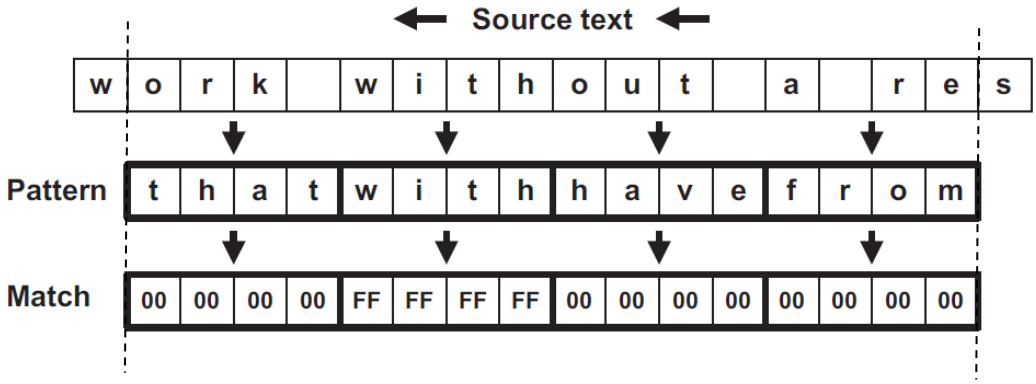
CSCI376 – Multicore and GPU Programming

Parallel String Search

The code in this tutorial shows an example of parallel string searching using vector comparisons.

Please refer to the code in tutorial9.cpp and string_search.cl. The program will search the text for the total number of occurrences of four input strings; “that”, “with”, “have” and “from”, and displays the search results.

The code in the kernel uses vector comparisons. An illustration of the search pattern is as follows:



The four input strings are placed in a `char16` vector (`pattern`), which is compared to another `char16` containing text from a text file (`text_vector`). The `==` operator compares the two vectors and places the result in a third vector (`check_vector`). If two characters match, the corresponding byte in the third vector will be set to 0xFF. If not, the corresponding byte will be set to 0x00. The `all` function can then be used to check whether the most significant bit (MSB) of all components in a vector are set.

In the host application, the search pattern is declared and initialised:

```
cl_char pattern[16] = { 't', 'h', 'a', 't', 'w', 'i', 't', 'h', 'h', 'a', 'v', 'e',
      'f', 'r', 'o', 'm' };
```

The number of occurrences for each of the 4 search strings will be read from the result buffer into:

```
cl_int result[4] = { 0, 0, 0, 0 };
```

The following will be used to distribute chunks of characters from the text to work-items:

```
cl_int charsPerWorkitem = 0, workgroupSize;
```

Find the maximum the number of work-items per workgroup:

```
workgroupSize = device.getInfo<CL_DEVICE_MAX_WORK_GROUP_SIZE>();
```

The input text is read from an input file and the number of characters to be processed per work-item is computed as follows (note the +1 is because the conversion of the division result from `float` to `int` truncates the decimal values):

```
charsPerWorkitem = (text.size() / workgroupSize) + 1;
```

Then the input text array is resized to avoid reading errors in the kernel for the last work-item:

```
text.resize((charsPerWorkitem + 1) * workgroupSize);
```

Buffer objects are created for the input text and results:

```
textBuffer = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
       sizeof(cl_char) * text.size(), &text[0]);

resultBuffer = cl::Buffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
       sizeof(result), &result[0]);
```

Next, determine the amount of local memory space to reserve:

```
localSpace = cl::Local(sizeof(result));
```

The kernel arguments are set as follows (note that the first and third arguments use private memory, and the fourth is used to allocate local memory):

```
kernel.setArg(0, sizeof(pattern), pattern);
kernel.setArg(1, textBuffer);
kernel.setArg(2, charsPerWorkitem);
kernel.setArg(3, localSpace);
kernel.setArg(4, resultBuffer);
```

The kernel accepts the respective arguments (note the private, local and global parameters):

```
__kernel void string_search(char16 pattern,
                            __global char* text,
                            int chars_per_item,
                            __local int* local_result,
                            __global int* global_result) {
```

Two char16 vectors are declared; one will be used to store the text string, the other will store the comparison results:

```
   char16 text_vector, check_vector;
```

Before starting the algorithm, the number of occurrences (in local memory) needs to be initialised to 0. The barrier is to ensure that the initialisation is complete before continuing:

```
   local_result[0] = 0;
   local_result[1] = 0;
```

```
local_result[2] = 0;
local_result[3] = 0;

barrier(CLK_LOCAL_MEM_FENCE);
```

Each work-item needs to operate on its respective chunk of the input text. The following calculates the start of the appropriate section of input text to perform comparisons on:

```
int item_offset = get_global_id(0) * chars_per_item;
```

Each work-item iterates through its respective set of the characters based on `chars_per_item` that was previously computed on the host:

```
for(int i=item_offset; i<item_offset + chars_per_item; i++) {
```

For every iteration, load the appropriate text string into the `text_vector`, then perform vector comparison:

```
    text_vector = vload16(0, text + i);
    check_vector = text_vector == pattern;
```

Then for each of the 4 input strings, check whether a match was found. If so, perform an `atomic_inc` operation (ensure that this is done sequentially so as to not have work-items performing this operation in parallel).

```
    if(all(check_vector.s0123))
       atomic_inc(local_result);

    if(all(check_vector.s4567))
       atomic_inc(local_result + 1);

    if(all(check_vector.s89AB))
       atomic_inc(local_result + 2);

    if(all(check_vector.sCDEF))
       atomic_inc(local_result + 3);
```

After the loop, a barrier is used to ensure that all work-items have completed their comparisons:

```
barrier(CLK_GLOBAL_MEM_FENCE);
```

Finally, perform a global reduction by using the first work-item in a work-group to perform an `atomic_add` (to ensure that this is done sequentially, so that values will not be overwritten if done in parallel):

```
if(get_local_id(0) == 0) {
   atomic_add(global_result, local_result[0]);
   atomic_add(global_result + 1, local_result[1]);
   atomic_add(global_result + 2, local_result[2]);
   atomic_add(global_result + 3, local_result[3]);
}
```

After the kernel has completed its computation, the results are read from the buffer object on the host:

```
queue.enqueueReadBuffer(resultBuffer, CL_TRUE, 0, sizeof(result), &result[0]);
```

Then, displayed on screen:

```
std::cout << "\nResults:" << std::endl;
std::cout << "Number of occurrences of 'that': " << result[0] << std::endl;
std::cout << "Number of occurrences of 'with': " << result[1] << std::endl;
std::cout << "Number of occurrences of 'have': " << result[2] << std::endl;
std::cout << "Number of occurrences of 'from': " << result[3] << std::endl;
```

References

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

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