Performing Reductions in OpenCL

Recall the OpenCL Memory Model

Here's the Problem We are Trying to Solve

Reduction Takes Place in a Single Work-Group

Like the first.cpp demo program, we are piecewise multiplying two arrays. Unlike the first demo program, we want to then add up all the products and return the sum.

A * B → prods
Σ prods → C

After the array multiplication, we want each work-group to sum the products within that work-group, then return them to the host in an array for final summing.

To do this, we will not put the products into a large global device array, but into a prods[] array that is shared within its work-group.

If we had 8 work-items in a work-group, we would like the threads in each work-group to execute the following instructions . . .

... but in a more general way than writing them all out by hand.
Here's What You Would Change in your Host Program

```c
size_t numWorkGroups = NUM_ELEMENTS / LOCAL_SIZE;

float * hA = new float[NUM_ELEMENTS];
float * hB = new float[NUM_ELEMENTS];
float * hC = new float[numWorkGroups];
size_t abSize = NUM_ELEMENTS * sizeof(float);
size_t cSize = numWorkGroups * sizeof(float);
cl_mem dA = clCreateBuffer(context, CL_MEM_READ_ONLY, abSize, NULL, &status);
cl_mem dB = clCreateBuffer(context, CL_MEM_READ_ONLY, abSize, NULL, &status);
cl_mem dC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, cSize, NULL, &status);
status = clEnqueueWriteBuffer(cmdQueue, dA, CL_FALSE, 0, abSize, hA, 0, NULL, NULL);
status = clEnqueueWriteBuffer(cmdQueue, dB, CL_FALSE, 0, abSize, hB, 0, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "ArrayMultReduce", &status);
status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &dA);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem), &dB);
status = clSetKernelArg(kernel, 2, LOCAL_SIZE * sizeof(float), NULL);
status = clSetKernelArg(kernel, 3, sizeof(cl_mem), &dC);
```

### Reduction Takes Place Within a Single Work-Group
Each work-item is run by a single thread

<table>
<thead>
<tr>
<th>Thread #0:</th>
<th>Thread #2:</th>
<th>Thread #4:</th>
</tr>
</thead>
</table>

```
A * B → prods
Σ prods → C
```

A work-group consisting of numItems work-items can be reduced to a sum in \(\log_2(\text{numItems})\) steps. In this example, numItems=8.

The reduction begins with the individual products in `prods[0] .. prods[7]`.

The final sum will end up in `prods[0]`, which will then be copied into `dC[wgNum]`.

### The Arguments to the Kernel
```
kernel void
ArrayMultReduce( global const float *dA, global const float *dB,
local float *prods, global float *dC )
```

- `int gid = get_global_id(0);` // 0 .. total_array_size-1
- `int numItems = get_local_size(0);` // # work-items per work-group
- `int trnum = get_local_id(0);` // thread (i.e., work-item) number in this work-group
- `int wgNum = get_group_id(0);` // which work-group number this is in

A * B → prods
Σ prods → C

### A Review of Bitmasks

#### Remember Truth Tables?

<table>
<thead>
<tr>
<th>F</th>
<th>F</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>&amp;</td>
<td>=</td>
</tr>
</tbody>
</table>

Or, with Bits:

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>=</td>
</tr>
</tbody>
</table>

Or, with Multiple Bits:

<table>
<thead>
<tr>
<th>000</th>
<th>011</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>&amp;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>000</th>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>=</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>100</th>
<th>011</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>=</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>010</th>
<th>011</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>=</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>110</th>
<th>001</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>=</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>11</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>=</td>
</tr>
</tbody>
</table>
Reduction Takes Place in a Single Work-Group
Each work-item is run by a single thread:

\[
\sum \text{prods} \rightarrow C
\]

And, Finally, in your Host Program

```c
// all threads execute this code simultaneously:
for( int offset = 1; offset < numItems; offset *= 2 )
{
    int mask = 2*offset - 1;
    barrier( CLK_LOCAL_MEM_FENCE );  // wait for all threads to get here
    if(  ( tnum & mask ) == 0 ) // bit-by-bit and'ing tells us which
    { // threads need to do work now
        prods[ tnum ] += prods[ tnum + offset ];
    }
}
```

```c
kernel void ArrayMultReduce( ... )
```