Bfast: Translating Futhark Code Efficiently in CUDA

Preamble

This project refers to writing an efficient CUDA implementation for a variation of the BFAST algorithm, which is one of the state-of-the-art methods for break detection given satellite image time series. For example BFAST can be used to detect environmental changes, such as deforestization in the Amazonian jungle, etc.

The attached paper "Massively-Parallel Change Detection for Satellite Time Series Data with Missing Values" (`bfast-paper.pdf`) presents the problem and solution in detail.

In essence the paper addresses the case in which data may be missing, for example because of clouds obscuring the image of the ground.

The attached Futhark implementation fixes the cloud issue by semantically "filtering" out the clouded pixels—encoded by NAN values–in the implementation of various (mathematical) operations. Note that filtering does not necessarily mean in this context the application of the `filter` operator, but rather it means that the iteration space is padded and `NAN` values are ignored. For example, the Futhark implementation of dotproduct is:

```
let dotprod [n] (xs: [n]f32) (ys: [n]f32): f32 =
    reduce (+) 0.0 (map2 (*) xs ys)
```

and the filtered implementation receives an additional flag array in which the invalid pixels, marked with the flag-value `false`, are ignored:

```
let dotprod_filt [n] (flgs: [n]bool) (xs: [n]f32) (ys: [n]f32) : f32 =
    f32.sum (map3 (\ flg x y -> if flg then x*y else 0) flgs xs ys)
```

1 Tasks

Your task is to write a CUDA program, which is semantically equivalent with the Futhark one, and which is as efficient as possible:

- Please provide built-in validation for the provided Sahara or Peru datasets—files `sahara.in.gz`, `sahara.out.gz` `peru.in.gz` and `peru.out.gz`. Note that we have observed that the algorithm is numerically unstable, so the CPU result might differ from the GPU result, e.g., the Peru result on GPU might differ than from the one on CPU (`sahara` should be fine). As such, I suggest you also write a sequential C version executed on the CPU and validate your GPU results according to that. When you make your validation, you would probably like to:
  (a) use a permissive formula for the individual values of CPU and GPU result arrays, such as `fabs(val_cpu - val_gpu)/val_cpu < epsilon`, for some reasonable epsilon.
  (b) report the percentage of values that do not validate under (a); it should hopefully be a small number.
- Parsing the input: Remember that you can write a simple Futhark program that just returns the input arrays and their sizes; if you run it *without* the `-b` option it will write the output in text form. (If you use some simple script/tool that removes the `i32` and `f32` terminations everywhere, do it.) You can then write a simple C function that parses the textual representation of the previous step. If you have problems parsing NaN values, you can modified the above mentioned Futhark program such as all NaN values are transformed

to some big number, e.g., `-100000000`, parse it and then either transform it back or work with that as NaN in your Cuda implementation.

- Identify the opportunities for optimizing locality of reference, and apply related optimizations whenever is possible and profitable to do so, for example:

  1 make sure that all accesses to global memory are coalesced (i.e., spatial locality),

  2 identify the opportunities for one/two/three dimensional tiling in shared memory and apply them whenever profitable;

  3 identify opportunities in which a kernel loop can be executed entirely (or mostly) in shared memory, for example in the case of matrix inversion (e.g., the `gauss_jordan` Futhark function).

  4 identify the cases in which a composition of `scan`, `filter`, `map` or `scatter` operations can be performed in (fast) shared memory. For example, if the size of the scanned segment is less than 1024, it is much faster to perform the `scan` within each block, than to perform a segmented scan across all elements in all blocks (why?). You can use block-level scan, etc., from the library handed in for weekly assignment 2.

- File `bfast-irreg.fut` already hints at a way in which parallel operations should be grouped into kernels. A detailed description of this, together with various optimizations is given in the paper.

- Try to report the impact of your optimizations both locally and globally.

  – Locally means to separately test the impact of optimizations: for example what is the speedup of tiling matrix-vector multiplication, with respect to its untiled version;

  – Globally means to test the speedup generated by an optimization (e.g., tiling) at the level of the whole application runtime.

- Also please report your global performance, for example in terms of percentage of the peak bandwidth of the system (and perhaps also in terms of the sequential program — could be the one generated by futhark-c).

2 Closing Remarks

- Please write a tidy report that puts emphasis on the parallelization and optimization strategy reasoning, and which explores and explains the design space of potential solutions, and argues the strengths and shortcomings of the one you have chosen. For example present on representative example how optimizations such as tiling were applied, and why their application is safe.

- Please devise a validation method (somehow), albeit it can be challenging (validate at least the Sahara dataset), and please compare the performance of your code against baseline implementations in Futhark (or from elsewhere if you have other), and comment on the potential differences between the observed performance and what you assumed it will be (based on the design-space exploration).