Weekly 3

Christian Påbøl

December 10, 2020

Contents

1 ISPC

For the first part of the weekly assignment, we program with the Intel Implicit SPMD Program Compiler, and use the user guide for reference. To run the assignments and get a runtime comparison, i created a make target which can be run by using `make assignment` from the `src/ispc/` directory.

1.1 Scan

First, we try to implement a parallel scan:

```
export void scan_ispc(uniform float output[],
      uniform float input[],
      uniform int n) {
    foreach(i = 0 ... n){
    // Exclusive scan
float in_ = input[i];
if (programIndex == 0 && i > 0)
    in_ += output[i-1];

float tmp = exclusive_scan_add(in_);
```

```
output[i] = shift(tmp, 1);
if (programIndex == programCount - 1){
    output[i] = tmp + input[i];
}
    }
}
```

This is simple enough, since an inclusive scan is just a shifted exclusive scan. This is also where we encounter the first differences between ispc and c. When calculating the scan, we have to account for $n > programCount$ and carry the previous output along with us. This will be the only task where the C compiler outdoes us, since it automagically vectorizes the sequential for loop and generates parallel machine code.

1.2 Pack

Pack was similarly not too tricky:

```
export uniform int pack_ispc(uniform int output[],
     uniform int input[],
     uniform int n) {
  uniform int m = 0;
  output[0] = input[0];
  foreach (i = 0 ... n) {
    float j = input[i];
    int keep = j != input[i-1];
    int offset = exclusive_scan_add(keep);
    if(!keep){
offset = programCount -1;
    }
    output[m + offset] = j;
    m += reduce_add(keep);
  }
  return m;
}
```

This code is heavily based upon the included `filter.ispc`, except this time, the `keep` bool records whether we have found a new unique integer. This time, we beat the sequential C compiler, and the code runs 1.5-2 times faster than its sequential counterpart.

1.3 Run-length encoding

This is the most tricky problem of the ISPC problems.

```
 1  export uniform int rle_ispc(uniform int output[],
 2       uniform int input[],
 3       uniform int n) {
 4      // While to handle elements
 5      uniform int c = 0; // Counting offset
 6      uniform int m;
 7      uniform int cur;
 8      uniform int encoded = 0; // use this to handle outputting
 9      while(c < n){
10   cur = input[c]; // Current element
11   m = c + 1;
12   // While to handle counting
13   while(m <= (n-programCount)){
14       // Increment count here
15       int element = input[m+programIndex];
16       if(all(element == cur)){ // If all elements match
17   m += programCount;
18       } else {
19   break;
20       }
21   }
22   // Once we break we need to check the remaining in the sequence
23   // A bit hackish count but who cares
24   while(m < n && input[m] == cur){
25       m++;
26   }
27   // Handle writing
28   output[encoded*2] = m - c;
29   output[encoded*2 + 1] = cur;
30   encoded += 1;
31   c = m;
32      }
33      return encoded*2;
34  }
```

To explain this i go through it section by section

```
1  export uniform int rle_ispc(uniform int output[],
2       uniform int input[],
3       uniform int n) {
4      // While to handle elements
5      uniform int c = 0; // Counting offset
6      uniform int m;
7      uniform int cur;
8      uniform int encoded = 0; // use this to handle outputtig
```

This just sets up the values we need to compute

```
 9  while(c < n){
10       cur = input[c]; // Current element
11       m = c + 1;
```

The outer while loop represents the sequence, one iteration per. We start off a sequence by loading the element in, and choosing where to start. Here, we start off by immediately counting the value in `input[c]`. `m` will be used to keep track of where we are in the sequence and the final count.

```
12  // While to handle counting
13  while(m <= (n-programCount)){
14       // Increment count here
15       int element = input[m+programIndex];
16       if(all(element == cur)){ // If all elements match
17    m += programCount;
18       } else {
19    break;
20       }
21  }
```

This snippet loads a vector into memory. If all elements in that vector are the same as the current, then we can just skip ahead to the next vector. This is where the parallelism comes into play. `all(element = cur)` Boils down to a vector compare, and allows us to check 8[1] thus gaining up to 8x speedup. Unfortunately we can't expect all our sequences to be multiples of ProgramCount, therefore we want to check the tail-end of our sequence sequentially.

[1] on my machine a vector is 8 wide

```
22  // Once we break we need to check the remaining in the sequence
23  // A bit hackish count but who cares
24  while(m < n && input[m] == cur){
25      m++;
26  }
```

Quite simply, this loops m till the sequence breaks, and is guaranteed to run less than `programCount` iterations.

```
27   // Handle writing
28   output[encoded*2] = m - c;
29   output[encoded*2 + 1] = cur;
30   encoded += 1;
31   c = m;
32      }
33      return encoded*2;
34  }
```

Finally, knowing where our sequence starts(`c`) and where it ends(`m`) we write it out to memory, and increase our counts, setting the next sequence start to be the end of our current sequence.

2 The Halide part

Unfortunately, i didn't have time to complete the halide part of this assignment.