int *a, int *b, int n)
{
    int tid = threadIdx.x;
    if (tid > n)
        return;
    int temp = a[tid];
    b[tid] = __any_sync(0xffffffff, temp > 48);
}
```

__uni_sync

```
int __uni_sync(unsigned mask, int predicate);
```

表示如果 warp 中的任何线程传入了非零 predicate或全零，则返回非零（即当且仅当被 mask 指定线程的 predicate 全部非零或全部为零时返回 1，否则返回 0）

参考例句：

```
__global__ void vote_union(int *a, int *b, int n)
{
    int tid = threadIdx.x;
    if (tid > n)
        return;
    int temp = a[tid];
    b[tid] = __uni_sync(0xffffffff, temp > 42 && temp < 53);
}
```

__ballot_sync

```
unsigned __ballot_sync(unsigned mask, int predicate);
```

返回一个 32 位无符号整数，代表了该线程束内变量 predicate 的非零值分布情况（即线程 predicate 为零的该函数返回值该位为 0，线程 predicate 非零的该函数返回值该位为 1 ）

参考例句：

```
unsigned mask = __ballot_sync(FULL_MASK, threadIdx.x < NUM_ELEMENTS);
if (threadIdx.x < NUM_ELEMENTS) {
    val = input[threadIdx.x];
    for (int offset = 16; offset > 0; offset /= 2)
        val += __shfl_down_sync(mask, val, offset);
    …
}
```

假设我们要计算数组 input[] 的所有元素的总和，其大小 NUM_ELEMENTS 小于线程块中的线程数，这个时候就可以考虑使用 __ballot_sync() 指定 predicate 为 thread.Idx.x < NUM_ELEMENTS 来计算 __shfl_down_sync() 需要的成员掩码 mask， 从而决定哪些线程将参与规约求和任务。__ballot_sync() 自身使用 FULL_MASK（32 个线程为 0xffffffff），因为我们假设所有线程都会执行它。

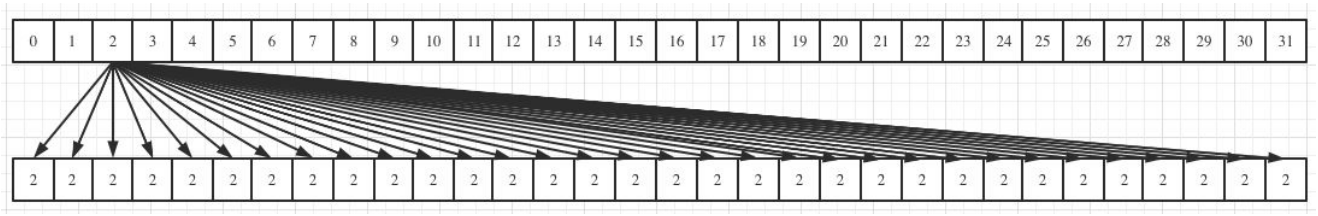
__shfl_sync

```
T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize);
```

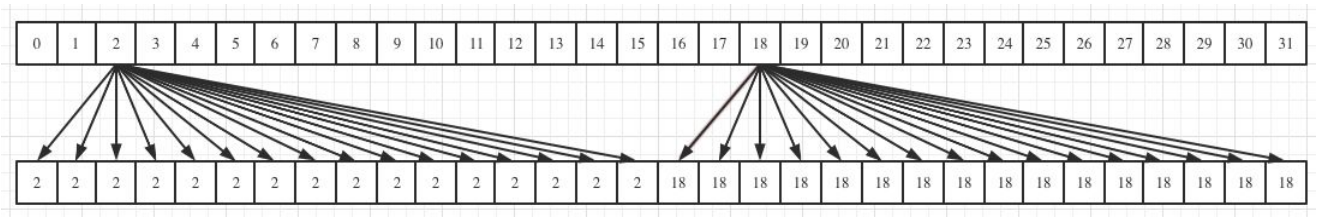
表示被 mask 指定的线程返回标号为 srcLane 的线程中的变量 var 的值，其余线程返回0 。类似 broadcast，mask 是参与的线程掩码，如0xffffffff，var 是待广播的值，srclane 是被广播的 laneid，warpsize 是参与 warp 大小；

参考例句：

__shfl_sync(0xffffffff,x,2)：表示 landid 为 2 的线程向 landid 为 0 ~ 31 的线程广播了其变量 x 的值；为了方便，这里 x 的值等于 landid，后面类似不在重复；



_shfl_sync(0xffffffff,x,2,16)：表示 laneid 为 2 的线程向标号为 0 ~ 15 的线程广播了其变量 x 的值；laneid 为 18 的线程向 laneid 为 16 ~ 31 的线程广播了其变量 x 的值。



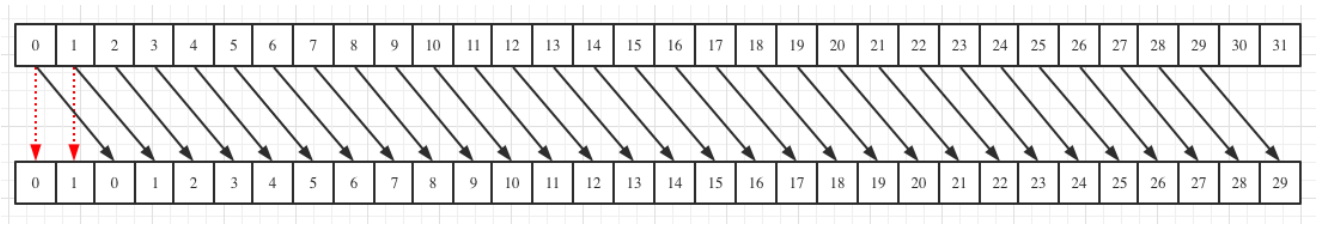
__shfl_up_sync

```
T __shfl_up_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
```

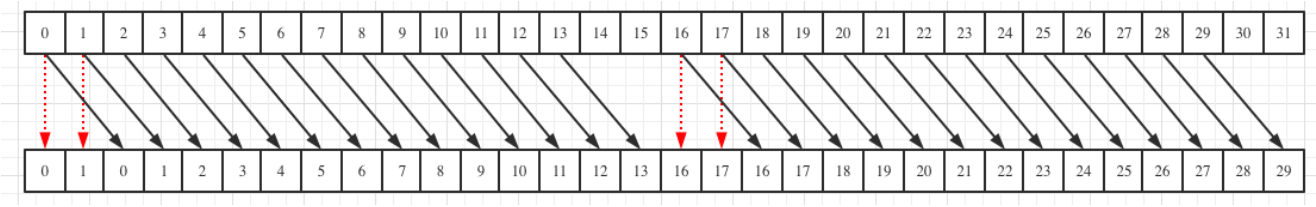
表示被 mask 指定的线程返回向前偏移为 delta 的线程中的变量 var 的值，其余线程返回0；

参考例句：

__shfl_up_sync(0xffffffff, x, 2)，表示 laneid 为 2 ~31 的线程分别获得 laneid 为 0 ~ 29 的线程中变量 x 的值；



__shfl_up_sync(0xffffffff, x, 2, 16)，表示 laneid 为 2 ~15 的线程分别获得 laneid 为 0 ~ 13 的线程中变量 x 的值；laneid 为 18 ~31 的线程分别获得 laneid 为 16 ~ 29 的线程中变量 x 的值；



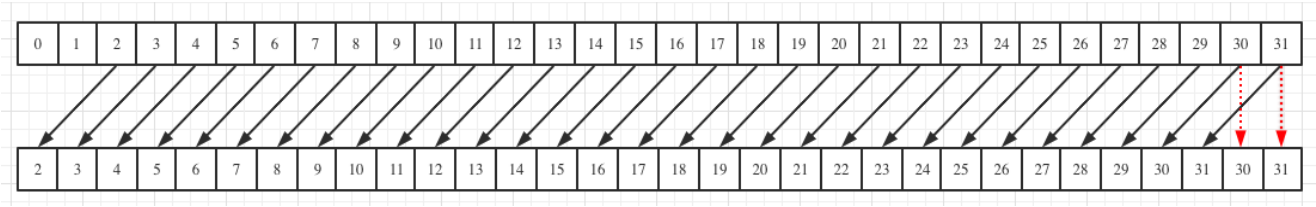
__shfl_down_sync

```
T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
```

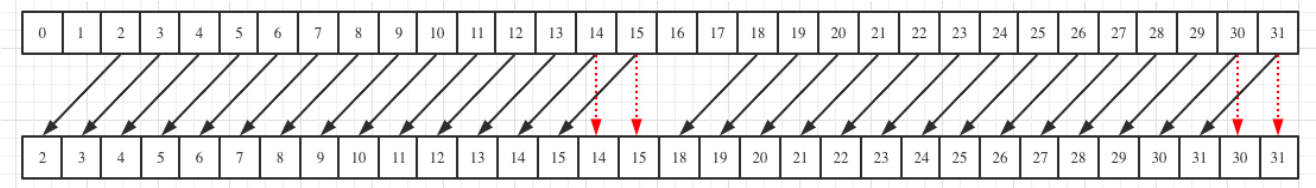
表示被 mask 指定的线程返回向后偏移为 delta 的线程中的变量 var 的值，其余线程返回0；

参考例句1：

__shfl_down_sync(0xffffffff, x, 2)，表示 laneid 为 0 ~29 的线程分别获得 laneid 为 2 ~ 31 的线程中变量 x 的值；



__shfl_down_sync(0xffffffff, x, 2, 16)，表示 laneid 为 0 ~13 的线程分别获得 laneid 为 2 ~ 15 的线程中变量 x 的值； laneid 为 16 ~29 的线程分别获得 laneid 为 18 ~ 31 的线程中变量 x 的值；



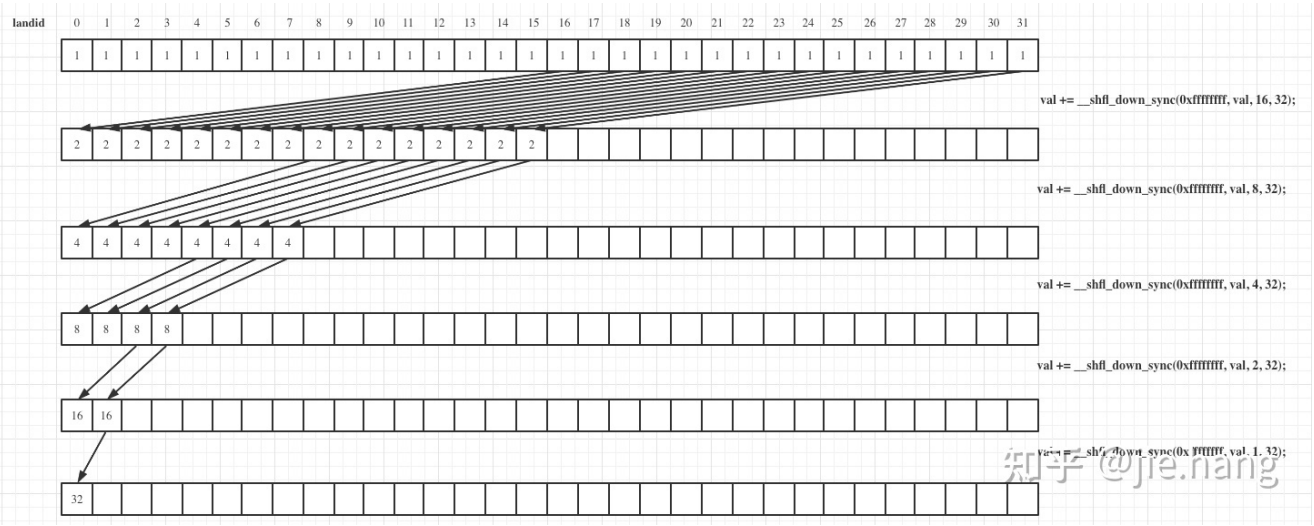
参考例句2：

__shfl_down_sync primitives 其实随处可见，在广泛应用的并行规约(parallel reductions)算法中，最常见的就是并行规约求和(BlockReduceSum)。理解 BlockReduceSum 之前先来看一下 WarpReduceSum 的实现过程，这里截取一段 pytorch 中实现的 WarpReduceSum 代码，共同学习一下；

```
// Sums `val` accross all threads in a warp.
//
// Assumptions:
//   - The size of each block should be a multiple of `warpSize`
template <typename T>
__inline__ __device__ T WarpReduceSum(T val) {
#pragma unroll
```

```
  for (int offset = (warpSize >> 1); offset > 0; offset >>= 1) {
    val += __shfl_down_sync(0xffffffff, val, offset, warpSize);
  }
  return val;
}
```

这段代码的实现逻辑可以借助下面这张图来辅助理解，假设初始 warp 内每个线程的 val 值为1，经过 5 轮循环之后**线程0获得最终正确的 reduce sum 结果val=32。**注意，这里只画了有助于理解最终规约结果的线程，实则所有线程都会改变值，只是他们在规约中并不会用到而已；



有了 WarpReduceSum 的基础，那么再截取一段 pytorch 中实现的 BlockReduceSum 代码，BlockReduce 主要借助 WarpReduce 来做，因此 blocksize 必须是 warp 的整数倍。整个流程如下：

1. 首先让所有线程执行 WarpReduceSum
2. 然后将每个线程束的 reduce 结果存储到 shared memory 中，注意这里是 lane_id=0 的线程去存储，因为前面提到了只有线程0上有正确的reduce结果
3. 从 shared memory 把数据读取出来，最后再用一个 warp 对其做 reduce，即可获得整个 block 的 reduce 结果

```
// Sums `val` accross all threads in a block.
//
// Assumptions:
//   - Thread blocks are an 1D set of threads (indexed with `threadIdx.x` only)
//   - The size of each block should be a multiple of `warpSize`
//   - `shared` should be a pointer to shared memory with size of, at least,
//     `sizeof(T) * number_of_warps`
template <typename T>
__inline__ __device__ T BlockReduceSum(T val, T* shared) {
  const int laneid = threadIdx.x % warpSize;
  const int warpid = threadIdx.x / warpSize;
  val = WarpReduceSum(val);
```

```
  __syncthreads();
  if (laneid == 0) {
    shared[warpid] = val;
  }
  __syncthreads();
  val = (threadIdx.x < blockDim.x / warpSize) ? shared[laneid] : T(0);
  if (warpid == 0) {
    val = WarpReduceSum(val);
  }
  return val;
}
```

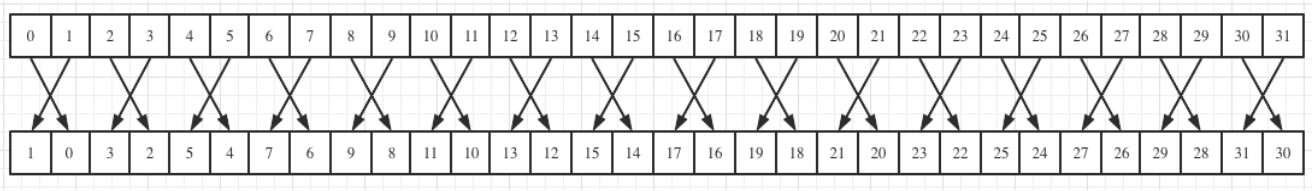
__shfl_xor_sync

```
T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize);
```

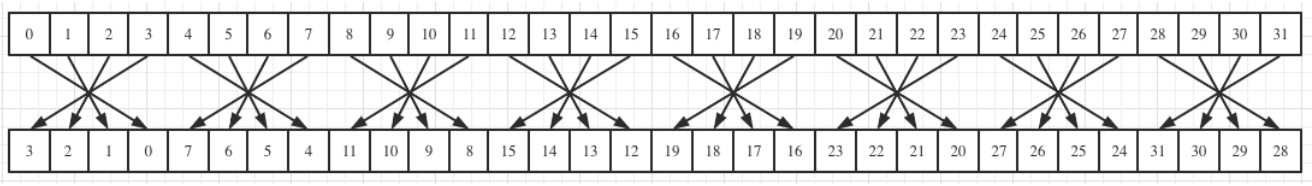
表示被 mask 指定线程的 laneid 与 laneMask 做按位异或计算，实现不同线程之间的 butterfly 数据交互；

参考例句：

_shfl_xor_sync(0xffffffff,x,1)，表示 laneid=0 的线程与laneid=1 的线程中变量 x 的值进行交互，laneid=2 的线程与laneid=3 的线程中变量 x 的值进行交互，后面其他线程依此类推；



_shfl_xor_sync (0xffffffff,x,3),



__match_any_sync

```
unsigned int __match_any_sync(unsigned mask, T value);
```

表示比较被 mask 指定的所有线程中的变量 value，返回具有相同值的线程编号构成的无符号整数。

__match_all_sync

```
unsigned int __match_all_sync(unsigned mask, T value, int *pred);
```

表示比较被 mask 指定的所有线程中的变量 value，当所有被指定的线程具有相同值的时候返回 mask 且 *pred 被置为 true，否则返回 0 且置 *pred 为 false。

总结一下：

参与调用以上介绍的每个原语的线程集都由第一个参数（即 32 位掩码）指定，且所有参与的线程必须同步，共同操作(collective operation)才能正常进行，因此，如果这些原语尚未同步，则这些原语首先需要先同步线程。

一个经常被问到的问题是“该如何指定掩码参数？”。您可以将掩码视为一个线程束(warp)中应参与共同操作(collective operation)的一组线程，这组线程是由程序逻辑决定的，通常可以通过程序流程中较早的一些分支条件来计算；比如前面介绍的 unsigned mask = __ballot_sync(FULL_MASK, threadIdx.x < NUM_ELEMENTS);

Active Mask Query

__activemask() 返回调用 warp 中所有当前激活线程的 32 位无符号整数掩码。换句话说就是调用线程中哪些线程也在执行相同的__activemask()。这对于作者后面介绍的 Opportunistic warp-level programming 技术以及调试和理解程序行为很有用。

```
unsigned __activemask();
```

然而，正确使用 __activemask() 还是比较重要的。下面这段代码就给出了一个不正确的用法。该代码尝试做一个求和规约，在分支内部使用__activemask()，这是不正确的，因为它会导致求出部分和而不是全部和。因为，CUDA 执行模型并不能保证所有接受分支的线程都将一起执行__activemask()。

```
//
// Incorrect use of __activemask()
//
if (threadIdx.x < NUM_ELEMENTS) {
    unsigned mask = __activemask();
    val = input[threadIdx.x];
    for (int offset = 16; offset > 0; offset /= 2)
        val += __shfl_down_sync(mask, val, offset);
    …
}
```

所以要在在进入分支之前使用 __ballo_sync() 计算出掩码，

```
unsigned mask = __ballot_sync(FULL_MASK, threadIdx.x < NUM_ELEMENTS);
if (threadIdx.x < NUM_ELEMENTS) {
    val = input[threadIdx.x];
    for (int offset = 16; offset > 0; offset /= 2)
        val += __shfl_down_sync(mask, val, offset);
    …
}
```

Warp Synchronization

当 warp 中的线程需要执行比 Data Exchange primitives 提供的更复杂的通信或集体操作时，可以使用 __syncwarp() 原语来同步 warp 中的线程。它类似于 __syncthreads() 原语（同步线程块中的所有线程），但粒度更细。

```
void __syncwarp(unsigned mask=FULL_MASK);
```

__syncwarp() 原语使执行线程等待，直到 mask 中指定的所有线程都执行了 __syncwarp()(使用相同的mask)，然后再继续执行。它还提供了一个 memory fence ，允许线程在调用原语之前和之后通过内存进行通信。

下面给了一个 shuffling 矩阵元素的示例。

```
float val = get_value(…);
__shared__ float smem[4][8];

//   0  1  2  3  4  5  6  7
//   8  9 10 11 12 13 14 15
//  16 17 18 19 20 21 22 23
//  24 25 26 27 28 29 30 31
int x1 = threadIdx.x % 8;
int y1 = threadIdx.x / 8;

//   0  4  8 12 16 20 24 28
//   1  5 10 13 17 21 25 29
//   2  6 11 14 18 22 26 30
//   3  7 12 15 19 23 27 31
int x2= threadIdx.x / 4;
int y2 = threadIdx.x % 4;

smem[y1][x1] = val;
__syncwarp();
val = smem[y2][x2];

use(val);
```

假设使用了一维线程块(即 threadIdx.y 始终为 0)，一个 warp 中的每个线程按行主索引(row-major indexing)方式负责处理 4 × 8 矩阵中的一个元素，也就是 laneid=0 的线程处理 smem[0][0]，laneid=1 的线程处理 smem[0][1]。这样，每个线程就完成了将其值存储到 4 × 8 大小的共享内存数组中的相应位置。然后使用 __syncwarp() 来确保每个线程准备从数组中的一个转置位置读取数据之前，所有线程都已经完成了存储。最后，warp 中的每一个线程再从smem中按列主索引(column-major indexing)方式读取数据写回val;

__syncwarp() 目的在于将共享内存读写分开，以避免争用情况(race condition)。下面这段代码就是错误的使用__syncwarp() 来实现基于共享内存实现的并行规约求和算法。在每两个 __syncwarp() 调用之间有一个共享内存读取，然后

是共享内存写入。 CUDA 编程模型不能保证所有的读操作都会在所有的写操作之前执行，因此存在竞争条件。

```
unsigned tid = threadIdx.x;

// Incorrect use of __syncwarp()
shmem[tid] += shmem[tid+16]; __syncwarp();
shmem[tid] += shmem[tid+8];  __syncwarp();
shmem[tid] += shmem[tid+4];  __syncwarp();
shmem[tid] += shmem[tid+2];  __syncwarp();
shmem[tid] += shmem[tid+1];  __syncwarp();
```

下面这段代码通过插入额外的 __syncwarp() 调用避免了竞争条件，CUDA 编译器可以在最终生成的代码中省略一些同步指令，这取决于硬件结构（例如，pre-Volta 架构）。

```
unsigned tid = threadIdx.x;
int v = 0;

v += shmem[tid+16]; __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+8];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+4];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+2];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+1];  __syncwarp();
shmem[tid] = v;
```

在最新的 Volta （和 future ） GPUs 上，也可以在线程发散(thread-divergent)分支中使用 __syncwarp() 来同步两个分支的线程，但是如果它们使用了warp-level primitives，线程可能会再次发散。

Opportunistic Warp-level Programming

在上一节 Synchronized Data Exchange 中介绍的每个原语中使用的掩码mask，通常是在程序流中的分支条件之前就得计算好的。但在许多情况下，程序需要沿着程序流传递掩码；例如，在只有一个函数参数的库函数内部使用 warp-level 原语时，因不能更改函数接口，所以可能就比较困难。

有些计算可以碰巧使用一起执行的任何线程。我们可以使用一种称为 opportunistic warp-level programming 的技术，如下例所示。（有关该算法的更多信息，请参见这个帖子 中的 warp aggregated atomics ；有关Cooperative Groups 如何使实现更简单的讨论，请参见 这个帖子 。这两篇后面也会考虑拿出来分享）

```
// increment the value at ptr by 1 and return the old value
__device__ int atomicAggInc(int *ptr) {
    int mask = __match_any_sync(__activemask(), (unsigned long long)ptr);
    int leader = __ffs(mask) - 1;    // select a leader
```

```
    int res;
    if(lane_id() == leader)                  // leader does the update
        res = atomicAdd(ptr, __popc(mask));
    res = __shfl_sync(mask, res, leader);    // get leader's old value
    return res + __popc(mask & ((1 << lane_id()) - 1)); //compute old value
}
```

atomicAggInc() 以原子方式将 ptr 指向的值递增 1 并返回旧值。使用 atomicAdd() 函数，可能会引起争论。为了减少争论，atomicAggInc 用 per-warp atomicAdd() 替换了 per-thread atomicAdd() 操作。第 3 行中的 __activemask() 在 warp 中查找将要执行原子操作的线程集。__match_any_sync() 返回具有相同值 ptr 的线程的位掩码，也就是将具有相同 ptr 值的线程划分成一组。每个组选择一个 leader 线程（第 4 行），负责为整个组执行 atomicAdd()（第 7 行）。然后通过调用 __shfl_sync 操作将从 atomicAdd() 返回的旧值 res 广播给每个线程（第8行）。第 9 行计算并返回当前线程从 atomicInc() 获得的旧值，如果当前线程调用的是该函数而不是 atomicAggInc。

Implicit Warp-Synchronous Programming is Unsafe

与 CUDA 9 原语相比，之前版本提供的原语不接受 mask 参数。例如，int __any(int predicate) 是 int __any_sync(unsigned mask, int predicate) 的旧版本。

如前所述， mask 参数决定了一个 warp 中必须参与原语的线程集。如果掩码指定的线程在执行过程中尚未同步，则新版本的原语将执行线程束内线程级(intra-warp thread-level)同步。

传统的 warp 级别原语不允许程序员指定所需的线程，也不执行同步。因此，参与 warp-level 操作的线程不是由 CUDA 程序明确表示的。这样一个程序的正确性取决于不明确的线程束同步行为(Implicit Warp-Synchronous)，这种行为可能随着硬件体系结构改变而变化，也有可能随着 CUDA 工具包版本的改变而变化（例如，由于编译器优化的变化），甚至有可能随着运行时执行状态的改变而改变。这种不明确的线程束同步是不安全的，有可能导致无法正常工作。

例如，在下面的代码中，假设 warp 中的所有 32 个线程一起执行第 2 行。第 4 行的 if 语句导致线程发散，奇数线程在第 5 行调用 foo()，偶数线程在第 8 行调用 bar() 。

```
// Assuming all 32 threads in a warp execute line 1 together.
assert(__ballot(1) == FULL_MASK);
int result;
if (thread_id % 2) {
    result = foo();
```

```
}
else {
    result = bar();
}
unsigned ballot_result = __ballot(result);
```

CUDA 编译器和硬件将尝试在第 10 行重新聚合线程，以获得更好的性能。但这一重新收敛是不保证的。因此，ballot_result 可能不包含来自所有 32 个线程的投票结果。

在 __ballot() 之前的第 10 行调用新的 __syncwarp() 原语，如下代码所示，也不能解决这个问题，这仍然属于 implicit warp-synchronous programming 。假设同一个 warp 中的线程一旦同步，将会一直保持同步，直到下一个线程发散分支为止。尽管这通常是真的，但在 CUDA 编程模型中并不能保证它。

```
__syncwarp();
unsigned ballot_result = __ballot(result);
```

正确的修复方法是使用 __ballot_sync()。

```
unsigned ballot_result = __ballot_sync(FULL_MASK, result);
```

一个常见的错误就是认为在旧版不带 sync 的 warp-level 原语之前和之后分别调用 __syncwarp()，以为在功能上就完全等同于调用带 sync 原语的版本。例如，

```
__syncwarp();
v = __shfl(0);
__syncwarp();
```

与直接调用 __shfl_sync 是等价的吗?

```
__shfl_sync(FULL_MASK, 0)
```

显然，答案是否定的，有两个原因。首先，如果在线程发散分支中使用上面这段代码，那么 __shfl(0) 就不会被所有线程一起执行。作者分享了一个案例，其中第 3 行和第 7 行的 __syncwarp() 将确保在执行第 4 行或第 8 行之前，warp 中的所有线程都会调用 foo() 。一旦线程离开 __syncwarp() ，奇数线程和偶数线程将再次发散。因此，第 4 行的 __shfl(0) 可能得到一个未定义的值，因为当第 4 行执行时，有可能 laneId=0 的线程还处于未激活的状态。 __shfl_sync(FULL_MASK, 0) 可以在线程发散的分支中使用，没有这个问题。

```
v = foo();
if (threadIdx.x % 2) {
    __syncwarp();
    v = __shfl(0);        // L3 will get undefined result because lane 0
    __syncwarp();         // is not active when L3 is executed. L3 and L6
} else {                  // will execute divergently.
    __syncwarp();
    v = __shfl(0);
    __syncwarp();
}
```

第二，即使所有线程一起调用上面这段代码， CUDA 执行模型也不能保证线程在离开 __syncwarp() 后保持收敛，例如下面这段代码，并不能保证不明确的 lock-step 的执行。请记住，线程收敛只在明确同步的 warp-level 原语中得到保证，就是前面介绍的带 sync 的 warp-level 原语。

```
assert(__activemask() == FULL_MASK); // assume this is true
__syncwarp();
assert(__activemask() == FULL_MASK); // this may fail
```

因为使用它们可能会导致不安全的程序，所以从 CUDA 9.0 开始就不推荐使用旧版的不带 sync 的 warp-level 原语。

Update Legacy Warp-Level Programming

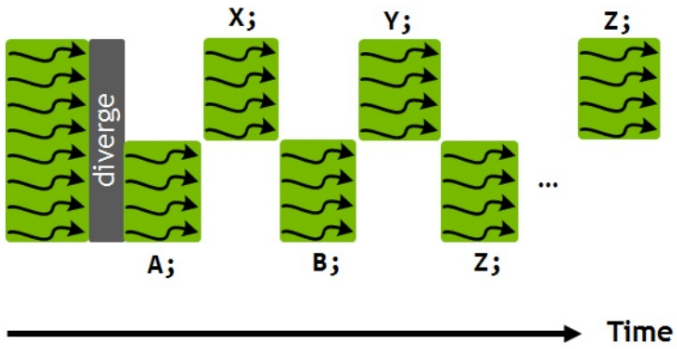
如果您的程序使用旧的 warp-level 原语或任何形式的隐式 warp 同步(implicit warp-synchronous)编程（例如在没有同步的 warp 线程之间通信），作者建议更新代码使用带 sync 版本的原语。根据需要可能还要重新构造代码以使用 Cooperative Groups，这提供了更高级别的抽象以及诸如多块同步等新功能。

使用 warp-level primitives 最棘手的部分是找出要使用的成员掩码。作者希望通过以上几节能给我们提供一些建议：

1. 不要只使用 FULL_MASK（即对于 32 个线程使用 0xffffffff）作为 mask 值。如果不是所有的线程都能根据程序逻辑到达原语，那么使用 FULL_MASK 可能会导致程序 hang。
2. 不要只使用 __activemask() 作为掩码值。__activemask() 会告诉您当函数被调用时，哪些线程会收敛，这可能与您希望在集合操作中的情况不同。
3. 分析程序逻辑并理解成员资格要求。根据程序逻辑提前计算掩码。
4. 如果您的程序存在前面介绍的 opportunistic warp-synchronous programming，请使用" detective" 函数，如__activemask() 和 __match_all_sync() 来找到正确的掩码。
5. 使用 __syncwarp() 来分离与 intra-warp 相关的操作。不要假设执行锁步(lock-step)。

最后一个诀窍。如果您现有的 CUDA 程序在 Volta architecture GPUs 上给出了不同的结果，并且您怀疑差异是由 Volta 新的独立线程调度 引起的，它可能会改变 warp 同步行为，您可能需要使用 nvcc 选项 -arch=compute_60 -code=sm_70 重新编译程序。这样的编译程序将会选择使用 Pascal 的线程调度。当有选择地使用时，它可以帮助更快地确定罪魁祸首模块，允许您更新代码以避免不明确的线程束同步编程(implicit warp-synchronous)。

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



Volta independent thread scheduling enables interleaved execution of statements from divergent branches. This enables execution of fine-grain parallel algorithms where threads within a warp may synchronize and communicate.