Performing Reductions in OpenCL

Recall the OpenCL Memory Model

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

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

\[ \text{Thread #0: } \text{prods}[0] += \text{prods}[1] \]
\[ \text{Thread #2: } \text{prods}[2] += \text{prods}[3] \]
\[ \text{Thread #4: } \text{prods}[4] += \text{prods}[5] \]
\[ \text{Thread #6: } \text{prods}[6] += \text{prods}[7] \]

. . . 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 * Nb = 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 = 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);
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

A work-group consisting of numItems work-items can be reduced to a sum in Log2(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 final sum will end up in `prods[0]`, which will then be copied into `dC[wgNum]`.

```c
prods[0] += prods[1];
prods[4] += prods[5];
prods[2] += prods[3];
prods[0] += prods[2];
prods[5] += prods[7];
prods[4] += prods[6];
prods[0] += prods[4];
prods[6] += prods[3];
```

Reduction Takes Place in a Single Work-Group

Each work-item is run by a single thread

Kernels can be Reduce operations within a single work-group.

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

The Arguments to the Kernel

- `global const float *dA, global const float *dB,
- `local float *prods, global float *dC`)

A * B → prods
Σ prods → C
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, "clEnqueueReadBuffer failed: ");
Wait(cmdQueue);

float sum = 0.;
for( int i = 0; i < numWorkgroups; i++ )
{
    sum += hC[i];
}
```

Reduction Performance

Work-Group Size = 32

![Graph showing reduction performance with work-group size 32]