Lincoln School of Computer Science
University of Lincoln
CMP3752M/CMP9773M, Parallel Programming

Parallel Libraries

In this final workshop session, we are going to learn how to use a parallel library called *Boost.Compute*. The tutorial code is hosted on GitHub. 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 4. Refer to the previous tutorial descriptions for more information about the workshops and their structure, and for more technical details on OpenCL.

1 Boost.Compute

Boost.Compute is a STL-like C++ wrapper around OpenCL functionality which implements many basic algorithms such as transform, reduction, sort, etc. in an efficient and easy to use way (see documentation for more details). It is developed as a part of the Boost library which is a very popular template library amongst the C++ users.

Task4U: Read through the documentation up to "Advanced Topics" to familiarise yourself with the library.

2 Vector Addition

In our first example, we will revisit our vector addition code from the first tutorial. The provided program adds two vectors A and B and stores the result in vector C. Inspect the provided code first, then build and run it. Compare the complexity of this code to the original implementation in OpenCL.

If you are not familiar with STL, certain function calls might look a bit cryptic. It might be a good idea to look at the fundamentals of STL presented in the following tutorial: http://www.tutorialspoint.com/cplusplus/cpp_stl_tutorial.htm.

To understand better the provided code, let us look at the `compute::transform` function. The transform function corresponds to a map parallel pattern and applies the same operator (in our case `plus<mytype>`) to all elements of two input vectors (`devA`, `devB`) and stores the result in the third one (`devC`). As function parameters in our case, we need to provide the first and last element of the first vector (`devA`) to define the range, and the first elements of the second (`devB`) and the third one (`devC`). The length of these last two vectors will be assumed to be the same as the first vector. Vector properties `begin()` and `end()` point to the first and last elements of the vector respectively.

Task4U: You can easily change the operator provided to the transform function. Check out the documentation first and then try subtraction and multiplication operations.

You can also easily add custom operators which the transform function can apply to the vectors. For example, an operator that will add 2 to all elements can be declared in the following way:

```
BOOST_COMPUTE_FUNCTION(int, plus_two, (int x), {
   return x + 2;
});
```

Such an operator can be then used with the transform function:

```
compute::transform(devA.begin(), devA.end(), devC.begin(), plus_two);
```

As a result, vector C should contain all elements from vector A with added value of 2. Try this new functionality with your code.

Task4U: Write an operator called `square` which will raise each element to the power of 2 and apply it to vector A. Inspect the final results.

Task4U: Repeat above tasks but for the `float` data type.

3 Reduce

The next important function is compute::reduce which corresponds to parallel reduction. The following code demonstrates how to reduce a vector using an addition operation.

```
int sum;
compute::reduce(devA.begin(), devA.end(), &sum);
```

Try this function first and check the results. It is possible to specify different operators for the reduce operator; if none is specified, then `compute::plus` is assumed. The following line will produce exactly the same results as the code above:

```
compute::reduce(devA.begin(), devA.end(), &sum, compute::plus<mytype>());
```

Task4U: Apply min and max operators with the reduce function and inspect the results.

Task4U: Use these functions to calculate the three basic statistics (avg, min, max) on temperature records from Lincolnshire used in the assignment.

4 Scatter & gather

The two parallel patterns of scatter and gather are also present in Boost.Compute as `compute::scatter` and `compute::gather` functions and can be used in a similar fashion as the previous functions. Both functions take as additional input a vector which defines the mapping locations.

Task4U: Apply both functions to the examples presented in lecture slides and compare the results.

5 Scan/Sort/Search

Additional functions provided by the Compute library which were covered in our module include scan (inclusive and exclusive), sort and binary_search. Experiment with these functions and check the results.

Task4U: Implement a histogram function using a Sort-Search algorithm as presented in the lecture slides. Apply this function to temperature records from Lincolnshire used in the assignment.

6 Platforms and Devices

So far in our examples, we have used a default OpenCL device. Boost.Compute allows for selecting both OpenCL platforms and devices and arranging them into different contexts. The following code demonstrates how to create OpenCL context and queue and use them in our basic vector addition example:

```
int main() {
   typedef int mytype;

   // get default device and setup context
   compute::device device = compute::system::default_device();
   compute::context context(device);
   compute::command_queue queue(context, device);

   cout << "Runinng on " << device.name() << endl;

   // create vectors on the host
   vector<mytype> A = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
   vector<mytype> B = { 0, 1, 2, 0, 1, 2, 0, 1, 2, 0 };
   vector<mytype> C(A.size());

   // create vectors on the device
   compute::vector<mytype> devA(A.size(), context);
   compute::vector<mytype> devB(B.size(), context);
   compute::vector<mytype> devC(C.size(), context);

   // copy input data to the device
   compute::copy(A.begin(), A.end(), devA.begin(), queue);
   compute::copy(B.begin(), B.end(), devB.begin(), queue);

   // perform C = A + B
   compute::transform(devA.begin(), devA.end(), devB.begin(), devC.begin(),
compute::plus<mytype>(), queue);

   // copy data back to the host
   compute::copy(devC.begin(), devC.end(), C.begin(), queue);

   cout << "A = " << A << endl;
   cout << "B = " << B << endl;
   cout << "C = " << C << endl;

   return 0;
}
```

Note how the context is used to declare the device buffers and the queue to perform the operations.

The following code demonstrates how to select a specific platform and device:

```
// get all platforms
vector<compute::platform> platforms = compute::system::platforms();
// get all devices for platform 0
vector<compute::device> devices = platforms[0].devices();
// get device 0 for platform 0
compute::device device = devices[0];

cout << "Runinng on " << platforms[0].name() << ", " << device.name() << endl;
```

Task4U: Adapt the main function, so it can accept additional command line parameters allowing for selecting a specific platform and device, similarly to the code from our previous tutorials.

7 Profiling

The performance timing tutorial demonstrates how to profile your code in Boost.Compute.

Task4U: Perform profiling of the reduce code on a large input vector (e.g. 10M elements) and compare its performance to our original OpenCL implementation.