Trinomial Princing: Flattening in Futhark and CUDA

Preamble

The handed out code for the project consists of several files, such as `header32.fut`, `header64.fut`, and `trinom-basic.fut` and `trinom-flat.fut`. Currently `trinom-basic.fut` can only be executed with the `futhark-c` compiler due to a internal-compiler bug; we are working on it. The headers are used as macros so that the `real` type can be easily switched from single (`f32`) to double-precision floats `f64` by (un)commenting in the main file `trinom-basic.fut` the lines of code:

```
--default(f64)
--import "header64"

default(f32)
import "header32"
```

The code comes with two datasets, available in `data/small.in` and `data/options-60000.in`; you can use `$ futhark-test -c trinom-basic.fut` for automatic validation due to the openning comment directives

```
-- Trinomial option pricing
-- ==
-- compiled input @ data/small.in
-- output @ data/small.out
--
-- compiled input @ data/options-60000.in
-- output @ data/options-60000.out
```

When we finally fix the compiler and you are able to run the parallel code you may be pleasently surprised at the speedup; if memory serves it is about 300× faster than sequential execution on `gpu02-diku-apl`.

1 The Code Structure of `trinom-basic.fut`

The last line of file `trinom-basic.fut` is actually where you should start—`map (trinomialSingle h_YieldCurve) options`—mainly, you would like to price a bunch of options. One option pricing is implemented by function `trinomialSimgle` which receives as arguments a (constant) array of `YieldCurveData` and an option; its signature is:

```
let trinomialSingle [ycCount] (h_YieldCurve : [ycCount]YieldCurveData)
                              (optionData   : TOptionData)
                            : real = ...
```

where, `YieldCurveData` and `TOptionData` are record types declared earlier in the file:

```
type YieldCurveData = { P : real, t : real }
type TOptionData = {
    StrikePrice   : real
  , Maturity      : real
  , NumberOfTerms : i32
  , ReversionRateParameter : real
  , VolatilityParameter    : real
}
```

The `trinomialSingle` function mainly consists of two sequential (convergence) loops of count `n`, which contain inner parallel operators of length `Qlen`, where `n` and `Qlen` are specific to each option (and thus vary across options). Figure 1 shows the (simplified) structure of `trinomialSingle`.

```
let trinomialSingle [ycCount] (h_YieldCurve : [ycCount]YieldCurveData)
                              (optionData : TOptionData) : real =

    let (Qlen,n) = depends-on(optionData)
    let alphas = replicate (n + 1) zero  -- alphas : [n+1]real
    let alphas[0] = #P (h_YieldCurve[0])
    let Q = map (\i -> if i == m then one else zero) (iota Qlen)

    -- forward propagation (convergence) loop
    let (_,alphas) =
        loop (Q:*[Qlen]real, alphas) for i < n do
            let Q         = map (\j -> ...) (iota Qlen)
            let tmps      = map (\j -> ...) (iota Qlen)
            let alpha_val = reduce (+) zero tmps
            let alphas[i+1] = alpha_val
            in  (Q,alphas)

    let Call = map (\j -> ...) (iota Qlen)

    -- backward propagation (convergence) loop
    let Call =
        loop (Call:*[Qlen]real) for ii < n do
            let i = n - 1 - ii
            -- ...
            in map (\j -> ...) (iota Qlen)

    in Call[m]
```

Figure 1: The Code Structure of Function `trinomialSingle`

Note that array `alphas` has option-dependent size `n+1` and hence Futhark cannot hoist its (memory-expanded) allocation outside of the outermost `map` across all options, which prevents parallelization in the general case. Similar considerations apply for arrays `Q`, `tmps` and `Call`, which have option-dependent size `Qlen`. Furthermore, even if we overcome these obstacles, a parallelization strategy that executes one option per thread—i.e., which would sequentialize all the parallel operators in `trinomialSingle` and only utilize the parallelism of the outer map—would potentially suffer from significant load imbalance, (e.g., thread divergence), because, not only the inner-loop operations have work proportional with (the option variant) `Qlen`, but the loop count itself `n` is variant as well.

Currently, this Futhark implementation uses a very dirty hack to measure at least the *ideal* speedup that can be obtained when you execute one option per thread, namely, it sets `WITH_ONLY_STRIKE_VARIANT=true`, and overwrites the dataset in such a way that the compiler can deduce that all options have the same `Qlen` and `n`—see the code in the `main` function:

```
let (maturities, numofterms, rrps, vols) =
    if WITH_ONLY_STRIKE_VARIANT
    then ( replicate q (maturities0[0]), replicate q (numofterms0[0])
```

```
          , replicate q (rrps0[0]), replicate q (vols0[0])            )
      else ( maturities0, numofterms0, rrps0, vols0)
```

Even so, the code would still not execute as desired (i.e., one option per thread), because the Futhark compiler will recognize and exploit an opportunity for increasing the degree of utilized parallelism by

- inlining the body of function `trinomialSingle` inside the outermost `map`, then by
- distributing the outermost `map` across the two loops and the remaining code, then by
- interchaning the outermost `map` inside the sequential loops and continuing distribution.

In order to really force the compiler to execute one option per thread, the constant variable `let FORCE_PER_OPTION_THREAD` is set to `true` and a branch is added to encompass the body of `trinomialSingle` in order to prevent the distribution of the `map` across the bindings (statements) of the body.

```
let trinomialSingle [ycCount]
                    (h_YieldCurve : [ycCount]YieldCurveData)
                    (optionData : TOptionData) : real = unsafe
  let X  = #StrikePrice   optionData
  -- ...
  let m  = jmax + 2

  in if FORCE_PER_OPTION_THREAD && X < zero
     then zero else
  -- ...
```

All this would allow one to get an idea of the "ideal" runtime of the one-option-per-thread strategy, but, in the best case this implementation is a subcase of a bigger implementation—for example you can write code that test whether the options have comparable `n` and `Qlen`, and if so you can use a version of code similar to this one (that assumes padding), but if `n` and `Qlen` are very different across options, then you need to plug in another implementation for that case.

This observation motivates your project work, which, in simple words referrs to flattening all available parallelism in the general (irregular) case, when `Qlen` varies across options. Your version of code should use only flat-parallel (rather than nested) operators, which have the parallelism degree equal to $\sum_{i=0}^{p-1} \texttt{Qlen}_i$, where p is the number of options. I recommand that both groups read the next section.

2 Project 1: Futhark-to-Futhark Flattening

Your task is to implement the function `trinomialChunk`, whose signature is given below, such that it contains flat parallelism of maximal degree:

```
let trinomialChunk [ycCount] [numAllOptions] [maxOptionsInChunk]
                   (h_YieldCurve : [ycCount]YieldCurveData)
                   (options : [numAllOptions]TOptionData)
                   (w: i32)
                   (n_max : i32)
                   (optionsInChunk: i32, optionIndices: [maxOptionsInChunk]i32)
                 : [maxOptionsInChunk]real = unsafe
```

The function `trinomialChunk` receives as parameters:

- the `h_YieldCurve` as before,
- an array containing all `options`,
- two integers `w` and `n_max`,
- the number of options `optionsInChunk` to be processed by an invocation of `trinomialChunk`, tupled with the indexes of these options, i.e., $\cup_{k \in \texttt{optionIndices}} \texttt{options[k]}$.

The implicit assumptions for having a safe call to `trinomialChunk` are that:

- the sum of `Qlen` across all options in the chunk is less than or equal to `w`, i.e., $\sum_{k \in \texttt{optionIndices}} \texttt{Qlen}_k \leq \texttt{w}$
- the maximum value of `n` across all options in the chunk is less than or equal to `n_max`.

The motivation for implementing this function, is that it would allow to easily explore various optimization strategies. For example, if the maximal `Qlen` across all options is less than a suitable CUDA-block size value `w`, then options could be bin-packed in parallel (e.g., see `bin-packing-ffh.fut`) into chunks (bins) of size (weight) `w` and then each CUDA-block would execute a chunk (bin). Since the sum of `Qlen`s in a chunk is less than `w`, it follows that all (expanded) inner arrays such as `Q`, `Call`, could be maintained in shared memory, which would change the program behavior from memory to compute bound (much like in the case of matrix multiplication). Note that option chunking is necessary because an option might have a `Qlen` that is too small to occupy alone a CUDA-block, e.g., empirical evidence shows that CUDA-block size should be at least 96, and over-partitioning improves load balancing.

Conversely, having such a function available, we can easily implement the default implementation that flattens the parallelism across all options and does not utilizes shared memory. The following code demonstrate such a use, where the `depends-on` function is assumed to compute the `Qlen` and `n` for an option:

```
let num_options = length options
let (ns, Qlens) = map depends-on options
let n_max = reduce (i32.max) 0 ns
let w     = reduce (+) 0 Qlens
in  trinomialChunk h_YieldCurve options w n_max (num_options, iota num_options)
```

2.1 Flattening Hints

(1) The flattening reasoning should probably start by observing that, assuming support for arbitrary nested parallelism, `trinomialChunk` can be defined as:

```
let trinomialChunk maxOptionsInChunk h_YieldCurve options
                   w n_max (optionsInChunk, optionIndices) =
    map (\i ->
                let ind    = optionIndices[i]
                let option = options[ind]
                in  trinomialSingle h_YieldCurve option
         ) (iota optionsInChunk)
```

(2) You probably will continue by inlining the body of `trinomialSingle` inside the `map`, and by applying the rules of flattening. However you are encouraged to simplify the implementation by using the values of `w`, `n_max`, `maxOptionsInChunk` for padding. For example, distributing the `map` over the creation points of array `Q`, `Call`, `tmps` (see Figure 1) would result in expanded arrays (of arrays) `Qs`, `Calls`, `tmpss` whose flat size is at most `w`

(and in practice is a good lower approximation of `w`). For simplicity, the size of `alphass` can be approximated to `maxOptionsInChunk * (n_max+1)`, where segment (subarray) i starts at offset i`*(n_max+1)` in the flat representation. Similarly, the sequential loops can have count `n_max`, but you should make sure that out-of-range iterations should not produce modifications to the segments that have semantically finished their loop execution (remember that each option/segment is supposed to run a loop of count `n`, which differs among options/segments). In essence, remember that each CUDA blocks is supposed to run a different `trinomialChunk` invocation, hence the padding described above would simplify inter-block bookkeeping.

(3) Feel free to take inspiration from file `trinom-flat.fut` which implements the flattened version, but make sure that you just take inspiration: meaning, look at it, but then close the file and try to do it yourselves. For example, several arrays created inside loops, such as `Qs`, `Calls`, `tmpss`, share the same flag and segment-index descriptors[1], which can be computed once before the loop. Dito for the `iota2mp1` array, in which each segment stores `iota Qlen`, where `Qlen=2*m+1`. They all should be stored into padded arrays of flat size `w`. The `Qlens` (or `ms` in code) should be similarly stored into a padded array of size `maxOptionsInChunk`, i.e., one `Qlen` per segment. You can use the segment descriptor to access the corresponding `Qlen` of a segment, for example flattening the nested parallelism corresponding to

```
let Qs = map (\k ->
                let m = ...
                map (\i -> if i == m then one else zero) (iota (2*m+1)))
             (iota optionsInChunk)
```

can be achieved by something like:

```
let ms       = map (\i -> ) (iota optionsInChunk)
let iota2mp1 = ...
let Qs = map (ı k -> if i == ms[sgm_inds[k]] then one else zero
             ) iota2mp1 (iota w)
```

meaning that, assuming we have already distributed the `map` across the first two instructions and obtained `ms:[optionsInChunk]i32` and `iota2mp1:[w]i32`, we can flatten the two-map nest as a flat map, in which we can access the correct `m` for each element `k`, by indexing into `ms` with the segment-index descriptor `sgm_inds[k]`, i.e., get me the `m` for the segment of the `k` element of the flat array.

(4) Good luck with it, and do not be shy to come asking for help!

3 Project 2: Futhark-to-CUDA Translation

Your task is to develop a CUDA implementation semantically equivalent to `trinom-flat.fut`, which has been described in the previous section, such that an invocation of `trinomialChunk` is efficiently executed in one CUDA block. By efficiently we mean that most of the arrays used by `trinomialChunk` should be stored in shared memory—e.g., the ones of size `w` and `maxOptionsInChunk`. The notable exception is the array `alphass`, which you may leave in global memory (because it is rarely accessed, i.e., ones per segment for each iteration of the loop). Implementation hints are:

[1] An array that contains three segments of lengths `4`, `2`, and `3` has the flag descriptor `[4,0,0,0,2,0,3,0,0]`, and the segment-index descriptor `[0,0,0,0,1,1,2,2,2]` and

(1) Make sure to put the necessary barrier synchronization between the parallel operations of `trinomialChunk`, otherwise you might get data races between warps. A block-level map is morally translated to CUDA by writing the body of the map in the kernel; and a scatter is morally translated as an in-place update. For segmented scan and reduce, you may use the block-level implementations that were already provided to you (or use CUB – your choice).

(2) There are about 8 (maximum 10) words of shared memory per thread; try to reuse some of the shared-memory buffers (across arrays) such that you fit in this budget—if you use too much shared memory, this would lead to spawning less-than-optimal blocks per multiprocessor, i.e., to hardware underutilization.

(3) We said that array `alphass` should be stored in global memory. Try to optimize the space: compute `maxOptionsInChunk*(n_max+1)` for each CUDA block (`n_max` may differ across blocks), and then perform a scan on the CUDA-block results, denoted `scan_lens`. The total flat length of `aplhass` is the last element `scan_len`. CUDA block number 0 will use the slice `[0...scan_len[0]-1]` of `alphass`, and any other CUDA block numbered *i* will use the slice `[scan_len[i-1]:scan_len[i]]` of `alphass`.

(4) Good luck with it, and do not be shy to come asking for help!

4 Closing Remarks For Both Projects

- Please write a tidy report that puts emphasis on the parallelization-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.

- Please make sure your code validates at least on the provided datasets, 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).