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.

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

This NULL is how you tell OpenCL that this is a local (shared) array, not a global array.

```c
A * B → prods
∑ prods → C
```

### Reduction Takes Place Within a Single Work-Group

Each work-item is run by a single thread.

- Thread #0:
  - prod[0] += prod[1];
  - offset = 4;
  - mask = 7;

- Thread #2:
  - prod[2] += prod[3];
  - offset = 2;
  - mask = 3;

- Thread #4:
  - prod[4] += prod[5];
  - offset = 1;

- Thread #6:
  - prod[6] += prod[7];

A work-group consisting of `numItems` work-items can be reduced to a sum in \(\log_2(numItems)\) 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

```c
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 tnum = 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
    prods[ tnum ] =  dA[ gid ] * dB[ gid ];   // multiply the two arrays together
    // now add them up – come up with one sum per work-group
    // it is a big performance benefit to do it here while “prods” is still available – and is local
    // it would be a performance hit to pass “prods” back to the host then bring it back to the device for reduction
    // local “prods” array – one per work-item
    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 );
}
```

### A Review of Bitmasks

**Truth Tables**

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>F</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Or, with Bits:

- 0 & 0 = 0
- 0 & 1 = 0
- 1 & 0 = 0
- 1 & 1 = 1

Or, with Multiple Bits:

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>001</td>
</tr>
</tbody>
</table>


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

```c
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 ];
        }
}
barrier( CLK_LOCAL_MEM_FENCE );

if( tnum == 0 )
    dC[ wgNum ] = prods[ 0 ];
```

Kernel void
```
ArrayMultReduce( ...) {
    int gid = get_global_id( 0 );
    int numItems = get_local_size( 0 );
    int tnum = get_local_id( 0 ); // thread number
    int wgNum = get_group_id( 0 ); // work-group number
}
```

Anding bits
```
Σ
prods → C
```

Reduction Performance
Work-Group Size = 32

Array Size (MegaNumbers)
GigaNumbers Multiplied and Reduced Per Second

And, Finally, in your Host Program
```
Wait( cmdQueue );
double time0 = omp_get_wtime( );
status = clEnqueueNDRangeKernel( cmdQueue, kernel, 1, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL );
PrintCLError( status, "clEnqueueNDRangeKernel failed: ");
Wait( cmdQueue );
double time1 = omp_get_wtime( );
status = clEnqueueReadBuffer( cmdQueue, dC, CL_TRUE, 0, numWorkGroups*sizeof(float), hC, 0, NULL, NULL );
PrintCLError( status, "clEnqueueReadBuffer failed: ");
Wait( cmdQueue );
float sum = 0.;
for( int i = 0; i < numWorkgroups; i++ )
{
    sum += hC[ i ];
}
```