Second Weekly Assignment for the PMPH Course

This is the text of the second weekly assignment for the DIKU course "Programming Massively Parallel Hardware", 2023-2024.

Hand in your solution in the form of a report in text or PDF format, along with the missing code. We hand-in incomplete code in the archive `w2-code-handin.tar.gz`. You are supposed to fill in the missing code such that all the tests are valid, and to report performance results.

Task 6 is optional---i.e., you will not loose points if it is missing---but you will learn a lot if you solve it, albeit it might take some time. It can also be used to replace any of your missing tasks from this assignment. For Task 6 you would also need to create the tests, validation and instrumentation for it.

Please send back the same files under the same structure that was handed in---implement the missing parts directly in the provided files. There are comments in the source files that are supposed to guide you (together with the text of this assignment).

Unziping the handed in archive `w2-code-handin.tar.gz` will create the `w2-code-handin` folder, which contains two folders: `primes-futhark` and `cuda`.

Folder `primes-futhark` contains the futhark source files related to Task 1, prime number computation.

Folder `cuda` contains the cuda sorce files related to the other tasks:

- A `Makefile` that by default compiles and runs all programs, but the built-in validation may fail because some of the implementation is missing.
- Files `hostSkel.cu.h`, `pbbKernels.cu.h`, `testPBB.cu` and `constants.cu.h` contain the implementation of reduce and (segmented) scan inclusive. The implementation is generic---one may change the associative binary operator and type; take a look---and allows one to fuse the result of a `map` operation with the reduce/scan.
- Files `spMV-Mul-kernels.cu.h` and `spMV-Mul-main.cu` contain the incomplete implementation of the sparse-matrix vector multiplication.

When implementing the CUDA tasks, please take a good look around you to see where and how those functions are used in the implementation of `reduce` and/or `scan`.

Task 1: Flat Implementation of Prime-Numbers Computation in Futhark (3 pts)

This task refers to flattening the nested-parallel version of prime-number computation, which computes all the prime numbers less than or equal to `n` with `D(n) = O(lg lg n)`. More detail can be found in lecture notes, section 4.3.2 entitled "Exercise: Flattening Prime Number Computation (Sieve)".

Please also read section 3.2.5 from lecture notes entitled "Prime Number Computation (Sieve)" which describes two versions: one flat parallel one, but which has sub-optimal depth, and the nested-parallel one that you are supposed to flatten. These versions are fully implemented in files

`primes-naive.fut` and `primes-seq.fut`, respectively. The latter corresponds morally to the nested-parallel version, except that it has been implemented with (sequential) loops, because Futhark does not support irregular parallelism.

Your task is to:

- fill in the missing part of the implementation in file `primes-flat.fut`;
- make sure that `futhark test --backend=opencl primes-flat.fut` succeeds;
- compile with `futhark c` and measure runtime for all three files; `primes-naive.fut` and `primes-seq.fut` are fully implemented. Also compile with `futhark opencl` files `primes-naive.fut` and `primes-flat.fut` and measure runtime (but **not** the `primes-seq` because it has no parallelism and it will crawl). Make sure to run with a big-enough dataset; for example computing the prime numbers less than or equal to `10000000` can be run with the command: `$ echo "10000000" | ./primes-flat -t /dev/stderr -r 10 > /dev/null`.
- Report the runtimes and try to briefly explain them in the report. Do they match what you would expect from their work-depth complexity?
- Please present in your report the code that you have added and briefly explain it.

Task 2: Copying from/to Global to/from Shared Memory in Coalesced Fashion (2pt)

This task refers to improving the spatial locality of global-memory accesses.

On NVIDIA GPUs, threads are divided into warps, in which a warp contains `32` consecutive threads. The threads in the same warp execute in lockstep the same SIMD instruction.

"Coalesced" access to global memory means that the threads in a warp will access (read/write) in the same SIMD instruction consecutive global-memory locations. This coalesced access is optimized by hardware and requires only one (or two) memory transaction(s) to complete the corresponding load/store instruction for all threads in the same warp. Otherwise, as many as `32` different memory transactions may be executed sequentially, which results in a significant overhead.

Your task is to modify in the implementation of `copyFromGlb2ShrMem` and `copyFromShr2GlbMem` functions in file `pbbKernels.cu.h` the (one) line that computes `loc_ind`---i.e.,`uint32_t loc_ind = threadIdx.x * CHUNK + i;`---such that the new implementation is semantically equivalent to the existent one, but it features coalesced access (read/write) to global memory. (The provided implementation exhibits uncoalesced access; read more in the comments associated with function `copyFromGlb2ShrMem`.)

Please write in your report:

- your one-line replacement;
- briefly explain why your replacement ensures coalesced access to global memory;
- explain to what extent your one-line replacement has affected the performance, i.e., which tests and by what factor.

I suggest to do the comparison after you also implement Task 3. You may keep both new and old implementations around for both Tasks 2 and 3 and select between them statically by `#if 1 #then ⋯ #else ⋯ #endif`. Then you can reason separately what is the impact of Task 2 optimization and of Task 3 optimization.

Task 3: Implement Inclusive Scan at WARP Level (2 pts)

This task refers to implementing an efficient WARP-level scan in function `scanIncWarp` of file `pbbKernels.cu.h`, i.e., any WARP of threads scans, independently of each other, its `32` consecutive elements stored in shared memory. The provided (dummy) implementation works correctly, but it is very slow because the warp reduction is performed sequentially by the first thread of each warp, so it takes `WARP-1 == 31` steps to complete, while the other `31` threads of the WARP are idle.

Your task is to re-write the warp-level scan implementation in which the threads in the same WARP cooperate such that the depth of your implementation is 5 steps (WARP==32, and lg(32)=5). The algorithm that you need to implement, together with some instructions is shown in document `Lab2-RedScan.pdf`---the slide just before the last one. The implementation does not need any synchronization, i.e., please do NOT use `__syncthreads();` and the like in there (it would break the whole thing!).

Please write in your report:

- the full code of your implementation of `scanIncWarp` (should not be longer than 20 lines)
- explain the performance impact of your implementation: which tests were affected and by what factor. Does the impact becomes higher for smaller array lengths?

(I suggest you correctly solve Task 2 before measuring the impact.)

Task 4: Find the bug in `scanIncBlock` (1 pts)

There is a nasty race-condition bug in function `scanIncBlock` of file `pbbKernels.cu.h` which appears only for CUDA blocks of size 1024. For example running from terminal with command `./test-pbb 100000 1024` should manifest it. (Or set 1024 as the second argument of `test-pbb` in `Makefile`.)

Can you find the bug? This will help you to understand how to scan CUDA-block elements with a CUDA-block of threads by piggy-backing on the implementation of the warp-level scan that is the subject of Task 3. It will also shed insight in GPU synchronization issues.

- Please explain in the report the nature of the bug, why does it appear only for block size 1024, and how did you fix it.

Task 5: Flat Sparse-Matrix Vector Multiplication in CUDA (2 pts)

This task refers to writing a flat-parallel version of sparse-matrix vector multiplication in CUDA. Take a look at Section 3.2.4 `Sparse-Matrix Vector Multiplication'' in lecture notes, page 40-41 and at section 4.3.1` Exercise: Flattening Sparse-Matrix Vector Multiplication".

Your task is to:

- implement the four kernels of file `spMV-Mul-kernels.cu.h` and two lines in file `spMV-Mul-main.cu` (at lines 154-155).
- run the program and make sure it validates.
- add your implementation in the report (it is short enough) and report speedup/slowdown vs sequential CPU.

Task 6: Partition2 implementation (Optional, Challange, can replace any missing task)

This task refers to implementing the `partition2` parallel operator, whose Futhark implementation is given below:

```
let partition2 [n] 't (p: (t -> bool)) (arr: [n]t) : ([n]t , i32) =
  let cs  = map p arr                             // First scan o map
  let tfs = map (\ f -> if f then 1 else 0) cs    // First scan o map
  let ffs = map (\ f -> if f then 0 else 1) cs    // First scan o map
  let isF = scan (+) 0 ffs                        // First scan o map
  let isT = scan (+) 0 tfs                        // First scan o map
  // (isT, isF) = unzip <| scan (\(a1,b1) (a2,b2) -> (a1+a2, b1+b2)) <| zip tfs ffs
  let i   = isT[n-1]                                 // Second kernel
  let isF'= map (+i) isF                             // Second kernel
  let inds= map3(\c iT iF->if c then iT-1 else iF-1) // Second kernel
                cs isT isF'
  let r = scatter (scratch n t) inds arr             // Second kernel
  in (r, i)
```

No code is provided for this, you are supposed to provide the full implementation:

- the second kernel in file `pbbKernels.cu.h`
- the generic host wrapper in file `hostSkel.cu.h`
- a use case together with validation and performance instrumentation in file `testPBB.cu`. Please make sure that `make run` and `make run-pbb` runs your implementation and displays useful information at the end of what it is currently provided.

Please note that the three maps and the two scans can be fused into one and implemented by

means of the provided `scanInc` function in file `hostSkel.cu.h`, which supports a `scan o map` composition.

For this you will need to define a datatype and specialized operator, see for example `MyInt4` and `MSSP` or even better, `ValFlg` and `LiftOP` in `pbbKernels.cu.h` .

Similarly, the computation after the scan can be fused into one CUDA kernel that you will have to write yourselves.

If you have problem with C++ templates, then you may write directly specialized code that applies to an array of `uint32_t` and the predicate is `even`. Otherwise, instantiate your generic code for the same example.

Describe in your report:

- whether your code validates against the result of your sequential implementation.
- the manner in which you have implemented the `scan o map` composition, i.e., show the datatypes and your host-wrapper function that uses `scanInc` underneath.
- the code of the second kernel.
- the performance of your implementation, i.e., how many GB/sec are achieved if you consider the number of accesses to be `3 * N`, where `N` denotes the length of the input array. Also the speed-up in comparison to a golden-sequential CPU implementation.