What does `restrict` really do, or being careful with memory accesses, Part 2

Dmitry Denisenko

restrict keyword

In Part 1, we wrote a two-tap filter kernel show below. If you studied the code, you probably noticed the `restrict` keyword on both the input and the output pointers.

```
kernel void do_sum (global int * restrict a, int n, int c0, int c1,
                    global int * restrict result) {
  int s0 = 0, s1 = 0;
  for (int i=0; i<n; i++) {
    s1 = a[i];
    if (i >= 1)
       result[i-1] = c0 * s0 + c1 * s1;
    s0 = s1;
} }
```

What does `restrict` really do? It tells the compiler that the marked pointer cannot *alias* with other points, i.e. the restricted pointer refers to a memory location that no other pointer does. In other words, you guarantee to the compiler that the kernel will not be called like this:

```
do_sum (array, n, c0, c1, array)
```

or even like this:

```
do_sum (array, n, c0, c1, array-3)
```

restrict keyword

Without restrict, the compiler has to assume that this situation may occur and create more conservative hardware. In this example, to account for possible overlap of memory pointed by a and `result`, the compiler needs to write each new value of `result`, wait for the write to complete, and only then read the next value of a!

You can observe this with the Optimization Report. If you compile the kernel with `restrict` keywords, you get a clean report (located in kernel/kernel.log file), meaning that a new value is computed on every clock cycle. If you now remove the `restrict` keywords, you see this in the report:

```
Successive iterations launched every 327 cycles due to:
<details of memory dependency between a load and a store>
```

327 cycles in this case is the round-trip time to external memory. So not using restrict just cut the performance of your kernel by 327x! Because it's so important, the compiler will issue a compiler warning every time you pass a kernel argument pointer without `restrict`.