CUDA trapezoidal rule II: improving performance

If you've read the Pthreads or OpenMP chapter, you can probably make a good guess at how to make the CUDA program run faster. For a thread's call to `atomicAdd` to actually be atomic, no other thread can update `*trap_p` while the call is in progress. In other words, the updates to `*trap_p` can't take place simultaneously, and our program may not be very parallel at this point.

One way to improve the performance is to carry out a tree-structured global sum that's similar to the tree-structured global sum we introduced in the MPI chapter (Section 3.4.1). However, because of the differences between the GPU architecture and the distributed-memory CPU architecture, the details are somewhat different.

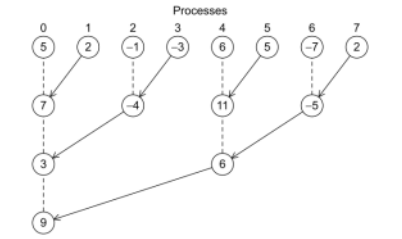

FIGURE 3.6
A tree-structured global sum.

Tree-structured communication

- In our previous implementation of the trap rule , threads are executed by our system and CUDA in more or less a random ordering when it comes to adding to our "global sum" that we called trap_p (trap variable)
  - **Table 6.6** Basic global sum with eight threads.

Time	Thread	my_trap	*trap_p
Start	—	—	9
t_0	5	11	20
t_1	2	5	25
t_2	3	7	32
t_3	7	15	47
t_4	4	9	56
t_5	6	13	69
t_6	0	1	70
t_7	1	3	73

    - T being the time or iteration
    - Notice that the threads are executing in a random order
    - At the start we note that we always do the first and last iterations manually so we start with 9
- What's important is that this approach may serialize the threads. So the computation may require a sequence of 8 calculations.
  - So rather than have each thread wait for its turn to do an addition into *trap_p, we can pair up the threads so that half of the "active" threads add their partial sum to their partner's partial sum. This gives us a structure that resembles a tree
  - Why is this good?

    In our figures, we've gone from requiring a sequence of 8 consecutive additions to a sequence of 4. More generally, if we double the number of threads and values (e.g., increase from 8 to 16), we'll double the length of the sequence of additions using the basic approach, while we'll only add one using the second, tree-structured approach. For example, if we increase the number of threads and values from 8 to 16, the first approach requires a sequence of 16 additions, but the tree-structured approach only requires 5. In fact, if there are t threads and t values, the first approach requires a sequence of t additions, while the tree-structured approach requires $\lceil \log_2(t) \rceil + 1$. For example, if we have 1000 threads and values, we'll go from 1000 communications and sums using the basic approach to 11 using the tree-structured approach, and if we have 1,000,000, we'll go from 1,000,000 to 21!

    - In summary, it saves us on performance
- There are two ways to implement this tree structure in CUDA: one implementation uses shared memory (best for devices with a compute capability < 3) and there are functions called warp shuffles that allow a collection of threads within a warp to

read variables stored on other threads in the warp (able to be used in devices with a compute capability >= 3)

- Before we talk about how warp shuffles work we will talk about how memory works in CUDA.
  - Local variables, registers, shared and global memory
  - we mentioned that SMs in an Nvidia processor have access to two collections of memory locations: each SM has access to its own "shared" memory, which is accessible only to the SPs belonging to the SM.
    - More precisely, the shared memory allocated for a thread block (SM) is only accessible to the threads in that block. On the other hand, all of the SPs and all of the threads have access to "global" memory.
    - The number of shared memory locations is relatively small, but they are quite fast, while the number of global memory locations is relatively large, but they are relatively slow.
  - We can say that a GPU has three levels of memory:
    - At the bottom, is the slowest, largest level: global memory.
    - In the middle is a faster, smaller level: shared memory.
    - At the top is the fastest, smallest level: the registers.
  - An obvious question here: what about local variables? How much storage is available for them? And how fast is it?
    - This depends on total available memory and program memory usage. If there is enough storage, local variables are stored in registers
    - However, if there isn't enough register storage, local variables are "spilled" to a region of global memory that's thread private, i.e., only the thread that owns the local variables can access them.
    - So as long as we have sufficient register storage, we expect the performance of a kernel to improve if we increase our use of registers and reduce our use of shared and/or global memory
- Warps and warp shuffles
  - In particular, if we can implement a global sum in registers, we expect its performance to be superior to an implementation that uses shared or global memory, and the warp shuffle functions introduced in CUDA 3.0 allow us to do this.
  - In CUDA a **warp** is a set of threads with consecutive ranks belonging to a thread block.
    - The number of threads in a warp is currently 32
      - Will need to have total threads in an SM be a multiple of 32, as we will see later
    - There is a variable initialized by the system that stores the size of a warp:

```
int warpSize
```

    - Essentially warps group up threads in an SM and each thread in the warp get some rank
      - The rank of a thread within a warp is called the **thread's lane**, and it can be computed using the formula

```
lane = threadIdx.x % warpSize;
```

    - The threads in a warp operate in SIMD fashion.
      - So threads in different warps can execute different statements with no penalty, while threads within the same warp must execute the same statement.
  - The warp shuffle functions allow the threads in a warp to read from registers used by another thread in the same warp.

```
__device__ float __shfl_down_sync(
           unsigned   mask              /* in */,
           float      var               /* in */,
           unsigned   diff              /* in */,
           int        width = warpSize  /* in */);
```

    - The mask argument indicates which threads are participating in the call
      - We use hexadecimal to define this argument.
      - We typically end up using all the threads in a warp so this argument is usually:

```
mask = 0xffffffff;
```

        Recall that 0x denotes a hexadecimal (base 16) value and 0xf is 15_10, which is 1111_2. So this value of mask is 32 1's in binary, and it indicates that every thread in the warp (32 threads in each warp) participates in the call to __shfl_down_sync.
    - The var argument is the variable we are passing/shuffling from one thread to the other
    - The diff argument is The number of lane positions a thread's value is moved downward within the warp.
      - lane + diff will match a thread in the warp with another thread in the warp to do the operation with
      - Typically what will happen is a thread with say lane = 1 will do a lane + diff to get another thread that is higher ranked say diff=3 so 1+ 3=4 and so in this case lane 1 calls for lane 4 and then lane 4 passes its var into lane 1 to compute
    - Width is amount of threads being considered
  - There are several possible issues that could occur:

We'll only use `width = warpSize`, and since its default value is `warpSize`, we'll omit it from our calls.

Note:

- Width is amount of threads being considered

- There are several possible issues that could occur:

We'll only use `width = warpSize`, and since its default value is `warpSize`, we'll omit it from our calls.

There are several possible issues:

- What happens if thread l calls `__shfl_down_sync` but thread $l + $ `diff` doesn't? In this case, the value returned by the call on thread l is *undefined*.
- What happens if thread l calls `__shfl_down_sync` but $l + $ `diff` $\geq$ `warpSize`? In this case the call will return the value in `var` already stored on thread l.
- What happens if thread l calls `__shfl_down_sync`, and $l + $ `diff` $<$ `warpSize`, but $l + $ `diff` $>$ largest lane in the warp. In other words, because the thread block size is not a multiple of `warpSize`, the last warp in the block has fewer than `warpSize` threads. Say there are w threads in the last warp, where $0 < w <$ `warpSize`. Then if

$$l + \texttt{diff} \geq w,$$

the value returned by the call is also undefined.

So to avoid undefined results, it's best if

- All the threads in the warp call `__shfl_down_sync`, and
- All the warps have `warpSize` threads, or, equivalently, the thread block size (`blockDim.x`) is a multiple of `warpSize`.

Note:

1) This one essentially says what if one thread calls the function and the other one doesn't
   a. It is best practice for all threads in a warp to call this function
2) This one is essentially saying that if lane + diff goes beyond the scope of the warpSize (32) then the lane calling for the other thread will just use its own var
   a. This essentially stops the thread from influencing other threads since it has no buddy
3) This is essentially reiterating that the thread block size must be a multiple of warpSize (32) so that all warp blocks in a warp have the same number of threads and no warp blocks are different

To reiterate the solutions to these problems:

- All the threads in the warp call `__shfl_down_sync`, and
- All the warps have `warpSize` threads, or, equivalently, the thread block size (`blockDim.x`) is a multiple of `warpSize`.

- Implementing tree-structured global sum with a warp shuffle
  - So we can implement a tree-structured global sum using the following code:

```
__device__ float Warp_sum(float var) {
   unsigned mask = 0xffffffff;

   for (int diff = warpSize/2; diff > 0; diff = diff/2)
      var += __shfl_down_sync_sync(mask, var, diff);
   return var;
}  /* Warp_sum */
```

Note:

- Diff will half in two every iteration because every iteration in this program we no longer need the values from the threads (lane + diff) that we fetched data from when doing lane + diff.
  - In this way we essentially cut off the threads we already fetched the data from the "crucial threads" still in play

  - It would look something like this:

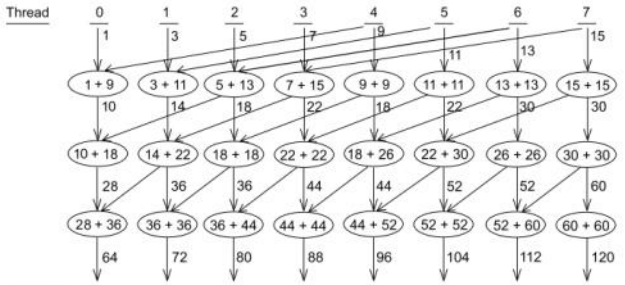

FIGURE 6.5

Tree-structured sum using warp shuffle.

Note:

- this is if warpSize was 8 and not 32, like it would typically be

Fig. 6.5 shows how the function would operate if `warpSize` were 8. (The diagram would be illegible if we used a `warpSize` of 32.) Perhaps the most confusing point in the behavior of `__shfl_down_sync` is that when the lane ID

$$l + \texttt{diff} \geq \texttt{warpSize},$$

the call returns the value in the *caller's* `var`. In the diagram this is shown by having only one arrow entering the oval with the sum, and it's labeled with the value just calculated by the thread carrying out the sum. In the row corresponding to `diff` = 4 (the first row of sums), the threads with lane IDs $l = 4, 5, 6$, and 7 all have $l + 4 \geq 8$. So the call to `__shfl_down_sync` returns their current `var` values, 9, 11, 13, and 15, respectively, and these values are doubled, because the return value of the call is added into the calling thread's variable `var`. Similar behavior occurs in the row corresponding to the sums for `diff` = 2 and lane IDs $l = 6$ and 7, and in the last row when `diff` = 1 for the thread with lane ID $l = 7$.

From a practical standpoint, it's important to remember that this implementation will only return the correct sum on the thread with lane ID 0. If all of the threads need the result, we can use an alternative warp shuffle function, `__shfl_xor`. See Exercise 6.6.

Notes:

- We already discussed this above but to go a little further…
- We halve the diff each time to remove from the game the threads (lane + diff) we have already read from
- Some confusing things happen as depicted in the image above when…
  - Lane + diff > warpSize, we see that the thread calls for a buddy and it doesn't have one in scope thus it just "plays" with itself
  - When we half the diff the threads which were "out of play" can now find a buddy to play with but since all of these are not the "crucial" threads they don't matter and thus we don't care what they do with each other

  - Note that thread with lane 0 in the warp will be the thread with the Final sum in this implementation

- Shared memory and an alternative to the warp shuffle
  - If your GPU has a computation capability < 3, then you cannot use the warp shuffle functions in your code and thus you cannot access the registers of other threads within the same warp.
  - However we can implement another way of creating a tree like structure by using shared memory where threads in the

same thread block can all access the same shared memory.
    - In fact, although shared memory access is slower than register access, we'll see that the shared memory implementation can be just as fast as the warp shuffle implementation
- Since the threads belonging to a single warp operate synchronously, we can implement something very similar to a warp shuffle using shared memory instead of registers.

```
__device__ float Shared_mem_sum(float shared_vals[]) {
   int my_lane = threadIdx.x % warpSize;

   for (int diff = warpSize/2; diff > 0;  diff = diff/2) {
      /* Make sure 0 <= source < warpSize */
      int source = (my_lane + diff) % warpSize;
      shared_vals[my_lane] += shared_vals[source];
   }
   return shared_vals[my_lane];
}
```

    - This should be called by all the threads in a warp, and the array shared_vals should be stored in the shared memory of the SM that's running the warp.
        - Technically we could have shared_vals[] be in the global memory but we should explicitly make it shared memory to take advantage of the speed shared memory has over global memory
    - Since the threads in the warp are operating in SIMD fashion, they effectively execute the code of the function in lockstep.
        - So there's no race condition in the updates to shared_vals: all the threads read the values in shared_vals[source] before any thread updates shared_vals[my_lane].
- Technically speaking, this isn't a tree-structured sum. It's sometimes called a dissemination sum or dissemination reduction. Fig. 6.6 illustrates the copying and additions that take place.

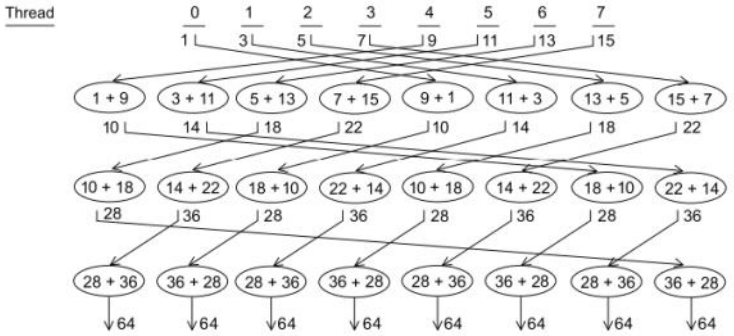

FIGURE 6.6

Dissemination sum using shared memory.

    - Also note that every thread reads a value from another thread in each pass through the for statement unlike in warp shuffle.
    - Also note, after all these values have been added in, every thread has the correct sum—not just thread 0.
        - This can be useful in other applications
- Also, each thread here is either executing the same instruction as every other thread or it is idle
- An obvious question here is: how does Shared_mem_sum make use of Nvidia's shared memory?
    - The answer is that it's not required to use shared memory. The function's argument, the array shared_vals, could reside in either global memory or shared memory.
        - In either case, the function would return the sum of the elements of shared_vals.
    - However, to get the best performance, the argument shared_vals should be defined to be __shared__ in a kernel.

```
__shared__ float shared_vals[32];
```

        - For each thread block this sets aside storage for a collection of 32 floats in the shared memory of the SM to which the block is assigned.
        - Alternatively, if it isn't known at compile time how much shared memory is needed, it can be declared as
            - `extern __shared__ float shared_vals[];`
                - and when the kernel is called, a third argument can be included in the triple angle brackets specifying the size in bytes of the block of shared memory
                - For example, if we were using Shared_mem_sum in a trapezoidal rule program, we might call the kernel Dev_trap with

```
Dev_trap <<<blk_ct, th_per_blk, th_per_blk*sizeof(float)>>>
      (... args to Dev_trap ...);
```

- This would allocate storage for th_per_blk floats in the shared_vals array in each thread block.

- Implementation of trapezoidal rule with warpSize thread blocks
  - We will showcase both implementations of the tree structure (warp shuffle and shared memory) for our program.
  - What we can assume in both versions and what things are similar:
    - For both versions we'll assume that the thread blocks consist of warpSize threads
    - we'll use one of our "tree-structured" sums to add the results of the threads in the warp
    - After computing the function values and adding the results within a warp, the thread with lane ID 0 in the warp will add the warp sum into the total using Atomic_add.
    - Host code:
      - For both the warp shuffle and the shared memory versions, the host code is virtually identical to the code for our first CUDA version. The only substantive difference is that there is no th_per_blk variable in the new versions, since we're assuming that each thread block has warpSize threads.
        - Th_per_blk will be the warpSize so no need to specify it, this also helps prevent those errors we saw in warp shuffle when we don't have a multiple of 32 for warps

  - Kernel warp Shuffle implementation:

```
1  __global__ void Dev_trap(
2        const float  a        /*  in   */,
3        const float  b        /*  in   */,
4        const float  h        /*  in   */,
5        const int    n        /*  in   */,
6        float*       trap_p   /* in/out */) {
7     int my_i = blockDim.x * blockIdx.x + threadIdx.x;
8
9     float my_trap = 0.0f;
10    if (0 < my_i && my_i < n) {
11       float my_x = a + my_i*h;
12       my_trap = f(my_x);
13    }
14
15    float result = Warp_sum(my_trap);
16
17    /* result is correct only on thread 0 */
18    if (threadIdx.x == 0) atomicAdd(trap_p, result);
19 }  /* Dev_trap */
```

Program 6.13: CUDA kernel implementing trapezoidal rule and using Warp_sum.

Notes:
- Very similar to original implementation
- However, instead of adding each
- thread's calculation directly into *trap_p, each warp (or, in this case, thread block) calls the Warp_sum function to add the values computed by the threads in the warp.
- Then, when the warp returns, thread (or lane) 0 adds the warp sum for its thread block (result) into the global total.
- Since, in general, this version will use multiple thread blocks, there will be multiple warp sums that need to be added to *trap_p. So if we didn't use atomicAdd, the addition of result to *trap_p would form a race condition.

  - Kernel with shared memory implementation:

```
1  __global__ void Dev_trap(
2        const float  a        /* in  */,
3        const float  b        /* in  */,
4        const float  h        /* in  */,
5        const int    n        /* in  */,
6        float*       trap_p   /* out */) {
7     __shared__ float shared_vals[WARPSZ];
8     int my_i = blockDim.x * blockIdx.x + threadIdx.x;
9     int my_lane = threadIdx.x % warpSize;
10
11    shared_vals[my_lane] = 0.0f;
12    if (0 < my_i && my_i < n) {
13       float my_x = a + my_i*h;
14       shared_vals[my_lane] = f(my_x);
15    }
16
17    float result = Shared_mem_sum(shared_vals);
18
19    /* result is the same on all threads in a block. */
20    if (threadIdx.x == 0) atomicAdd(trap_p, result);
21 }  /* Dev_trap */
```

Program 6.14: CUDA kernel implementing trapezoidal rule and using shared memory.

Notes:
- Very similar to the warp shuffle implantation
- The main differences are that it declares an array of shared memory in Line 7;
  - it initializes this array in Lines 11 and 14;
- and, of course, the call to Shared_mem_sum is passed this array rather than a scalar register
- Since we know at compile time how much storage we'll need in shared_vals, we can define the array by simply preceding the ordinary C definition with the CUDA qualifier __shared__: __shared__ float shared_vals [WARPSZ] ;
  - Note that the CUDA defined variable warpSize is not defined at compile-time. So our program defines a preprocessor macro

```
#define WARPSZ 32
```

- Note that for both these implementations, the functions Warp_sum and shared_mem_sum we talked about earlier (towards the top of the page)
- Performance:

6.12.4 Performance

Of course, we want to see how the various implementations perform. (See Table 6.8.) The problem is the same as the problem we ran earlier (see Table 6.5): we're integrating $f(x) = x^2 + 1$ on the interval $[-3, 3]$, and there are $2^{20} = 1{,}048{,}576$ trapezoids. However, since the thread block size is 32, we're using 32,768 thread blocks ($32 \times 32{,}768 = 1{,}048{,}576$).

6.13 CUDA trapezoidal rule III: blocks with more than one warp

Table 6.8 Mean run-times for trapezoidal rule using block size of 32 threads (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		14.4		0.210
Shared Memory		15.0		0.206

We see that on both systems and with both sum implementations, the new programs do significantly better than the original. For the GK20A, the warp shuffle version runs in about 70% of the time of the original, and the shared memory version runs in about 72% of the time of the original. For the Titan X, the improvements are much more impressive: both versions run in less than 7% of the time of the original. Perhaps most striking is the fact that on the Titan X, the warp shuffle is, on average, slightly slower than the shared memory version.