Report of GPU programming using OpenCL

- Accelerated Part

To accelerate the enhancing image program, the computation of histogram should be parallelized. Since in this part, the program will do compute the histogram for every pixel, which is definitely a large computation.

- Code in OpenCL and Its Mechanism

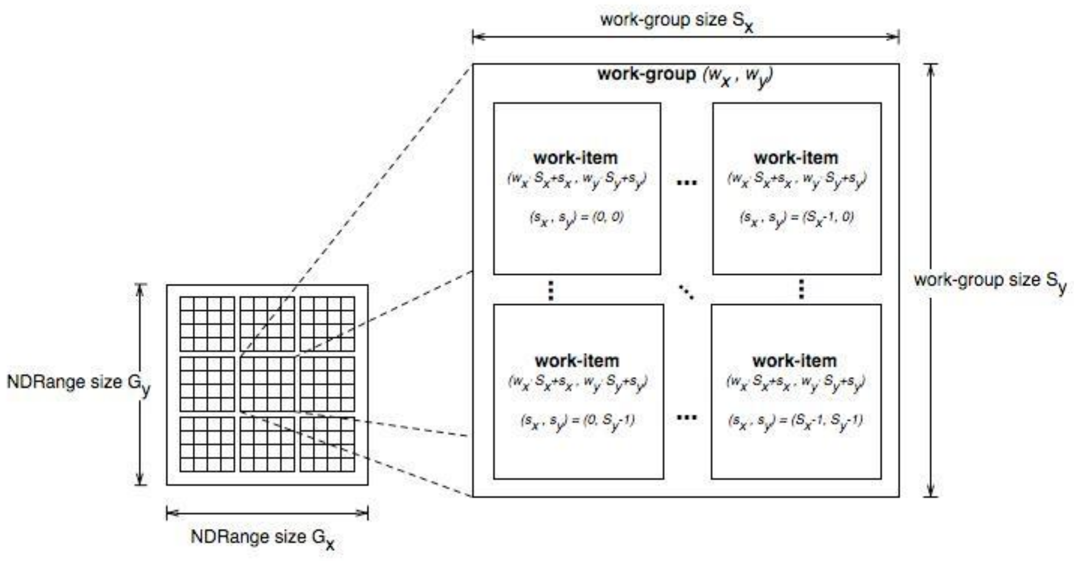
The code below is the parallel part using OpenCL:

```
//Returns the unique local work-item ID.
size_t localId = get_local_id(0);
//Returns the unique global work-item ID
size_t globalId = get_global_id(0);
//Returns the work-group ID.
size_t groupId = get_group_id(0);
//Returns the number of local work-items.
size_t groupSize = get_local_size(0);

//initialize the work item
for(int i = 0; i < binSize; ++i)
        workItem[localId * binSize + i] = 0;

barrier(CLK_LOCAL_MEM_FENCE);
//compute the local histogram in a work item
for(int i = 0; i < binSize; ++i)
{
        uint value = imageData[globalId * binSize + i];
        workItem[localId * binSize + value] += (float)256/(width * height);
}
barrier(CLK_LOCAL_MEM_FENCE);
```

To illuminate the mechanism of OpenCL, I use a picture of it to demonstrate:



The ND Range is the upmost level of OpenCL, it is initialized in the host(CPU), and it contains several workgroups, and in each workgroup, there are several work-items, the work-item is the smallest entity for OpenCL, the work-items are executed on the device. By separating all the work into work-items, the program can be parallelized.

In the code above, the localId is the ID for local work-item, and the array workItem[groupsize * binSize] represents one work-item. The code between two barriers is computing the histogram in the work-item.

- Number of work-items

The total number of work-items is depend on different devices.

In OpenCL, the device memory can be checked using code below:

```
//Device memory size
cl_ulong local_mem_size;
clGetDeviceInfo(device, CL_DEVICE_LOCAL_MEM_SIZE, sizeof(local_mem_size),
&local_mem_size, NULL);
```

In my computer, the local memory size is 32768 bytes. Since I use a float array in the work-item, the total number of work-items should not be larger than 8192. Since there are 256 work-items in each workgroup, the max workgroup size is 32.

- Data Read and Write between CPU and GPU

The GPU just executes the work-items, to distribute the work to device, we should use host(CPU).

In the host, we should initialize the OpenCL environment and write essential data to the buffer of device.

The code below is allocate the data to device:

```
//Create the buffer in device
deviceImageBuffer = clCreateBuffer(context,    CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR,    sizeof(cl_uchar) * width    * height, data,    0);

subDeviceHistogramBuffer = clCreateBuffer( context,    CL_MEM_READ_WRITE,
sizeof(cl_float) * binSize * subHistogramNum, NULL, 0);

//the the arg in kernel
errNum = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&deviceImageBuffer);
errNum = clSetKernelArg(kernel, 1, sizeof(int), &width);
errNum = clSetKernelArg(kernel, 2, sizeof(int), &height);
errNum = clSetKernelArg(kernel, 3, groupSize * binSize * sizeof(cl_float), NULL);
errNum = clSetKernelArg(kernel, 4, sizeof(cl_mem), (void*)&subDeviceHistogramBuffer);
```

As we can see, we store the data into deviceImageBuffer and set it to be the arg of kernel. Also, width and height is used in the kernel.

The fourth arg in the kernel is the memory allocated for work-items.

The last arg is the result of computation in device, and at last it should be read from the kernel, as the code shown below:

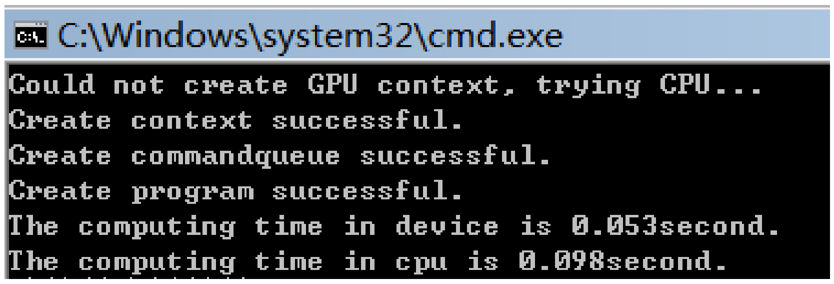
```
//read the sub histogram from device back to host
errNum = clEnqueueReadBuffer(commandQueue, subDeviceHistogramBuffer, CL_TRUE, 0,
subHistogramNum * binSize * sizeof(cl_float), subDeviceHistogram, 0, NULL, NULL);
```

So far, the computation using OpenCL is ended. And the parallelism is completed.

- Results using OpenCL

Since my computer just has an Intel(R) HD Graphics 3000 graphic card, it cannot been used as a device to execute the OpenCL. I use CPU to execute the program, it may not achieve the obviously result compared to GPU program, but the program does accelerate.

Here is the result of execute the program:



```
Could not create GPU context, trying CPU...
Create context successful.
Create commandqueue successful.
Create program successful.
The computing time in device is 0.053second.
The computing time in cpu is 0.098second.
```

The speedup is about 1.85.