Assignment 1

Programming Massively Parallel Hardware 2015

Martin Jørgensen
University of Copenhagen
Department of Computer Science
`tzk173@alumni.ku.dk`

September 12, 2015

Contents

1 Task 1

Using List-Homomorphism promotions listed below (from [1, Slide 40]) prove the following invariant:

```
(reduce myop e) . (map f)
            ==
(reduce myop e) . (map ( (reduce myop e) . (map f) ) ) . distr_p
```

The promotions are as follows:

1. $(\text{map } f)\ .\ (\text{map } g) \equiv \text{map}(f\ .\ g)$
2. $(\text{map } f)\ .\ (\text{reduce } (++)\ [])\equiv(\text{reduce } (++)\ [])\ .\ (\text{map } (\text{map } f))$
3. $(\text{reduce } \odot\ e_{\odot})\ .\ (\text{reduce } (++)\ []) \equiv (\text{reduce } \odot\ e_{\odot})\ .\ (\text{map } (\text{reduce } \odot\ e_{\odot}))$

```
(reduce myop e) . (map f)

(reduce myop e) . (map f)                  . (reduce (++) []) . distr_p # From ass

(reduce myop e) . (reduce (++) [])         . (map (map f))    . distr_p # By 2

(reduce myop e) . (map (reduce myop e))    . (map (map f))    . distr_p # By 3

(reduce myop e) . (map ((reduce myop e) . (map f)) )          . distr_p # By 1
```

The last line is the form that we wished to create and we have proven that the invariant holds.

2 Task 2

For this task I had to finish the implementation of a function for calculating the "Longest Satisfying Segment". The code written for the solution can be senn in Figure 1. The part I finished is path of the LLS operator. It takes information about two adjoining segments and creates information about the segment resulting from joining the segments together.

```
    connect = p [lastx,firsty]
    newlss  = lssx `max` lssy `max` (if connect then lcsx+lisy else 0)
    newlis  = lisx `max` (if connect then (tlx+lisy) else 0)
    newlcs  = lcsy `max` (if connect then (tly+lcsx) else 0)
    newok   = okx && oky
```

Figure 1: The code written for the second task of the assignment.

Line 1 Check wether the predicate p holds over the edge between the segments, that is: If we have a list of the last element from list x and the first element of list y, does p still hold. The is used to determine how the two lists can be connected.

Line 2 Sets the length of the new longest satisfying segment in the joined list based on which value is larger: The LSS from list x, the LSS fomr list y and if the `connect` value from Line 1 is "true" then the length of the concluding segment of x joined with the length of the initial segment of y.

Line 3 Sets the length of the longest satisfying initial segment for the joined list by selecting the maximum of either: Length of the initial segment of list x or if the `connect` value is true, it will take the total length of x plus the length of the initial segment of y.

Line 4 Sets the length of the longest satisfying concluding segment for the joined list. Uses same method as above, and the length values of longest satisfying concluding segment for y or if `connect` is set, the total length of y plus the length of the concluding segment of x.

Line 5 Determines wether or not the joined segment is ok by doing an "and" on the ok status of x and y.

3 Task 3

For this section I implemented a filter function called `segmSpecialFilter` which takes a predicate as well as an input list of data, and an info list with the segmentation information for the data list. The function then orders the segments based on the predicate result for each element and returns the segments joined into a list and a new segmentation information list. Figure 2 shows my implementation of the function.

The first thing that happens is that the sizes list have all it's zeroes removed, so it is essentially a list of segment lengths. Afterwords a helper function is called with the list of lengths as well as the input list which splits the list into a list of lists which corresponds to the segments. The `splitter` function is recursive and thus not flat, due to time constraints I was unable to figure out a flat and thus more parallisable way to do this. After this a `parFilter` with the predicate is applied to each segment using map, and the result is unzipped into a tuple with two lists: The restulting arrays and resulting segment information lists. These lists are then reduced individually with the concatenation operator and an empty list to form a list with the filtered segments and a list with the segment information.

```haskell
segmSpecialFilter :: (a->Bool) -> [Int] -> [a] -> ([Int],[a])
segmSpecialFilter cond sizes arr = let
    segLengths = filter (\x -> x /= 0) sizes
    segments = splitter segLengths arr
    (resArr, resSizes) = unzip $ map (parFilter cond) segments
    res1 = reduce (++) [] resArr
    res2 = reduce (++) [] resSizes
    in (res2,res1)
    where
        -- Helper function to split list into segments.
        splitter :: [Int] -> [a] -> [[a]]
        splitter [l] xs = let
            (l1, l2) = splitAt l xs
            in [l1]
        splitter (l:ls) xs = let
            (l1, l2) = splitAt l xs
            in l1 : (splitter ls l2)
```

Figure 2: The code written for the third task of the assignment.

4 Task 4

The attached program will map the function $\backslash x->(x/(x-2.3))^3$ unto the array $[1,..,753411]$ both sequanteially and using a CUDA kernel for doing it in parallel. Both runs are timed and the results are compared to verify that the CPU and GPU are agreeing on the result. For all the runs I did the results we're close enough to satisfy the above function.

The timing is done only for the calculation and not copying data to the graphics cards or allocating memory. The verification of the results is done using the following check $\text{abs}(cpu_t - gpu_t) < \epsilon$. All test runs was run on one of the compute machines we we're granted access to as part of the course.

The timing results for different array sizes are shown in Table 1 and shows that the CPU generally are only faster at very small array sizes. According to my measurements already between 100-200 elements, the GPU code runs faster than the sequential CPU code. Furthermore the increase in compute time rises a lot faster for the CPU code compared to the GPU version.

Since the measurements are in microseconds on a time shared machine there was some inaccuracies, the values in Table 1 is the average of running the program 5 times after each other.

Array Size	**CPU Time**	**GPU Time**
100	50	63
200	90	65
300	73	66
400	82	65
500	91	103
600	130	107
700	111	105
800	120	80
900	152	80
1000	143	76
1100	144	77
1300	162	68
1500	184	150
2000	247	77
3000	360	75
5000	482	77
10000	1122	95
15000	1320	80
50000	5657	149
100000	8233	173
150000	12266	244
200000	16357	278
250000	20409	341
300000	24426	356
350000	28524	434
400000	32492	444
500000	42428	578
600000	48733	635
700000	56853	747
753411	61204	806

Table 1: The runtimes reported by the program, measured in microseconds.

References

[1] C. E. Oancea and T. Henriksen. Introduction: Hardware trends and list homomorphism, 2015.