CUDA trapezoidal rule III: blocks with more than one warp (need to rereview was tired and did not really absorb info)

Limiting ourselves to thread blocks with only 32 threads reduces the power and flexibility of our CUDA programs.

- For example, devices with compute capability ≥ 2.0 can have blocks with as many as 1024 threads or 32 warps, and CUDA provides a fast barrier that can be used to synchronize all the threads in a block.
- So if we limited ourselves to only 32 threads in a block, we wouldn't be using one of the most useful features of CUDA: the ability to efficiently synchronize large numbers of threads.

So what would a "block" sum look like if we allowed ourselves to use blocks with up to 1024 threads?

- Warp shuffle implementation:
- We could use one of our existing warp sums to add the values computed by the threads in each warp. Then we would have as many as 1024/32 = 32 warp sums, and we could use one warp in the thread block to add the warp sums.
- __syncthreads

  We might try to use the following pseudocode for finding the sum of the values computed by all the threads in a block:

  -
    ```
    Each thread computes its contribution;
    Each warp adds its threads' contributions;
    Warp 0 in block adds warp sums;
    ```
    - However, there's a race condition. Do you see it? When warp 0 tries to compute the total of the warp sums in the block, it doesn't know whether all the warps in the block have completed their sums.
    - Recall that the threads in a warp operate in SIMD fashion: no thread in the warp proceeds to a new instruction until all the threads in the warp have completed (or skipped) the current instruction. But the threads in warp 0 can operate independently of the threads in warp 1.
      - So if warp 0 finishes computing its sum before warp 1 computes its sum, warp 0 could try to add warp 1's sum to its sum before warp 1 has finished, and, in this case, the block sum could be incorrect.
  - So we must make sure that warp 0 doesn't start adding up the warp sums until all of the warps in the block are done. We can do this by using CUDA's fast barrier:

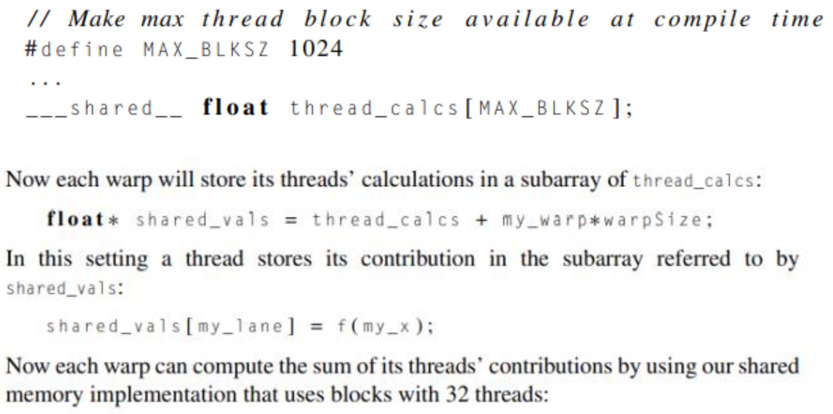
    ```
    __device__ void __syncthreads(void);
    ```
  - New Pseudocode would look like:

    ```
    Each thread computes its contribution;
    Each warp adds its threads' contributions;
    __syncthreads();
    Warp 0 in block adds warp sums;
    ```
    - Now warp 0 won't be able to add the warp sums until every warp in the block has completed its sum
- There are a couple of important caveats when we use __syncthreads
  - First, , it's critical that all of the threads in the block execute the call.
    - If not we get errors
  - The second caveat is that __syncthreads only synchronizes the threads in a block.
    - If a grid contains at least two blocks, and if all the threads in the grid call __syncthreads then the threads in different blocks will continue to operate independently of each other. So we can't synchronize the threads in a general grid with __syncthreads.

- Using shared memory to compute warps:
  - If we try to implement the pseudocode in CUDA, we'll see that there's an important detail that the pseudocode doesn't show: after the call to __syncthreads, how does warp 0 obtain access to the sums computed by the other warps?

    ```
    __shared__ float warp_sum_arr[WARPSZ];
    int my_warp = threadIdx.x / warpSize;
    int my_lane = threadIdx.x % warpSize;
    // Threads calculate their contributions;
    ...
    float my_result = Warp_sum(my_trap);
    if (my_lane == 0) warp_sum_arr[my_warp] = my_result;
    __syncthreads();
    // Warp 0 adds the sums in warp_sum_arr
    ...
    ```

- Shared memory implementation:
- If we're using shared memory instead of warp shuffles to compute the warp sums, we'll need enough shared memory for each warp in a thread block.
  - Since shared variables are shared by all the threads in a thread block, we need an array large enough to hold the contributions of all of the threads to the sum.
    - So we can declare an array with 1024 elements—the largest possible block size—and partition it among the warps:



```
// Make max thread block size available at compile time
#define MAX_BLKSZ 1024
...
___shared__ float thread_calcs[MAX_BLKSZ];
```

Now each warp will store its threads' calculations in a subarray of `thread_calcs`:

```
float* shared_vals = thread_calcs + my_warp*warpSize;
```

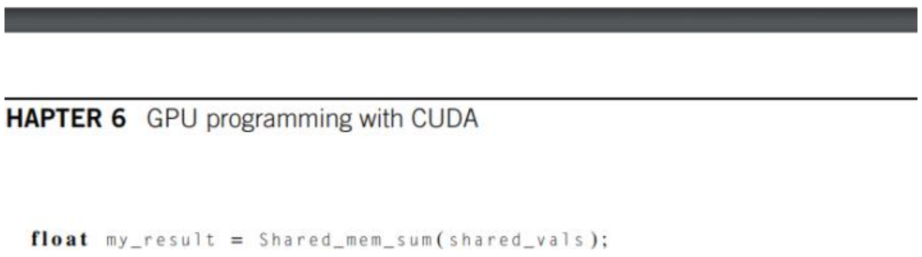
In this setting a thread stores its contribution in the subarray referred to by `shared_vals`:

```
shared_vals[my_lane] = f(my_x);
```

Now each warp can compute the sum of its threads' contributions by using our shared memory implementation that uses blocks with 32 threads:

Note: being lazy so took screenshot
- Each warp will store in its shared memory a section of the thread_calcs array to add its final computation into

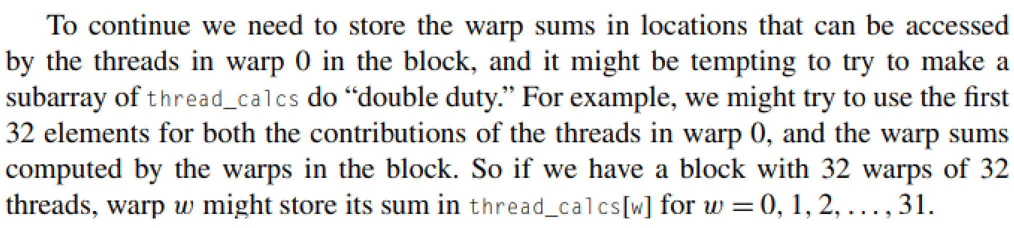
  -



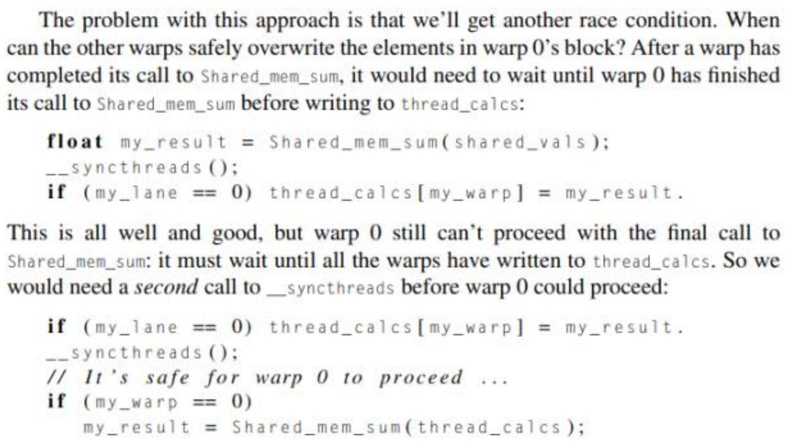
HAPTER 6 GPU programming with CUDA

```
float my_result = Shared_mem_sum(shared_vals);
```

-



To continue we need to store the warp sums in locations that can be accessed by the threads in warp 0 in the block, and it might be tempting to try to make a subarray of `thread_calcs` do "double duty." For example, we might try to use the first 32 elements for both the contributions of the threads in warp 0, and the warp sums computed by the warps in the block. So if we have a block with 32 warps of 32 threads, warp w might store its sum in `thread_calcs[w]` for $w = 0, 1, 2, \ldots, 31$.



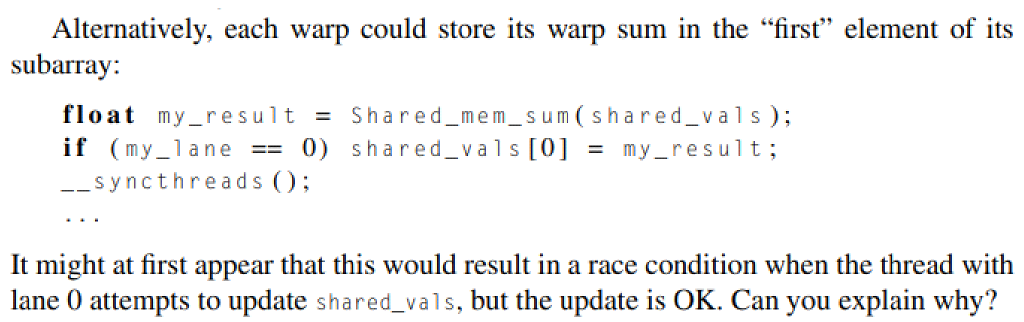
The problem with this approach is that we'll get another race condition. When can the other warps safely overwrite the elements in warp 0's block? After a warp has completed its call to `Shared_mem_sum`, it would need to wait until warp 0 has finished its call to `Shared_mem_sum` before writing to `thread_calcs`:

```
float my_result = Shared_mem_sum(shared_vals);
__syncthreads();
if (my_lane == 0) thread_calcs[my_warp] = my_result.
```

  - This is all well and good, but warp 0 still can't proceed with the final call to `Shared_mem_sum`: it must wait until all the warps have written to `thread_calcs`. So we would need a *second* call to `__syncthreads` before warp 0 could proceed:

```
if (my_lane == 0) thread_calcs[my_warp] = my_result.
__syncthreads();
// It's safe for warp 0 to proceed ...
if (my_warp == 0)
   my_result = Shared_mem_sum(thread_calcs);
```

    - We get a race condition if we try this
    - We could fix it by adding another __syncthreads() but those are costly and we should try to minimize its usage if possible
  - An alternative approach would be:

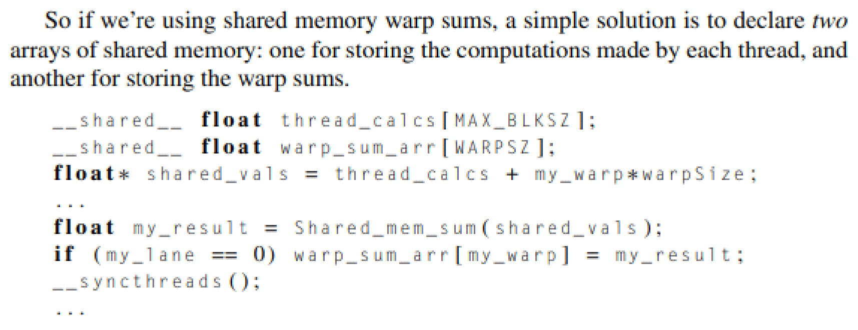


Alternatively, each warp could store its warp sum in the "first" element of its subarray:

```
float my_result = Shared_mem_sum(shared_vals);
if (my_lane == 0) shared_vals[0] = my_result;
__syncthreads();
...
```

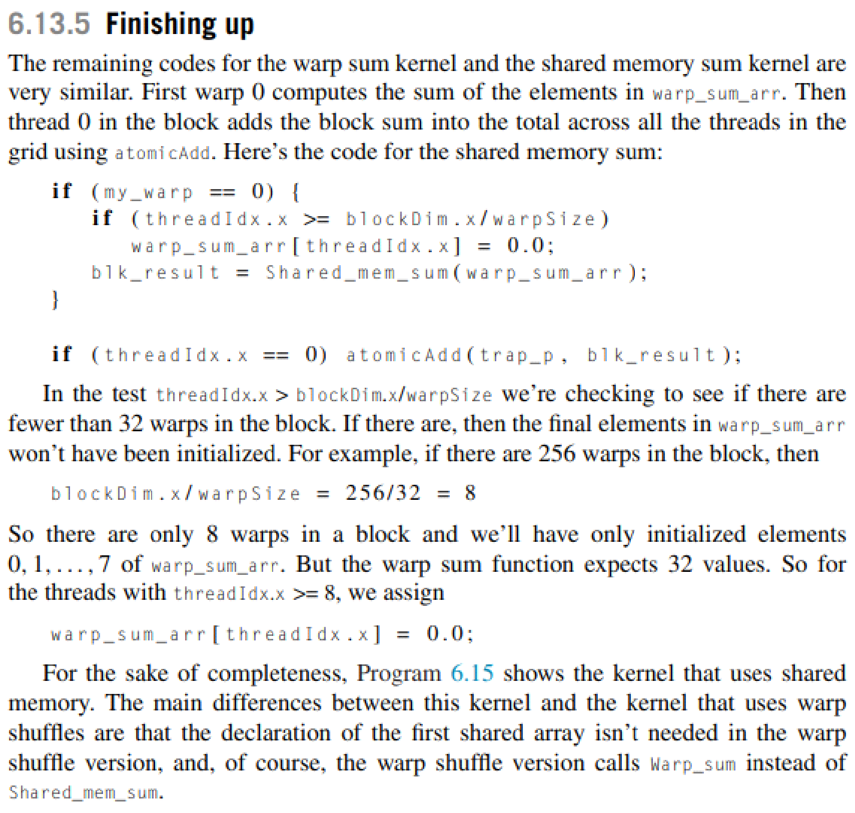
It might at first appear that this would result in a race condition when the thread with lane 0 attempts to update `shared_vals`, but the update is OK. Can you explain why?

- Because of how NVIDIA designed shared memory this still isnt the best approach, here is a simple but effective solution:



So if we're using shared memory warp sums, a simple solution is to declare *two* arrays of shared memory: one for storing the computations made by each thread, and another for storing the warp sums.

```
__shared__ float thread_calcs[MAX_BLKSZ];
__shared__ float warp_sum_arr[WARPSZ];
float* shared_vals = thread_calcs + my_warp*warpSize;
...
float my_result = Shared_mem_sum(shared_vals);
if (my_lane == 0) warp_sum_arr[my_warp] = my_result;
__syncthreads();
...
```



6.13.5 Finishing up

The remaining codes for the warp sum kernel and the shared memory sum kernel are very similar. First warp 0 computes the sum of the elements in `warp_sum_arr`. Then thread 0 in the block adds the block sum into the total across all the threads in the grid using `atomicAdd`. Here's the code for the shared memory sum:

```
if (my_warp == 0) {
   if (threadIdx.x >= blockDim.x/warpSize)
      warp_sum_arr[threadIdx.x] = 0.0;
   blk_result = Shared_mem_sum(warp_sum_arr);
}

if (threadIdx.x == 0) atomicAdd(trap_p, blk_result);
```

In the test `threadIdx.x` > `blockDim.x/warpSize` we're checking to see if there are fewer than 32 warps in the block. If there are, then the final elements in `warp_sum_arr` won't have been initialized. For example, if there are 256 warps in the block, then

```
blockDim.x/warpSize = 256/32 = 8
```

So there are only 8 warps in a block and we'll have only initialized elements $0, 1, \ldots, 7$ of `warp_sum_arr`. But the warp sum function expects 32 values. So for the threads with `threadIdx.x` >= 8, we assign

```
warp_sum_arr[threadIdx.x] = 0.0;
```

For the sake of completeness, Program 6.15 shows the kernel that uses shared memory. The main differences between this kernel and the kernel that uses warp shuffles are that the declaration of the first shared array isn't needed in the warp shuffle version, and, of course, the warp shuffle version calls `Warp_sum` instead of `Shared_mem_sum`.

- Code:

```
__global__ void Dev_trap(
      const float a        /* in  */,
      const float b        /* in  */,
      const float h        /* in  */,
      const int   n        /* in  */,
      float*      trap_p   /* out */) {
   __shared__ float thread_calcs[MAX_BLKSZ];
   __shared__ float warp_sum_arr[WARPSZ];
   int my_i = blockDim.x * blockIdx.x + threadIdx.x;
   int my_warp = threadIdx.x / warpSize;
   int my_lane = threadIdx.x % warpSize;
   float* shared_vals = thread_calcs + my_warp*warpSize;
   float blk_result = 0.0;

   shared_vals[my_lane] = 0.0f;
   if (0 < my_i && my_i < n) {
      float my_x = a + my_i*h;
      shared_vals[my_lane] = f(my_x);
   }

   float my_result = Shared_mem_sum(shared_vals);
   if (my_lane == 0) warp_sum_arr[my_warp] = my_result;
   __syncthreads();

   if (my_warp == 0) {
      if (threadIdx.x >= blockDim.x/warpSize)
         warp_sum_arr[threadIdx.x] = 0.0;
      blk_result = Shared_mem_sum(warp_sum_arr);
   }

   if (threadIdx.x == 0) atomicAdd(trap_p, blk_result);
}  /* Dev_trap */
```

Program 6.15: CUDA kernel implementing trapezoidal rule and using shared memory. This version can use large thread blocks.

Performance:

- Both do better than previous implementations

Table 6.10 Mean run-times for trapezoidal rule using arbitrary block size (times in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Original	33.6	20.7	4.48	3.08
Warp Shuffle, 32 ths/blk		14.4		0.210
Shared Memory, 32 ths/blk		15.0		0.206
Warp Shuffle		12.8		0.141
Shared Memory		14.3		0.150