Lincoln School of Computer Science
University of Lincoln
CMP3110M/CMP9057M Parallel Computing

Reductions in OpenCL

The tutorial code is hosted on GitHub https://github.com/gcielniak/OpenCL-Tutorials. You can either download and extract the repository from a zip file (the green "Clone or download" button) or clone to a local directory by issuing 'git clone https://github.com/gcielniak/OpenCL-Tutorials.git' command from the command line. This time we will use Tutorial 3. This new project features a padding functionality which extends the length of an input vector so that it is divisible by the work group size – this makes the code more efficient and easier to write. Refer to the previous tutorial descriptions for more information about the workshops and their structure, and for more technical details on OpenCL.

1 Reduce

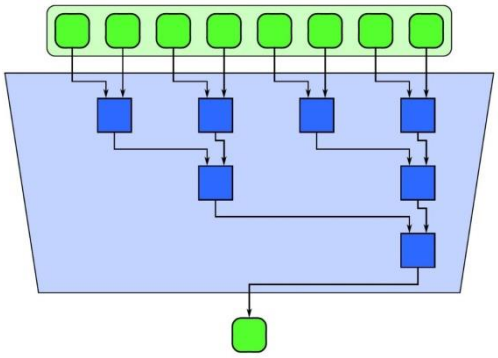

Figure 1The Reduce parallel pattern (after http://parallelbook.com/sites/parallelbook.com/files/SC13_20131117_Intel_McCool_Robison_Reinders_Hebenstreit.pdf).

1.1 Simple reduce on global memory

The reduce pattern combines all elements of the input data into a single output element (see Fig. 1). Let us look at a simple implementation of a reduce kernel performing addition operations which results in a single value representing a total sum of all input elements. We first start with a version which performs reductions within one work group using global memory. In our new project Tutorial 3, `reduce_add_1`implements such a kernel for a fixed number of steps so you can see how each individual reduction is implemented. Note, how a modulo operator is used to choose the right index, how the stride changes and boundary checks are performed. The barriers between each step assure that the global memory operations are completed before the next step of the algorithm commences. This was not necessary in the map or stencil patterns which had no data dependencies. It is important to remember that the barriers only work for work items from the same work group!

Task4U: Run the kernel code. The kernel has a fixed number of steps. What is the maximum number of elements this kernel could reduce?

The next kernel implementation `reduce_add_2` implements the same code but automatically calculates the number of required steps. Note how the FOR loop takes you over each individual step.

Task4U: Run the kernel code and compare the results with those obtained in the previous task. What is the maximum number of elements this kernel could reduce?

1.2 Local memory + large vectors

Operating directly on global memory is slow and affects the performance of parallel algorithms. A common strategy is to use so called local memory (a form of cache) to speed up operations on intermediate results; this is also called a privatisation technique. The kernel `reduce_add_3` demonstrates how to use local memory to store partial sums resulting from each reduction step. The procedure first copies the data from global to local memory, calculates all reduction steps on local memory and then copies the result to the output variable. Since we read the global memory only once at the beginning the barriers between the steps refer now to local memory. This is a much faster operation. The size of local memory can be defined in the host code and is specified as an additional kernel parameter (you can simply uncomment a corresponding code line). Note the size of local memory – it corresponds to the number of work items in a workgroup.

Task4U: Run the code and check if the results are the same as for the global memory version.

To perform reduce on larger data sets we will need to combine the results from individual work groups/tiles. There are many implementation strategies to achieve that but we are going to use a simple one based on atomic functions. In kernel `reduce_add_4`, at the end of each work group reduction we simply copy the resulting reduction from local memory to the first component of the output vector. This part of the process is performed sequentially which might be a bottleneck in situations when many work groups are executed at the same time.

Task4U: Run the kernel and check the results again. You may notice now that we are using only a single element in the output vector (i.e. buffer B). You can therefore adjust its original length to 1 – this way there will be more memory available on a device for the input vector.

Task4U: Profile the kernel for different group size using three different input vector lengths (e.g. 100K, 1M, 10M) and on different devices. As a workgroup size use either powers of two or identify the suggested value by using the `CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE` property. Keep all input values as ones so that their sum is always equal to the length of the input vector – this will make it easier to see if the results are correct. What is the optimal work group size for GPU and CPU devices?

*** Task4U: Implement a reduction kernel using sequential addressing as presented during the lectures. Compare its performance to the interleaved addressing kernel (i.e. `reduce_add_4`) on a large problem (e.g. 10M elements). How significant is the difference?*

2 Scatter

The scatter pattern writes data into output locations indicated by an index array (see Fig. 2). This operation is similar to the gather pattern which used an index array for looking at the input location.

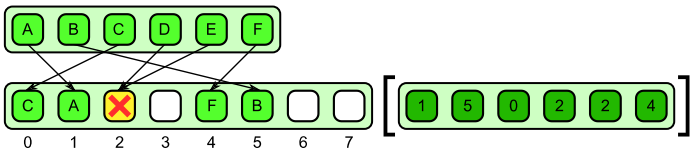

Figure 2TheScatter parallel pattern (after http://parallelbook.com/sites/parallelbook.com/files/SC13_20131117_Intel_McCool_Robison_Reinders_Hebenstreit.pdf).

2.1 Histogram

One of the simplest examples of the scatter pattern is a histogram. The histogram uses a set of value ranges (i.e. bins) and counts the number of input values falling into each bin.

Let us use a simple implementation of a histogram operating on natural numbers with the bin width equal to one. Each histogram bin will count a number of occurrences of a particular natural number in the input vector. The corresponding kernel code is implemented in `hist_simple` and you should adapt the host code to run it. Set the number of bins to 10 (i.e. the length of bufferB) and populate the input vector (i.e. A) with numbers ranging from 0 to 9. Do not forget to set the `local_size` parameter to the input length. Run the kernel and inspect the results.

Task4U: Currently, the histogram calculation will fail if a given input number is greater than 9. Add an additional parameter to the histogram kernel called `nr_bins` and use it to detect such numbers. You can either ignore these values or alternatively add them to the last bin.

Task4U: Larger input vectors would require padding to enable their work with an arbitrary work group size. What should be the value of a neutral element in the histogram calculation and how would you modify your code to accommodate these elements such that their count is not included in the final histogram? You can specify the value of that neutral element as an additional parameter to your kernel.

Task4U: Modify the histogram kernel to accommodate integer numbers which include also negative numbers. Add an additional parameter called `min_value` which will define the first bin of the histogram. You can treat the numbers falling outside the range of your histogram similarly as in the previous task.

Task4U: Modify the histogram kernel to accommodate variable bin widths. Add an additional parameter called `max_value` which will define the last bin of the histogram.

*Task4U: **Write a histogram kernel function which will take the number of bins as a parameter and calculate the min and max values automatically (you will need to use the reduction kernel to calculate min and max).*

3 Scan

The scan parallel pattern is similar to the reduction pattern but it also keeps all partial results (see Fig. 3).

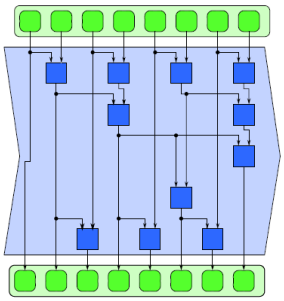

Figure 3 The Scan parallel pattern (after http://parallelbook.com/sites/parallelbook.com/files/SC13_20131117_Intel_McCool_Robison_Reinders_Hebenstreit.pdf).

This pattern can be applied to solve many seemingly sequential problems such as sorting, search, merging, etc. We will concentrate mainly on a general idea behind parallel scan calculation in this task. The kernel function `scan_add` contains an implementation of a span efficient inclusive scan by Hillis and Steele (1986). This implementation requires two local memory buffers to work which are swapped after each reduction step to avoid data overwriting. Before you run the scan kernel, you will need to add an additional kernel parameter in the host code which defines the size of the second buffer – its size should be exactly the same as the first one.

Task4U: Run the scan kernel on a small input vector consisting of ones only so the results are easier to interpret. Do not forget to set the work group size so it matches the input size. Inspect the results. Now, half the size of the work group and compare the obtained results.

The provided kernel scan allows only for calculating partial reductions in a single work group separately. Blelloch (1991) describes a method for extending the basic scan to enable a full scan operation on large vectors (see Fig. 4). The following steps are required:

1. Calculate all local/block scans and store the partial results in the output vector.
2. Create a separate buffer to store block sums i.e. the last elements from each local scan.
3. Run the scan operator on the block sums.
4. Add the resulting cumulative sums block by block to the resulting vector.

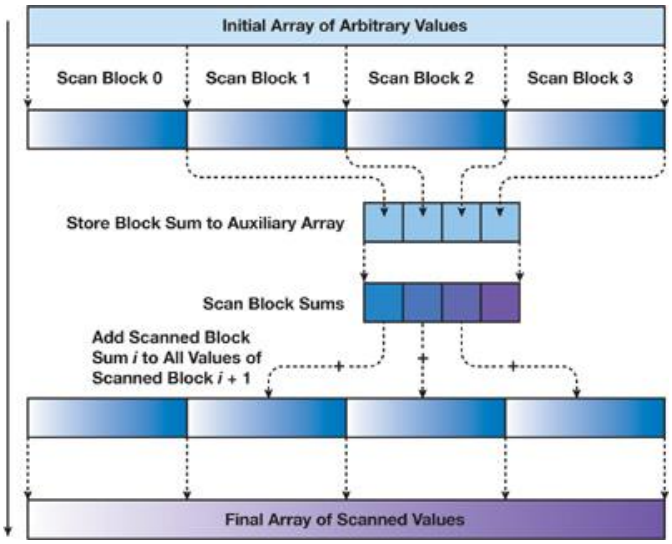

Figure 4 The scan operation for large vectors (Blelloch, 1991).

The tutorial code contains four device kernels which can be used to implement that procedure:

- `scan_add`: performs local scans on the input vector.
- `block_sum`: extracts the block sums from the vector containing local scans and stores them in a separate array; the array length is equal to the number of work groups. This kernel requires an additional parameter which defines the work group size.
- `scan_add_atomic`: performs an exclusive scan on the block sums vector and store them in a separate buffer to avoid overwriting. This additional vector should have the same length as the block sums vector and should be set to 0 before use.
- `scan_adjust`: will add the scan results from the previous step to the output vector.

Task4U: Your task is to modify the host code in such a way that it can run the four kernels listed above. You will need to create some additional buffers and set the right parameters for each kernel. The easiest way to do it is by introducing a single step at once whilst checking the results from each procedure. When completed try the scan procedure on a large vector with thousands of elements to check its correctness.

If you have finished all the tasks in this tutorial you can start working on your assignment.

References

W. Daniel Hillis and Guy L. Steele, Jr.. Data parallel algorithms. Commun. ACM 29, 12 (December 1986), 1170-1183.

Guy E. Blelloch. Prefix Sums and Their Applications. In "Synthesis of Parallel Algorithms", Edited by John H. Reif, Morgan Kaufmann, 1991.