CUDA trapezoidal rule I

Recall what the trapezoidal rule is:

```
1  float Serial_trap(
2        const float  a  /* in */,
3        const float  b  /* in */,
4        const int    n  /* in */) {
5     float x, h = (b-a)/n;
6     float trap = 0.5*(f(a) + f(b));
7
8     for (int i = 1; i <= n-1; i++) {
9        x = a + i*h;
10       trap += f(x);
11    }
12    trap = trap*h;
13
14    return trap;
15 } /* Serial_trap */
```

Program 6.11: A serial function implementing the trapezoidal rule for a single CPU.

- Note, this is a serial implementation for the trapezoidal rule

Implementing the trapezoidal rule with CUDA

- Applying Fosters methodology
  - If n is large, the vast majority of the work in the serial implementation is done by the for loop.
  - we're mainly interested in two types of tasks:
    - the first is the evaluation of the function f at x_i
    - the second is the addition of f (x_i) into trap.
  - We also note that the second task depends on the first so we package them together
  - This suggests that each thread in our CUDA implementation might carry out one iteration of the serial for loop.
    - We can assign a unique integer rank to each thread as we did with the vector addition program.
- What we can do is compute an x-value, the function value, and add the function value to the "running sum"
  - The kernel would look something like this:

    - There are many problems wit this implementation:
      1. We haven't initialized `h` or `trap`.
      2. The `my_i` value can be too large or too small: the serial loop ranges from 1 up to and including $n - 1$. The smallest value for `my_i` is 0 and the largest is the total number of threads minus 1.
      3. The variable `trap` must be shared among the threads. So the addition of `my_trap` forms a race condition: when multiple threads try to update `trap` at roughly the same time, one thread can overwrite another thread's result, and the final value in trap may be wrong. (For a discussion of race conditions, see Section 2.4.3.)
      4. The variable `trap` in the serial code is returned by the function, and, as we've seen, kernels must have **void** return type.
      5. We see from the serial code that we need to multiply the total in `trap` by `h` after all of the threads have added their results.

How would we write the CUDA implementation that would deal with the problems highlighted above

- The program:

```
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     /* f(x_0) and f(x_n) were computed on the host.  So */
10    /* compute f(x_1), f(x_2), ..., f(x_(n-1))         */
11    if (0 < my_i && my_i < n) {
12       float my_x = a + my_i*h;
13       float my_trap = f(my_x);
14       atomicAdd(trap_p, my_trap);
15    }
16 }  /* Dev_trap */
17
18 /* Host code */
19 void Trap_wrapper(
20       const float  a           /* in  */,
21       const float  b           /* in  */,
22       const int    n           /* in  */,
23       float*       trap_p      /* out */,
24       const int    blk_ct      /* in  */,
25       const int    th_per_blk  /* in  */) {
26
27    /* trap_p storage allocated in main with
28     * cudaMallocManaged */
29    *trap_p = 0.5*(f(a) + f(b));
30    float h = (b-a)/n;
31
32    Dev_trap<<<blk_ct, th_per_blk>>>(a, b, h, n, trap_p);
33    cudaDeviceSynchronize();
34
35    *trap_p = h*(*trap_p);
36 }  /* Trap_wrapper */
```

Program 6.12: CUDA kernel and wrapper implementing trapezoidal rule.

- Below we are going to look deeper into why we made the choices we made in this CUDA implementation and how it handled the problems above
- One way we could try to solve many of the problems (1-5) listed above is by having one thread, say thread 0 from block 0, initialize all of the things we need and then send out that data to all the other threads.
  - This is not a good idea and wrong.
  - The reason we first bring this up is to make the point that kernel and function arguments are private to the executing thread
    - Each thread has its own stack, private variables (h and trap).
    - Any changes to these variables made on one thread wont be visible to other threads.
- The more sensible approach would be to initialize these variables, like h and trap, on the host and then give each thread a copy of these variables
  - The variable h is fine since it never changes
  - Things are more complicated with trap variable since it is updated by the threads.
    - We can first create a pointer to the trap variable and allocate trap and finally before the call to the kernel we can pass that trap pointer to the kernel using cudaMallocMemory.

```
/* Host code */
float* trap_p;
cudaMallocManaged(&trap_p, sizeof(float));
...
```



```
*trap_p = 0.5*(f(a) + f(b));

/* Call kernel */
...

/* After return from kernel */
*trap_p = h*(*trap_p);
```

    - Note that we use a pointer to float because if we tried to use a regular float value, we would have to return it on the kernel side which isnt allowed, kernels cannot return values
    - When we do this each thread essentially gets a copy of the trap variable but all the copies of trap will refer to the same memory location on the GPU (or kernel side of things).

A wrapper function

A wrapper function

If you look at the code in Program 6.12, you'll see that we've placed most of the code we use before and after calling the kernel in a **wrapper function**, `Trap_wrapper`. A wrapper function is a function whose main purpose is to call another function. It can perform any preparation needed for the call. It can also perform any additional work needed after the call.

- Using the correct threads

We assume that the number of threads, `blk_ct*th_per_blk`, is at least as large as the number of trapezoids. Since the serial **for** loop iterates from 1 up to `n`−1, thread 0 and any thread with `my_i` $> n - 1$, shouldn't execute the code in the body of the serial **for** loop. So we should include a test before the main part of the kernel code

```
if (0 < my_i && my_i < n) {
   /* Compute x, f(x), and add in to *trap_p */
   ...
}
```

See Line 11 in Program 6.12.

- This code ensures that since each thread is handling one iteration, the thread is withing the bounds of the number of trapizoids we are using.
  - All the other threads not in this scope should do nothing
- Updating the return value and atomicAdd
  - What happens when we update the trap variable in the kernel

```
*trap_p += my_trap;
```

  - An issue is that we will have a race condition if we don't handle this variable since it is being updated by all the threads.
  - We will solve this problem by using a function provided by CUDA function called atomicAdd
    - What does it mean for an operation to be atomic?
      if it appears to all the other threads as if it were "indivisible."
      - So if another thread tries to access the result of the operation or an operand used in the operation, the access will occur either before the operation started or after the operation completed.
      - Effectively, then, the operation appears to consist of a single, indivisible, machine instruction.
      - This all means that to make addition atomic, no other thread should be able to update trap in the kernel while one is updating it
    - Adding is not ordinarily atomic as there are mulitple instructions at play when adding
      - Meaning that if one thread is executing an addition, it's possible for another thread to access the operands and the result while the addition is in progress.
    - For this reason, we CUDA provided many atomic addition funcitons, we use atomicAdd

```
__device__ float atomicAdd(
      float*  float_p   /* in/out */,
      float   val       /* in     */);
```

      - This atomically adds the contents of val to the contents of the memory referred to by float_p and stores the result in the memory referred to by float_p. It returns the value of the memory referred to by float_p at the beginning of the call.
- Performance of the CUDA trapezoidal rule
  - We can find the run-time of our trapezoidal rule by finding the execution time of the Trap_wrapper function.
    - The execution of this function includes all of the computations carried out by the serial trapezoidal rule, including the initialization of *trap_p (Line 29) and h (Line 30), and the final update to *trap_p (Line 35).
    - It also includes all of the calculations in the body of the serial for loop in the Dev_trap kernel.
  - So we can effectively determine the run-time of the CUDA trapezoidal rule by timing a host function, and we only need to insert calls to our timing functions before and after the call to Trap_wrapper.

```
double start, finish;
...
GET_TIME(start);
Trap_wrapper(a, b, n, trap_p, blk_ct, th_per_blk);
GET_TIME(finish);
printf("Elapsed time for cuda = %e seconds\n",
      finish-start);
```

  - We typically want to measure the mean or median of our elapsed time
    - We would run something like 50 executions to get the median or mean
    - This is because CPU and GPU speeds play a role in the timings

Table 6.5 Mean run-times for serial and CUDA trapezoidal rule (times are 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
Run-time	33.6	20.7	4.48	3.08

- Example:

When we run the serial trapezoidal and the CUDA trapezoidal rule functions many times and take the means of the elapsed times, we get the results shown in Table 6.5. These were taken using $n = 2^{20} = 1{,}048{,}576$ trapezoids with $f(x) = x^2 + 1$, $a = -3$, and $b = 3$. The GPUs use 1024 blocks with 1024 threads per block for a total of 1,048,576 threads. The 192 SPs of the GK20A are clearly much faster than a fairly slow conventional processor, an ARM Cortex-A15, but a single core of an Intel Core i7 is much faster than the GK20A. The 3072 SPs on a Titan X *were* 45% faster than the single core of the Intel, but it would seem that with 3072 SPs, we should be able to do better.