List Reduction

Due Date: October 23 at 11:59 pm

Objective

Implement a kernel and associated host code that performs reduction of a 1D list stored in a C array. The reduction should give the sum of the list. You should implement the improved kernel discussed in the lecture. Your kernel should be able to handle input lists of arbitrary length.

Instruction

Edit the code to perform the following:

- allocate device memory
- copy host memory to device
- initialize thread block and kernel grid dimensions
- invoke CUDA kernel
- copy results from device to host
- deallocate device memory
- implement the improved reduction kernel
- use shared memory to reduce the number of global accesses, handle the boundary conditions when loading input list elements into the shared memory
- implement a CPU loop to perform final reduction based on the sums of sections generated by the thread blocks after copying the partial sum array back to the host memory

Instructions about where to place each part of the code is demarcated by the `//@@` comment lines.

Suggestions (for all labs)

- Backup your code regularly.
- Do not modify the template code provided -- only insert code where the `//@@` demarcation is placed
- Develop your solution incrementally and test each version thoroughly before moving on to the next version
- Do not wait until the last minute to attempt the lab.

- If you get stuck with boundary conditions, grab a pen and paper. It is much easier to figure out the boundary conditions there.
- Implement the serial CPU version first, this will give you an understanding of the loops
- Get the first dataset working first. The datasets are ordered so the first one is the easiest to handle
- Make sure that your algorithm handles non-regular dimensional inputs (not square or multiples of 2). The slides may present the algorithm with nice inputs, since it minimizes the conditions. The datasets reflect different sizes of input that you are expected to handle
- Make sure that you test your program using all the datasets provided (the datasets can be selected using the dropdown next to the submission button)
- Check for errors: for example, when developing CUDA code, one can check for if the function call succeeded and print an error if not via the following macro:

  ```
  #define wbCheck(stmt) do {                                          \
          cudaError_t err = stmt;                                     \
          if (err != cudaSuccess) {                                   \
              wbLog(ERROR, "Failed to run stmt ", #stmt);             \
              wbLog(ERROR, "Got CUDA error ...  ",
  cudaGetErrorString(err));                                           \
              return -1;                                              \
          }                                                           \
      } while(0)
  ```

  An example usage is `wbCheck(cudaMalloc(...))`.

Plagiarism

Plagiarism will not be tolerated. The first offense will result in the two parties getting a 0 for the machine problem. Second offense results in a 0 for the course.