Performing Reductions in OpenCL

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License
Recall the OpenCL Memory Model

- Kernel
  - Global Memory
  - Constant Memory
  - WorkGroup
    - WorkGroup
      - WorkGroup
        - Shared Memory
          - Work-Item
            - Private Memory
          - Work-Item
            - Private Memory
          - Work-Item
            - Private Memory
Here's the Problem We are Trying to Solve

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.

\[
A \times B \rightarrow \text{prods} \\
\sum \text{prods} \rightarrow C
\]

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.
Reduction Takes Place in a Single Work-Group

numItems = 8;

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

(Thread #0:
prods[ 0 ] += prods[ 1 ];

Thread #2:
prods[ 2 ] += prods[ 3 ];

Thread #4:
prods[ 4 ] += prods[ 5 ];

Thread #6:
prods[ 6 ] += prods[ 7 ];

... 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 );
// local “prods” array is dimensioned the size of each work-group
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.
The Arguments to the Kernel

```c
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 );
```

```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
                                           // 0 .. numItems-1
    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

    A * B → prods
```
**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>
</tr>
</thead>
<tbody>
<tr>
<td>prods[ 0 ] += prods[ 1 ];</td>
<td>prods[ 2 ] += prods[ 3 ];</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread #4:</th>
<th>Thread #6:</th>
</tr>
</thead>
</table>

offset = 1;  
mask = 1;  

offset = 2;  
mask = 3;  

offset = 4;  
mask = 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]`.
A Review of Bitmasks

Remember *Truth Tables*?

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

Or, with Bits:

<table>
<thead>
<tr>
<th></th>
<th>0 &amp; 0</th>
<th>0 &amp; 1</th>
<th>1 &amp; 0</th>
<th>1 &amp; 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Or, with Multiple Bits:

<table>
<thead>
<tr>
<th></th>
<th>000 &amp; 011</th>
<th>001 &amp; 011</th>
<th>010 &amp; 011</th>
<th>011 &amp; 011</th>
<th>100 &amp; 011</th>
<th>101 &amp; 011</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>000</td>
<td>001</td>
</tr>
<tr>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reduction Takes Place in 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>
<th>Thread #6:</th>
</tr>
</thead>
</table>

offset = 1;  
mask = 1;  

offset = 2;  
mask = 3;  

offset = 4;  
mask = 7;  

numItems = 8;

Anding bits

Σ prods → C

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


// 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];
        }
}

barrier( CLK_LOCAL_MEM_FENCE );
if( tnum == 0 )
    dC[wgNum] = prods[0];
And, Finally, in your Host Program

```c
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, "clEnqueueReadBufferI failed: " );
Wait( cmdQueue );

float sum = 0.;
for( int i = 0; i < numWorkgroups; i++ )
{
    sum += hC[ i ];
}
```
Reduction Performance
Work-Group Size = 32

Array Size (MegaNumbers) vs GigaNumbers Multiplied and Reduced Per Second