Single-Pass Scan Project

This project refers to the algorithm presented in the paper “Single-pass Parallel Prefix Scan with Decoupled Look-back” by Merrill and Garland, that essentially implements a scan in one CUDA kernel that reaches about 80% of memcpy performance. Please read the paper first.

I suggest you use the following steps to approach the implementation of the algorithm:

1. adjust the code that you improved during the second assignment (pbb-library) to scan the elements assigned to each CUDA block, independently across blocks. For example get rid of the “virtualization” loop. Denoting with Q (statically known) the number of elements processed by each thread and with B the CUDA-block size, then each block with scan a chunk of Q*B elements from the input array as follows:
    1. collectively copy with the threads in the block the Q*B elements from global to shared to register memory. (The latter is declared as an array of size Q, e.g., `float reg[Q];`)
    2. Each thread sequentially scans its Q elements (residing in register memory), and places its last scanned element in a shared memory buffer.
    3. Perform a block-level scan on the buffer (use the function that contained the bug)
    4. each thread tid grabs its prefix from the shared-memory buffer (at position tid-1 and zero for tid==0), and then it applies it to each of its elements.
    5. Collective copy from register-to-shared-to-global memory. Of course, make sure to insert barrier as dictated by correctness.
2. Once step 1 validates, and obtains performance competitive with memcpy, then you can move on. Next step is to implement a dynamic numbering of blocks. The wrinkle here is that CUDA/OpenCL does not guarantee that block k is allocated before block k+**1.** This may result in deadlock, since if block k is not allocated and block k+1 is allocated execution bandwidth, then block k+1 waits for the result of block k, which may never come, hence deadlock. The solution is to perform a “dynamic numbering” of the CUDA blocks using a counter in global memory via atomic operations. Essentially, you will need to replace throughout the code “blockIdx.x” with the dynamic block index that you have implemented.
3. Implement the simplest version of the algorithm in the paper. The algorithm uses global memory arrays Fs, As and Ps whose lengths equals the number of CUDA blocks, i.e., one element per CUDA block. When passing them as parameters, remember to declared them as volatile. Fs holds the flags, which can be of three kinds: an X denotes nothing is known, an A at some index idx means that the partial result of block idx is available at position idx of the As array, and P denotes that the total prefix up to the corresponding block is available in the Ps array. The simplest version correspond to using only the first thread in the block to compute the

total-prefix up until that block by traversing backwards the X array. As long as you hit A elements you will maintain a running “sum”. If you hit an “X” you will busy wait until it changes. Finally, when you hit a P, you “add” the corresponding element from the Ps array and you are done computing the prefix.

4. The baseline implementation it is not going to be terribly efficient, so you can think of various optimizations, for example,
    1. you use a WARP of threads to read 32 consecutive elements from Fs, As and Ps, and store them to shared memory. Then you can do the lookback sequentially with one thread. (The read from global memory is the expensive one; this requires one memory transaction, so the parallel read with a WARP of threads should achieve good performance)
    2. Other possible implementation is to implement the lookback in shared memory by means of a parallel reduction that is executed by one WARP of threads.
    3. …

The report should

1. explain each individual step based on tidy pseudocode, with emphasis on the algorithm that allows inter-block communication, and the rationale of the various optimizations you have tried.
2. contain a systematic evaluation of the performance, e.g., compare the performance of your implementation with that of CUB. But also show the impact of each of your optimizations (e.g., in step 4 above). The evaluation should report results on datasets of various array lengths and perhaps on different element types, e.g., 8-bit, 16-bit, 32-bit, 64-bit integers.
3. Of course, you should have a decent structure of the report, e.g., an introduction that introduces the problem, motivates the approach (e.g., all the other approaches require at least three accesses to global memory per element, while this one requires only 2), and so on ...