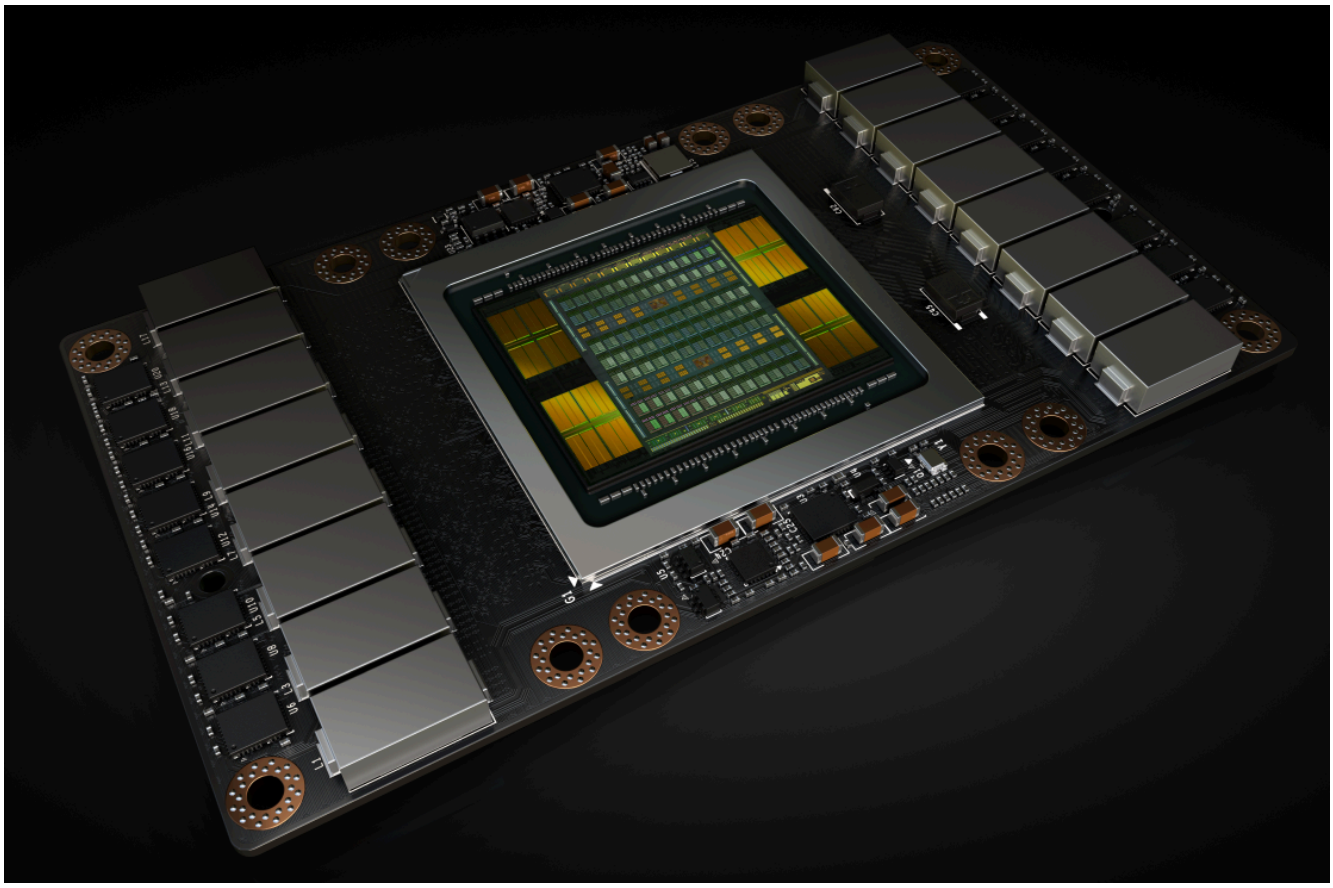


Figure 1: The Tesla V100 Accelerator with Volta GV100 GPU. SXM2 Form Factor.

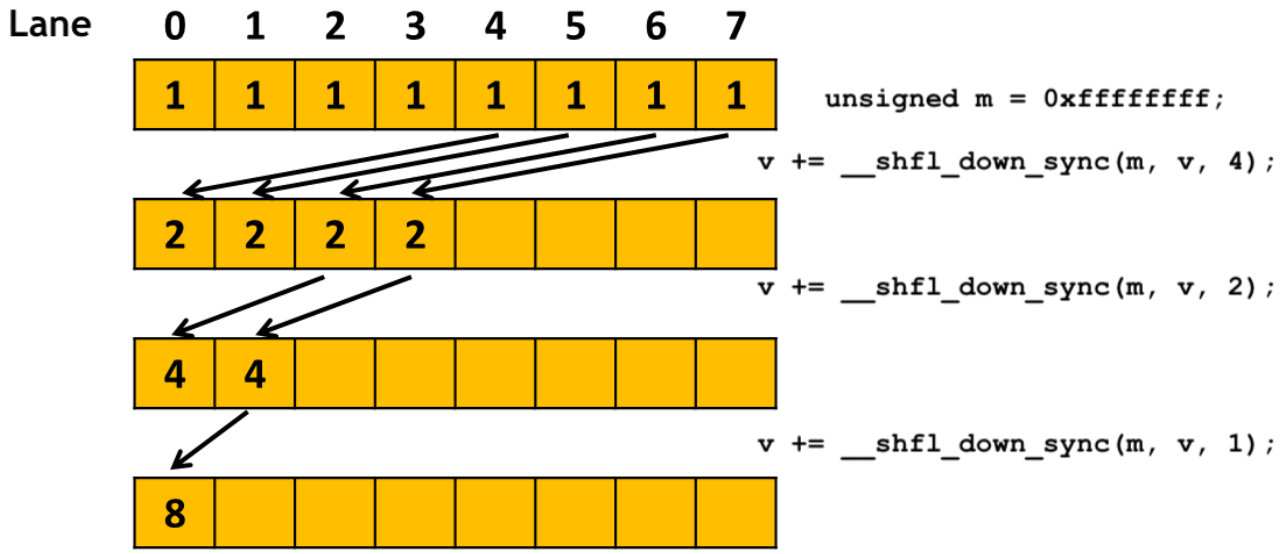
NVIDIA GPUs execute groups of threads known as *warps* in SIMT (Single Instruction, Multiple Thread) fashion. Many CUDA programs achieve high performance by taking advantage of warp execution. In this blog we show how to use primitives introduced in CUDA 9 to make your warp-level programing safe and effective.

Warp-level Primitives

NVIDIA GPUs and the CUDA programming model employ an execution model called SIMT (Single Instruction, Multiple Thread). SIMT extends Flynn's Taxonomy of computer architectures, which describes four classes of architectures in terms of their numbers of instruction and data streams. One of Flynn's four classes, SIMD (Single Instruction, Multiple Data) is commonly used to describe architectures like GPUs. But there is a subtle but important difference between SIMD and SIMT. In a SIMD architecture, each instruction applies the same operation in parallel across many data elements. SIMD is typically implemented using processors with vector registers and execution units; a scalar thread issues vector instructions that execute in SIMD fashion. In a SIMT architecture, rather than a single thread issuing vector instructions applied to data vectors, multiple threads issue common instructions to arbitrary data.

The benefits of SIMT for programmability led NVIDIA's GPU architects to coin a new name for this architecture, rather than describing it as SIMD. NVIDIA GPUs execute warps of 32 parallel threads using SIMT, which enables each thread to access its own registers, to load and store from divergent addresses, and to follow divergent control flow paths. The CUDA compiler and the GPU work together to ensure the threads of a warp execute the same instruction sequences together as frequently as possible to maximize performance.

While the high performance obtained by warp execution happens behind the scene, many CUDA programs can achieve even higher performance by using explicit warp-level programming. Parallel programs often use collective communication operations, such as parallel reductions and scans. CUDA C++ supports such collective operations by providing warp-level primitives and Cooperative Groups collectives. The Cooperative Groups collectives (described in this previous post) are implemented on top of the warp primitives, on which this article focuses.



Part of a warp-level parallel reduction using shfl_down_sync().

Listing 1 shows an example of using warp-level primitives. It uses `__shfl_down_sync()` to perform a tree-reduction to compute the sum of the `val` variable held by each thread in a warp. At the end of the loop, `val` of the first thread in the warp contains the sum.

```
#define FULL_MASK 0xffffffff
for (int offset = 16; offset > 0; offset /= 2)
    val += __shfl_down_sync(FULL_MASK, val, offset);
```

A warp comprises 32 *lanes*, with each thread occupying one lane. For a thread at lane `X` in the warp, `__shfl_down_sync(FULL_MASK, val, offset)` gets the value of the `val` variable from the thread at lane `X+offset` of the same warp. The data exchange is performed between registers, and more efficient than going through shared memory, which requires a load, a store and an extra register to hold the address.

CUDA 9 introduced three categories of new or updated warp-level primitives.

1. Synchronized data exchange: exchange data between threads in warp.
    - `__all_sync`, `__any_sync`, `__uni_sync`, `__ballot_sync`
    - `__shfl_sync`, `__shfl_up_sync`, `__shfl_down_sync`, `__shfl_xor_sync`
    - `__match_any_sync`, `__match_all_sync`
2. Active mask query: returns a 32-bit mask indicating which threads in a warp are active with the current executing thread.
    - `__activemask`
3. Thread synchronization: synchronize threads in a warp and provide a memory fence.
    - `__syncwarp`

Please see the CUDA Programming Guide for detailed descriptions of these primitives.

Synchronized Data Exchange

Each of the "synchronized data exchange" primitives perform a collective operation among a set of threads in a warp. For example, Listing 2 shows three of these. Each thread that calls `__shfl_sync()` or `__shfl_down_sync()` receives data from a thread in the same warp, and each thread that calls `__ballot_sync()` receives a bit mask representing all the threads in the warp that pass a true value for the predicate argument.

```
int __shfl_sync(unsigned mask, int val, int src_line, int width=warpSize);
int __shfl_down_sync(unsigned mask, int var, unsigned detla,
                     int width=warpSize);
int __ballot_sync(unsigned mask, int predicate);
```

The set of threads that participates in invoking each primitive is specified using a 32-bit mask, which is the first argument of these primitives. All the participating threads must be synchronized for the collective operation to work correctly. Therefore, these primitives first synchronize the threads if they are not already synchronized.

A frequently asked question is "what should I use for the `mask` argument?". You can consider the mask to mean the set of threads in the warp that should participate in the collective operation. This set of threads is determined by the program logic, and can usually be computed by some branch condition earlier in the program flow. Take the reduction code in Listing 1 as an example. Assume we want to compute the sum of all the elements of an array `input[]`, whose size `NUM_ELEMENTS` is less than the number of threads in the thread block. We can use the method in Listing 3.

```
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);
    …
}
```

The code uses the condition `thread.idx.x < NUM_ELEMENTS` to determine whether or not a thread will participate in the reduction. `__ballot_sync()` is used to compute the membership mask for the `__shfl_down_sync()` operation. `__ballot_sync()` itself uses `FULL_MASK` (`0xffffffff` for 32 threads) because we assume all threads will execute it.

On Volta and later GPU architectures, the data exchange primitives can be used in thread-divergent branches: branches where some threads in the warp take a different path than the others. Listing 4 shows an example where all the threads in a warp get the value of `val` from the thread at lane 0. The even- and odd-numbered threads take different branches of an `if` statement.

```
if (threadIdx.x % 2) {
    val += __shfl_sync(FULL_MASK, val, 0);
…
}
else {
val += __shfl_sync(FULL_MASK, val, 0);
…
}
```

On the latest Volta (and future) GPUs, you can run library functions that use warp synchronous primitives without worrying whether the function is called in a thread-divergent branch.

Active Mask Query

`__activemask()` returns a 32-bit `unsigned int` mask of all currently active threads in the calling warp. In other words, it shows the calling thread which threads in its warp are also executing the same `__activemask()`. This is useful for the :opportunistic warp-level programming" technique we explain later, as well as for debugging and understanding program behavior.

However, it's important to use `__activemask()` correctly. Listing 5 illustrates an incorrect use. The code tries to perform the same sum reduction shown in Listing 4, but instead of using `__ballot_sync()` to compute the mask before the branch, it uses `__activemask()` inside the branch. This is incorrect, as it would result in partial sums instead of a total sum. The CUDA execution model does not guarantee that all threads taking the branch together will execute the `__activemask()` together. Implicit lock step execution is not guaranteed, as we will explain.

```
//
// 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);
    …
}
```

Warp Synchronization

When threads in a warp need to perform more complicated communications or collective operations than what the data exchange primitives provide, you can use the `__syncwarp()` primitive to synchronize threads in a warp. It is similar to the `__syncthreads()` primitive (which synchronizes all threads in the thread block) but at finer granularity.

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

The `__syncwarp()` primitive causes the executing thread to wait until all threads specified in `mask` have executed a `__syncwarp()` (with the same `mask`) before resuming execution. It also provides a memory fence to allow threads to communicate via memory before and after calling the primitive.

Listing 6 shows an example of shuffling the ownership of matrix elements among threads in a warp.

```
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);
```

Assume a 1-D thread block is used (i.e. threadIdx.y is always 0). At the beginning of the code, each thread in a warp owns one element of a 4×8 matrix with row-major indexing. In other words, lane 0 owns `[0][0]` and lane 1 owns `[0][1]`. Each thread stores its value into the corresponding position of a 4×8 array in shared memory. Then `__syncwarp()` is used to ensure all threads have done the store, before each thread reads from a transposed position in the array. In the end, each thread in the warp owns one element of the matrix with column-major indexing: lane 0 owns `[0][0]` and lane 1 owns `[1][0]`.

Make sure that `__syncwarp()` separates shared memory reads and writes to avoid race conditions. Listing 7 illustrates an incorrect use in a tree sum reduction in shared memory. There is a shared memory read followed by a shared memory write between every two `__syncwarp()` calls. The CUDA programming model does not guarantee that all the reads will be performed before all the writes, so there is a race condition.

```
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();
```

Listing 8 fixes the race condition by inserting extra `__syncwarp()` calls. The CUDA compiler may elide some of these synchronization instructions in the final generated code depending on the target architecture (e.g. on pre-Volta architectures).

```
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;
```

On the latest Volta (and future) GPUs, you can also use `__syncwarp()` in thread-divergent branches to synchronize threads from both branches. But once they return from the primitive, the threads will become divergent again. See Listing 13 for such an example.

Opportunistic Warp-level Programming

As we showed in the Synchronized Data Exchange section, the membership `mask` used in the synchronized data exchange primitives is often computed before a branch condition in the program flow. In many cases, the program needs to pass the mask along the program flow; for example, as a function argument when warp-level primitives are used inside a function. This may be difficult if you want to use warp-level programming inside a library function but you cannot change the function interface.

Some computations can use whatever threads happen to be executing together. We can use a technique called opportunistic warp-level programming, as the following example illustrates. (See this post on

warp-aggregated atomics for more information on the algorithm, and this post for discussion of how Cooperative Groups makes the implementation much simpler.)

```
// 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()` atomically increments the value pointed to by `ptr` by 1 and returns the old value. It uses the `atomicAdd()` function, which may incur contention. To reduce contention, `atomicAggInc` replaces the per-thread `atomicAdd()` operation with a per-warp `atomicAdd()`. The `__activemask()` in line 4 finds the set of threads in the warp that are about to perform the atomic operation. `__match_any_sync()` returns the bit mask of the threads that have the same value `ptr`, partitioning the incoming threads into groups whose members have the same `ptr` value. Each group elects a leader thread (line 5), which performs the `atomicAdd()` (line 8) for the whole group. Every thread gets the old value from the leader (line 9) returned by the `atomicAdd()`. Line 10 computes and returns the old value the current thread would get from `atomicInc()` if it were to call the function instead of `atomicAggInc`.

Implicit Warp-Synchronous Programming is Unsafe

CUDA toolkits prior to version 9.0 provided a (now legacy) version of warp-level primitives. Compared with the CUDA 9 primitives, the legacy primitives do not accept a `mask` argument. For example, `int __any(int predicate)` is the legacy version of `int __any_sync(unsigned mask, int predicate)`.

The `mask` argument, as explained previously, specifies the set of threads in a warp that must participate in the primitives. The new primitives perform intra-warp thread-level synchronization if the threads specified by the mask are not already synchronized during execution.

The legacy warp-level primitives do not allow programmers to specify the required threads and do not perform synchronization. Therefore, the threads that must participate in the warp-level operation are not explicitly expressed by the CUDA program. The correctness of such a program depends on implicit warp-synchronous behavior, which may change from one hardware architecture to another, from one CUDA toolkit release to another (due to changes in compiler optimizations, for example), or even from one run-time execution to another. Such implicit warp-synchronous programming is unsafe and may not work correctly.

For example, in the following code, let's assume all 32 threads in a warp execute line 2 together. The `if` statement at line 4 causes the threads to diverge, with the odd threads calling `foo()` at line 5 and the even threads calling `bar()` at line 8.

```
// 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);
```

The CUDA compiler and the hardware will try to re-converge the threads at line 10 for better performance. But this re-convergence is not guaranteed. Therefore, the `ballot_result` may not contain the ballot result from all 32 threads.

Calling the new `__syncwarp()` primitive at line 10 before `__ballot()`, as illustrated in Listing 11, does not fix the problem either. This is again implicit warp-synchronous programming. It assumes that threads in the same warp that are once synchronized will stay synchronized until the next thread-divergent branch. Although it is often true, it is not guaranteed in the CUDA programming model.

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

The correct fix is to use `__ballot_sync()` as in Listing 12.

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

A common mistake is to assume that calling `__syncwarp()` before and/or after a legacy warp-level primitive is functionally equivalent to calling the `sync` version of the primitive. For example, is `__syncwarp(); v = __shfl(0); __syncwarp();` the same as `__shfl_sync(FULL_MASK, 0)`? The answer is no, for two reasons. First, if the sequence is used in a thread-divergent branch, then `__shfl(0)` won't be executed by all threads together. Listing 13 shows an example. The `__syncwarp()` at line 3 and line 7 would ensure `foo()` is called by all threads in the warp before line 4 or line 8 is executed. Once threads leave the `__syncwarp()`, the odd threads and the even threads become divergent again. Therefore, the `__shfl(0)` at line 4 will get an undefined value because lane 0 is inactive when line 4 is executed. `__shfl_sync(FULL_MASK, 0)` can be used in thread-divergent branches without this problem.

```
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();
}
```

Second, even when the sequence is called by all the threads together, the CUDA execution model does not guarantee threads will stay convergent after leaving `__syncwarp()`, as Listing 14 shows. Implicit lock-step execution is not guaranteed. Remember, thread convergence is guaranteed only within explicitly synchronous warp-level primitives.

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

Because using them can lead to unsafe programs, the legacy warp-level primitives are deprecated starting in CUDA 9.0.

Update Legacy Warp-Level Programming

If your program uses legacy warp-level primitives or any form of implicit warp-synchronous programming (such as communicating between threads of a warp without synchronization), you should update the code to use the `sync` version of the primitives. You may also want to restructure your code to use Cooperative Groups, which provides a higher level of abstraction as well as new features such as multi-block synchronization.

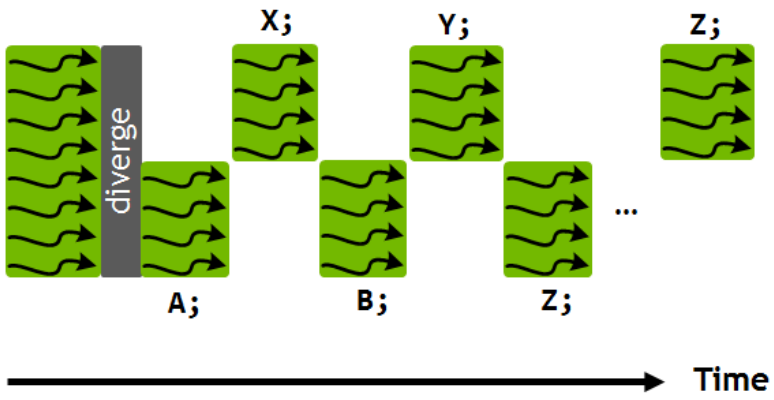
The trickiest part of using the warp-level primitives is figuring out the membership mask to be used. We hope the above sections give you a good idea where to start and what to look out for. Here is a list

of suggestions:

1. Don't just use `FULL_MASK` (i.e. `0xffffffff` for 32 threads) as the `mask` value. If not all threads in the warp can reach the primitive according to the program logic, then using `FULL_MASK` may cause the program to hang.
2. Don't just use `__activemask()` as the mask value. `__activemask()` tells you what threads happen to be convergent when the function is called, which can be different from what you want to be in the collective operation.
3. Do analyze the program logic and understand the membership requirements. Compute the mask ahead based on your program logic.
4. If your program does opportunistic warp-synchronous programming, use "detective" functions such as `__activemask()` and `__match_all_sync()` to find the right mask.
5. Use `__syncwarp()` to separate operations with intra-warp dependences. Do not assume lock-step execution.

One last trick. If your existing CUDA program gives a different result on Volta architecture GPUs, and you suspect the difference is caused by Volta's new independent thread scheduling which can change warp synchronous behavior, you may want to recompile your program with `nvcc` options `-arch=compute_60 -code=sm_70`. Such compiled programs opt-in to Pascal's thread scheduling. When used selectively, it can help pin down the culprit module more quickly, allowing you to update the code to avoid implicit warp-synchronous programming.

```
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.