UOW AUSTRALIA

SCIT

School of Computing & Information Technology

CSCI376 – Multicore and GPU Programming

Parallel Reduction

This tutorial examines implementations of the parallel reduction design pattern in OpenCL. Refer to the lecture notes for further details.

Parallel Reduction using Scalars and Vectors

The code in tutorial10.cpp and reduction.cl is related to this section. The example shows an implementation of partial parallel reduction on an array of 2^{17} = 131072 `float`s.

Start by looking at the code in the host application. There are 2 kernel functions; the first uses the `float` scalar data type, whereas the second uses the `float4` vector data type. The program profiles the two methods to compare their computation times.

```
kernel[SCALAR] = cl::Kernel(program, "reduction_scalar");
kernel[VECTOR] = cl::Kernel(program, "reduction_vector");
```

The program aims to invoke as many work-items per work-group as possible, but it needs to ensure that there is sufficient local memory and also what work-group size can run on the device for that kernel. As such, the code queries the maximum work-group size, local memory size supported by the device and allowable work-group size for the kernel on the device:

```
workgroupSize = device.getInfo<CL_DEVICE_MAX_WORK_GROUP_SIZE>();
localMemorySize = device.getInfo<CL_DEVICE_LOCAL_MEM_SIZE>();
kernelWorkgroupSize =
      kernel[SCALAR].getWorkGroupInfo<CL_KERNEL_WORK_GROUP_SIZE>(device);
```

If the allowable kernel work-group size is 1, abort the program because the parallel reduction kernel cannot be done adequately:

```
if (kernelWorkgroupSize == 1)
    quit_program("Abort: Cannot run reduction kernel, because kernel workgroup size
            is 1.");
```

Also, if the kernel work-group size is not 1, but is smaller than the maximum work-group size of the device, set the work-group size to this size:

```
if (workgroupSize > kernelWorkgroupSize)
    workgroupSize = kernelWorkgroupSize;
```

Next, check whether there is enough local memory for that number of work-groups. If necessary, the work-group size is reduced until sufficient local memory is available:

```
while (localMemorySize < sizeof(float) * workgroupSize * 4)
```

```
{
	workgroupSize /= 4;
}
```

Based on this, the total number of work-groups is calculated and memory is allocated to store the partial sums of the reduction process for all the work-groups. Note that for the vector approach, the amount of required memory is divided by 4 since this approach adds the contents of the resulting 4 component vector before returning it as the result of that work-group:

```
numOfGroups = NUM_OF_ELEMENTS / workgroupSize;
scalarSum.resize(numOfGroups);
vectorSum.resize(numOfGroups/4);
```

The appropriate buffer objects are created for the input data (read-only), and the output buffers for the scalar and vector approaches. The program initialises the output buffers contents to all 0s. This is just a pre-caution, because in this example it is acceptable to just reserve the appropriate amount of memory without initialising it, since the results will be written to the output buffers in the kernel anyway.

```
for (int i = 0; i < numOfGroups; i++)
{
	scalarSum[i] = 0.0f;
}
for (int i = 0; i < numOfGroups/4; i++)
{
	vectorSum[i] = 0.0f;
}

dataBuffer = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
	sizeof(cl_float) * NUM_OF_ELEMENTS, &data[0]);
scalarBuffer = cl::Buffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
	sizeof(cl_float) * numOfGroups, &scalarSum[0]);
vectorBuffer = cl::Buffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
	sizeof(cl_float) * numOfGroups/4, &vectorSum[0]);
```

The kernel arguments are set for either the scalar reduction kernel or the vector reduction kernel. The second argument reserves local memory, which the kernel function will use to store initial data read from global memory and subsequently used to store the partial sums. Note that the amount of local memory reserved is different for the scalar and vector versions. Also note that the `globalSize` (total number of work-items) which will be used to enqueue the kernels are also different.

```
cl::NDRange offset(0);
cl::NDRange globalSize(0);
cl::NDRange localSize(workgroupSize);

for (int i = 0; i < 2; i++)
{
	kernel[i].setArg(0, dataBuffer);
```

For scalar reduction:

```
	if (i == SCALAR)
	{
		localSpace = cl::Local(sizeof(float) * workgroupSize);
		kernel[i].setArg(1, localSpace);
```

```
        kernel[i].setArg(2, scalarBuffer);
        globalSize = NUM_OF_ELEMENTS;
    }
```

For vector reduction:

```
    else
    {
        localSpace = cl::Local(sizeof(float) * workgroupSize * 4);
        kernel[i].setArg(1, localSpace);
        kernel[i].setArg(2, vectorBuffer);
        globalSize = NUM_OF_ELEMENTS/4;
    }
```

The kernel is enqueued with the appropriate global size (total number of work-items), local size (number of work-items per work-group), and an event which will be used to obtain timing information:

```
queue.enqueueNDRangeKernel(kernel[i], offset, globalSize, localSize, NULL,
    &profileEvent);
```

The finish function is used to stall the program until all preceding command-queue commands have completed, before the timing information is obtained.

```
queue.finish();

timeStart = profileEvent.getProfilingInfo<CL_PROFILING_COMMAND_START>();
timeEnd = profileEvent.getProfilingInfo<CL_PROFILING_COMMAND_END>();

timeTotal = timeEnd - timeStart;
```

The kernel results are then read to the host application, and the host performs the remainder of the reduction by summing all the partial sums, which were computed the work-groups.

For scalar reduction:

```
queue.enqueueReadBuffer(scalarBuffer, CL_TRUE, 0, sizeof(cl_float) * numOfGroups,
    &scalarSum[0]);

sum = 0.0f;
for (int i = 0; i < numOfGroups; i++)
{
    sum += scalarSum[i];
}
```

For vector reduction:

```
queue.enqueueReadBuffer(vectorBuffer, CL_TRUE, 0, sizeof(cl_float) * numOfGroups / 4,
    &vectorSum[0]);

sum = 0.0f;
for (int i = 0; i < numOfGroups / 4; i++)
{
    sum += vectorSum[i];
}
```

Following this, the results are checked. Note that the result checking caters for rounding errors by giving 0.01 margin for error.

```
if (fabs(sum - correctSum) > 0.01f * fabs(sum))
{
	std::cout << "Check failed." << std::endl;
}
else
{
	std::cout << "Check passed." << std::endl;
}
```

Scalar reduction kernel

Now look at the scalar reduction kernel function. It accepts the appropriate buffer objects that are set in the host application.

```
__kernel void reduction_scalar(__global float* data,
	__local float* partial_sums, __global float* output) {
```

First thing it does is to obtain the work-item's local ID (i.e. ID within the work-group) and the total number of work-items in the work-group.

```
int lid = get_local_id(0);
int group_size = get_local_size(0);
```

Next, it reads the elements that the work-group is supposed to work on from global memory into local memory. This is because local memory (which is shared by all work-items within a work-group) is faster than global memory, so optimised kernels will typically perform computation using local memory for shared data where possible. Note that each work-item gets its respective portion of the data from global memory based on its global ID (a work-item's unique ID within the index space), and stores this in local memory based on its local ID within the work-group.

```
partial_sums[lid] = data[get_global_id(0)];
```

Before doing anything with the contents of local memory, the kernel needs to ensure that all work-items have completed reading from global memory into local memory. Otherwise, computations done on the data may potentially be incorrect if some work-items have not completed storing data in local memory. To do this, a barrier function is used to stall any further operations until **all** work-items in the work-group reach the barrier.

```
barrier(CLK_LOCAL_MEM_FENCE);
```

Once the transfer to local memory is complete, the reduction algorithm can commence. Since the reduction algorithm adds pairs of values, the value of `i` is initially set to half the number of work-items in the work-group. For all subsequent iterations this value is shifted to the right by 1, which effectively divides it by 2 (i.e. halving the number of elements to sum). The iteration continues until there is 1 value left.

```
for(int i = group_size/2; i>0; i >>= 1) {
```

During every iteration, half the remaining values are added with the corresponding half. Note that the `if` statement ensures that only the appropriate work-items will continue performing the partial sums.

```
	if(lid < i) {
	   partial_sums[lid] += partial_sums[lid + i];
```

```
    }
```

Before continuing, a barrier is again used to ensure that all work-items complete their operations. This is to ensure that all computations have been completed and stored in local memory before any further accesses to the values in local memory.

```
    barrier(CLK_LOCAL_MEM_FENCE);
  }
```

When the iteration phase completes, the array values that were assigned to that work-group would have been accumulated into a single value that will now be stored by the first work-item of the work-group. The next section of code ensures that only the first work-item will write its contents to global memory as the reduction result for that work-group.

```
  if(lid == 0) {
    output[get_group_id(0)] = partial_sums[0];
  }
}
```

Vector reduction kernel

The vector reduction kernel is very similar to its scalar counterpart. The only difference is that `float4`s are used when passing the input data and for computations using local memory. The output still uses `float`s because the output of a work-group is a single value.

```
__kernel void reduction_vector(__global float4* data,
    __local float4* partial_sums, __global float* output) {
```

The other difference is that before the first work-item in the work-group writes the reduction result to the output buffer, it computes the sum of all the vector's components using a dot product.

```
  if(lid == 0) {
    output[get_group_id(0)] = dot(partial_sums[0], (float4)(1.0f));
  }
```

The profiling results of the program show that the vector version will typically execute much faster than the scalar version. This is because vector operations on the components of a vector are performed in parallel (the number of vector components operated on in parallel is limited by a device's capabilities).

References

Among others, the material in this tutorial was sourced from:

- M. Scarpino, "OpenCL in Action: How to Accelerate Graphics and Computation," Manning Publications