Chapter 6
Basics of OpenCL

Takashi Nishitsuji

Abstract Open Computing Language (OpenCL), which is generally called a heterogeneous computing system, is an open programming framework of parallel computing for a calculation system comprising different computers (e.g., CPU, GPU, DSP, FPGA). Although CUDA only applies to NVIDIA's GPU, OpenCL can drive the GPUs of different vendors (AMD, NVIDIA, Intel, Qualcomm), as well as the CPU or another computer, via the same OpenCL-written source code. Thus, OpenCL is more portable than CUDA. In this chapter, OpenCL, as well as the strategy for constructing a calculation environment, is briefly introduced employing a source code example for calculating a computer-generated hologram (CGH). Based on the contents of this chapter, holography calculations employing OpenCL can be attempted. Readers who wish to improve their OpenCL coding skills, programming guides that are published by chip vendors, etc., may be consulted.

6.1 General Introduction of OpenCL

Open Computing Language (**OpenCL**) is an open framework of parallel computing for many devices (GPU, CPU, FPGA); it is dissimilar to CUDA that only supports NVIDIA's GPU. The specification of OpenCL was developed by the Khronos group [1], which is an open consortium of software frameworks.

Although device vendors supply the Software Development Kits (SDKs) of OpenCL that comply with the specifications of the Khronos groups, the extension deviates from the approved specifications. They exhibit two types of application programming interfaces (APIs): one is a candidate for future specifications, and the

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-981-99-1938-3_6.

T. Nishitsuji (✉)
Faculty of Systems Design, Tokyo Metropolitan University, 6-6 Asahigaoka, Hino-shi, Tokyo, Japan
e-mail: nishitsuji@gmail.com

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2023

T. Shimobaba and T. Ito (eds.), *Hardware Acceleration of Computational Holography*,
https://doi.org/10.1007/978-981-99-1938-3_6

other is vendor dependent and can be distinguished by their names [2]. Thus, the consumers must consider the conformance of each API.

Although OpenCL is based on the C language, there are some wrappers for other languages, e.g., the official C++ wrapper [3] and PyOpenCL [4] for python (further information are descriptive on the website of STREAM HPC [5]), enabling many software engineers to utilize GPGPU. This chapter focuses on OpenCL based on C/C++.

The basic techniques for accelerating a program with OpenCL and CUDA are almost similar; thus, this chapter focuses on clarifying the technique for utilizing OpenCL on your devices, as well as its differences with CUDA. However, owing to the page limit, the details of OpenCL (the definition of APIs) cannot be discussed; thus, the programming guides, which are released by vendors of the computing device, can be referenced by readers who wish to learn OpenCL detailedly [6–8].

6.2 Setting Up an OpenCL Environment

Most vendors of OpenCL-supporting devices avail their SDKs for developers; these SDKs include the OpenCL library of their devices and standard headers (.h), as well as other headers for extended functions that support only their devices. Therefore, intending users of OpenCL must first download and install the SDKs of their devices.

Notably, a Windows 10 64-bit environment was employed in this chapter, although readers employing other environments, e.g., macOS and Linux, can substitute the filenames or extensions according to the available environment, e.g., OpenCL.dll -> OpenCL.so for Linux users. The static"OpenCL.lib" and dynamic link "OpenCL.dll" libraries are the required libraries for developing and executing the OpenCL program. "OpenCL.lib" is available in the directories of an SDK, while "OpenCL.dll" is preinstalled in the system directories of Windows, following the installation of the graphics driver. Further, a header file ("cl.h"), which is available in the directory of an SDK, should be included in the program.

6.3 Constructing an OpenCL Program

This section introduces the construction of an OpenCL program employing a simple computer-generated hologram (CGH) calculation source code as a "Hello, world" program of OpenCL, which is depicted on Listings 6.1 (host program) and 6.2 (device program). Readers who have already set up the OpenCL environment can attempt to execute the sample codes by copying Listing 6.1 (with an appropriate name for a C++ file) to your computer and Listing 6.2 with the name, "CGH_helloworld.cl," which should be placed in the same directory with an executable file of Listing 6.1. After executing the program, a kinoform-type CGH with a resolution of 1024×1024 in the "bfh_CGH" buffer can be obtained, as shown in Fig. 6.1.

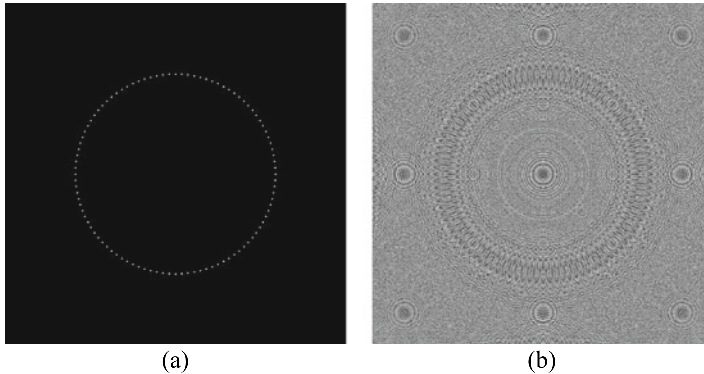

Fig. 6.1 Input and output of the example code: **a** a 3D model with 100 point clouds (input, generated in the program), **b** kinoform-type CGH (output)

An OpenCL program comprises two types of source codes, the host (.c or .cpp, .h) and device (.cl) codes. A standard OpenCL program adopts the online compile of the device code to improve its portability. Therefore, a C/C++ compiler, e.g., clang, gcc, and Visual C++, compiles the host code employing the OpenCL static library and creates the executable file, which will read and compile the device code according to specified devices for the program, following its execution. Noteworthily, OpenCL also supports offline compile.

The most significant differences between CUDA and OpenCL are the concepts of the platform and the devices. Since OpenCL supports many computing devices, an OpenCL program requires the availability of the available devices; users must specify the desired devices to execute the program. Every device must correspond to a platform. For example, when executing OpenCL on a CPU Intel Core–i7 8700K CPU employing an Intel OpenCL SDK environment, the platform would be "Intel OpenCL," and two devices (Integrated GPU, Intel UHD Graphics 630, and Intel Core i7-8700K CPU), which are available on the platform, would be utilized. The platforms and devices are specified by IDs; thus, many OpenCL APIs requests set the IDs in the arguments.

6.3.1 Creating OpenCL Objects That Are Not Required in CUDA

Dissimilar to CUDA, OpenCL defines many objects, e.g., the memory and kernel objects, to manage the device-related information, such as memory address and binary code of an executing program, since OpenCL is assumed to be executed on different platforms and devices. Thus, OpenCL requires the creation of such objects before the execution of a kernel. Table 6.1 and Fig. 6.2 exhibit the required

Table 6.1 Definition of the objects in OpenCL

Name of object	Role	Defined per
Context	Manages all the objects on a platform	Platform
Command queue	Manges all the commands to a device	Device
Program object	Manages the device program	Device source code
Kernel object	Compiles the kernel function of the device	Kernel function
Memory object	Manages the memory space on a device	Buffer

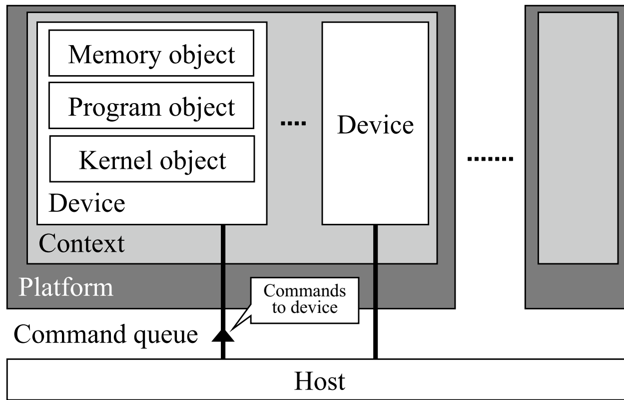

Fig. 6.2 Calculation model of OpenCL employing relation between the objects

object in a standard OpenCL program and the roles and relation between the objects, respectively. The OpenCL objects that are not required in CUDA are introduced in this subsection with reference to the sample code in Listing 6.1.

Context object is a fundamental object for managing all the objects on a platform; thus, it must be declared on the first line of an OpenCL program with the intended platform ID, as well as the number of devices on the platform. The available platforms and devices can be obtained by "clGetPlatformIDs()," which was employed on Lines 65 and 69 of Listing 6.1, and "clGetDeviceIDs()," which was used on Line 86 of Listing 6.1, for the platforms and devices, respectively. Detailed information on the platforms and devices can be obtained by "clGetPlatformInfo()" and "clGetDeviceInfo()," which employed utilized on Lines 77 and 91, respectively. Here, this program obtains the names of the platforms and devices. The context object is created by API "clCreateContext()," which was employed on Line 115 of the list.

command-queue object is an interface that manages all the commands, e.g., the execute-the-kernel and the transfer-the-data-in-a-buffer functions; thus, it must be declared per all to-be-utilized devices. A command-queue object is created by "clCreateCommandQueueWithProperties()" with a corresponding device ID, which is depicted on Line 118 of the list. The commands to a device are queued by the "clEnqueue***()" API via a command-queue object. For example, to copy data from the

Table 6.2 Corresponding names of memory

CUDA	OpenCL
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Register	Private memory
Local memory	

memory of a device to a host, "clEnqueueReadBuffer()," Line 165 of the list, is called employing the command-queue object in the first argument. Worthy, the commands are only enqueued; thus, the time of executing is unknown, and it depends on the preceding commands on the queue.

program object is an object that manages a raw (readable text) source code, as well as the compiled program of a device function. Thus, it must read a device source code as a text buffer before creating it. Lines 122–133 on the list show an example of reading the device source code from a file (CGH_helloworld.cl) to a char buffer (src), as well as creating a program object with "clCreateProgramWithSource()" on Line 128. After creating the program object, it can be built by "clBuildProgram()" employing a specified platform ID, as shown on Line 131 of the List.

kernel object is an object, which is created by the "clCreateKernel()" function employing program object and named the kernel function, that specifies the kernel function in a program object; thus, it must be created per device functions to be executed. On the List, only one device function is defined in the device code (Listing 6.2); therefore, only one kernel object is created on Line 136 of the Listing 6.1.

memory object is an object that manages the memory buffer on a device. It functions as a memory pointer. The memory object is created by "clCreateBuffer()" with context object and attributions that pertain to memory (size and writability), as obtainable in "cudaMalloc()" of CUDA. On Listing 6.1, four memory objects were created on Lines 139–142. Noteworthy, the hierarchical memory architectures of OpenCL and CUDA are almost the same (Table 6.2), and the memory buffer, which was created by "clCreateBuffer()," is assigned on the global memory.

The creations of the discussed objects indicate that the preparation for executing the kernel is almost completed. Further, the following section introduces the procedure for driving the OpenCL kernel.

6.3.2 *Executing the Kernel Function*

Dissimilar to CUDA, OpenCL requires a two-step setup before executing the enqueued kernel. The first step involves setting up the arguments of the kernel function via the "clSetKernelArg()" function (Lines 151–155 on Listing 6.1). Notably, all the arguments must be passed by a void* type pointer.

The second step involves the definition of the division unit for parallel execution; these units are called the grid, block, and thread in CUDA. However, the "grid," "block," and "thread" correspond to "NDRange," "workgroup," and "workitem," respectively. The sizes of NDRange and workgroup are specified by multidimensional size_t-type arrays, as exhibited on Lines 158 and 159 of the List. In the sample code, the size of NDRange was set to be equal to the size of the CGH, and the size of the workgroup was set to 256 $\times$ 1. The maximum number of workitems in a work group is defined by the specifications of hardware.

After the two-step preparation, the command for executing the kernel function can be enqueued by "clEnqueueNDRange()" employing the sizes of NDRange (globalSize), workgroup (localSize), and the queue object (Line 162 of the list).

Finally, the kernel function can be executed by transferring the buffer data from the device to the host. "clEnqueueReadBuffer()" is a transfer function; it is executed to transfer the buffer data from the device to the host (Line 165 of the list), and it is equivalent to "cudaMemcopy()" in CUDA. To ensure complete transfer, a call function for synchronizing the device to the host must be executed before subjecting the data to the host buffer (bfh_CGH). In the sample code, the "clFinish()" function, which was waiting to execute the last command that was enqueued in the command queue, was executed. Noteworthy, there are other functions, e.g., clWaitForEvents() with an event object, for achieving a finer synchronization; thus, those APIs can be referenced by readers who wish to construct a more complex OpenCL program.

This subsection only discusses the method for executing data-parallel-type computation. However, OpenCL comprises methods for parallelizing the calculation in a task unit, as obtainable in CUDA. Readers who wish to employ the task-parallel program may refer to the instruction manual of OpenCL, which is supplied by the vendors of devices.

To summarize the above introductions, the standard structure of the host program of OpenCL is, as follows:

1. Determine an available platform, as well as devices, and specify the appropriate devices.
2. Create a context object, which manages all the objects on a platform.
3. Create a command-queue object, which is connected to a device to manage the commands to be executed therein.
4. Read a device program as a text and build it, thereby treating it as a program object.
5. Create the kernel objects from a program object by specifying the name of the function that was written in the .cl file
6. Create the memory objects, which manage the memory space on a device.
7. Set the arguments and workgroup size, which are to be executed by the kernel.
8. Execute the kernel function.
9. Copy the result from the device memory.

Table 6.3 Corresponding names of the modifiers of the variables and memories

CUDA	OpenCL	Meaning
__device__	__global	On the global memory
__constant__	__constant	On the constant memory
__shared__	__local	On the shared memory

Table 6.4 Corresponding names of the modifier of the functions

CUDA	OpenCL	Meaning
__global__	__global	Kernel function
__device__	Not required	Inner function of the kernel

6.3.3 Writing the Kernel Function

The kernel function is one, which would be executed by a device. The grammars and syntaxes of the kernel functions of OpenCL and CUDA are almost the same, although the names of the modifiers of their variables, memories, and functions, as well as the methods for obtaining their index values, e.g., "gridDim" in CUDA, are different. Tables 6.3, 6.4, and 6.5 present the correlations of the modifiers and other basic functions of CUDA and OpenCL. N in Table 6.5 indicates that a dimension must be obtained employing the functions; thus, blockDim.x in CUDA is equivalent to get_num_groups(0);

The standard kernel function for calculating CGH is presented on Listing 6.2, which is a simplified version of the sample code of calculating CGH employing CUDA (Listing 10.2). For the readers who wish to execute an OpenCL program, the modification of Listing 6.2 is an easy technique for first building the OpenCL program. Here (Listing 6.2), three pre-processors are defined to substitute the constant

Table 6.5 Corresponding methods for obtaining the index values: N is a dimension

CUDA	OpenCL	Meaning
gridDim	get_num_groups(N)	Number of blocks per grid
blockDim	get_local_size(N)	Size of a block
blockIdx	get_group_id(N)	Index of a block
threadIdx	get_local_id(N)	Index of a thread
threadIdx + blockIdx * blockDim	get_global_id(N)	Global index of a thread
gridDim * blockDim	get_global_size(N)	Size of a grid

values. “CNS_255_DIV_2_PI” and “CNS_2_PI_DIV_LAMBDA” correspond to $\frac{255}{2\pi}$ and $\frac{2\pi}{\lambda}$, respectively ($\lambda = 532$ [nm] and “CNS_PITCH” represents the pixel pitch of a displaying device.

The calculation times for this execution are 95.2 ms with NVIDIA Quadro P1200 GPU and CUDA 11.0, 1738 ms with an Intel Core i7-8850H CPU, and 324 ms with an Intel UHD Graphics 630 GPU, all of them are evaluated with OpenCL. The kernel source code (Listing 6.2) is a very simple structure to understand; thus, applying the optimization techniques that are mentioned in Chapter 6 will be quite fast. Unfortunately, the techniques described in those sections are not within the scope of OpenCL, although readers who already briefly understand the differences and similarities of CUDA and OpenCL can easily apply those techniques in their OpenCL codes.

Moreover, only a few literature illustrate the fast calculation of CGH via OpenCL, although readers can refer to [9] as a practical example of implementing OpenCL to calculate CGH.

Listing 6.1 Simple CGH calculation employing OpenCL (host code)

```
1  #include <CL/cl.h>
2  #include <stdio.h>
3  #include <math.h>
4
5  #define MAX_CL_SOURCE_SIZE 10000
6  #define PI 3.14159265358979323846
7
8  int main()
9  {
10   //Constants********************************
11   const char st_CLSrcName[1024] = "CGH_helloworld.cl";
12   const int numPLS = 100;  //number of PLS
13   const int cgh_width = 1024;  //width of CGH [pixel]
14   const int cgh_height = 1024; //height of CGH [pixel]
15   const float p = 0.000008;  //pixel pitch for displaying device [m]
16
17   //Classes**********************************
18   FILE* fp_CLSrc = fopen(st_CLSrcName, "rb");
19
20   //Control variables for OpenCL
21   cl_int status = 0;
22
23   cl_platform_id v_SelectedPlatformID = 0;
24   cl_platform_id* v_PlatformIDs;
25   unsigned int v_SelectedPlatform;
26   unsigned int v_NumPlatforms;
27
28   cl_device_id v_SelectedDeviceID = 0;
29   cl_device_id** v_DeviceIDs;
30   unsigned int v_SelectedDevice;
31   unsigned int v_NumDevices;
32
33   cl_context context;
34   cl_command_queue queue;
```

```
35 cl_program prog;
36 cl_kernel ker_CGH;
37
38 //Buffers**********************************
39 //(host)
40 cl_uchar* bfh_CGH = new cl_uchar[cgh_width * cgh_height];
41 cl_float* bfh_ox = new cl_float[numPLS];
42 cl_float* bfh_oy = new cl_float[numPLS];
43 cl_float* bfh_oz = new cl_float[numPLS];
44
45 //(device)
46 cl_mem bfd_CGH;
47 cl_mem bfd_ox;
48 cl_mem bfd_oy;
49 cl_mem bfd_oz;
50
51 //===Create Point cloud (circle)===
52 float r = 300 * p;   //radius of circle
53 float cx = cgh_width * 0.5 * p; //center of circle (x)
54 float cy = cgh_width * 0.5 * p; //center of circle (y)
55
56 for (int i = 0; i < numPLS; i++)
57 {
58   bfh_ox[i] = r * cos(i / (float)numPLS * 2.0 * PI) + cx;
59   bfh_oy[i] = r * sin(i / (float)numPLS * 2.0 * PI) + cy;
60   bfh_oz[i] = 0.1 + 0.001*i;
61 }
62
63 //====Select the platform and devices to use====//
64 //Obtain the number of available platforms
65 status = clGetPlatformIDs(0, NULL, &v_NumPlatforms);
66 v_PlatformIDs = new cl_platform_id[v_NumPlatforms];
67
68 //Obtain the IDs of available platform
69 status = clGetPlatformIDs(v_NumPlatforms, v_PlatformIDs, &v_NumPlatforms);
70 v_DeviceIDs = new cl_device_id*[v_NumPlatforms];
71
72 //Show available platforms and device IDs
73 char msg[1024];
74 for (int i = 0; i < v_NumPlatforms; i++)
75 {
76   //Obtain platform information (name of platform)
77   clGetPlatformInfo(v_PlatformIDs[i], CL_PLATFORM_NAME, sizeof(msg), msg,
         NULL);
78   printf("[%d] : %s\n", i, msg);
79
80   //Obtain the number of available devices on the platform
81   status = clGetDeviceIDs(v_PlatformIDs[i], CL_DEVICE_TYPE_ALL, NULL, NULL,
         &v_NumDevices);
82   printf("Found %d devices\n", v_NumDevices);
83
84   //Obtain the IDs of available platform
85   v_DeviceIDs[i] = new cl_device_id[v_NumDevices];
```

```
86    status = clGetDeviceIDs(v_PlatformIDs[i], CL_DEVICE_TYPE_ALL, v_NumDevices,
          v_DeviceIDs[i], &v_NumDevices);
87
88    //Show the avaialble devices in the platform
89    for (int j = 0; j < v_NumDevices; j++)
90    {
91      clGetDeviceInfo(v_DeviceIDs[i][j], CL_DEVICE_NAME, sizeof(msg), msg, NULL);
92      printf("\t[%d][%d]%s\n", i, j, msg);
93    }
94  }
95
96  //Select the platform and devices to use
97  printf("Select platform ID to use: ");
98  scanf_s("%d", &v_SelectedPlatform);
99
100 v_SelectedPlatformID = v_PlatformIDs[v_SelectedPlatform];
101 clGetPlatformInfo(v_SelectedPlatformID, CL_PLATFORM_NAME, sizeof(msg), msg,
        NULL);
102 printf("Selected: %s\n\n", msg);
103
104 printf("Select device ID to use: ");
105 scanf_s("%d", &v_SelectedDevice);
106 v_SelectedDeviceID = v_DeviceIDs[v_SelectedPlatform][v_SelectedDevice];
107 clGetDeviceInfo(v_SelectedDeviceID, CL_DEVICE_NAME, sizeof(msg), msg, NULL);
108 printf("Selected: %s\n\n", msg);
109
110 //====Create a context====//
111 //obtain the number of devices in the selected platform
112 clGetDeviceIDs(v_SelectedPlatformID, CL_DEVICE_TYPE_ALL, NULL, NULL, &
        v_NumDevices);
113
114 //Create a context for the selected platform
115 context = clCreateContext(NULL, v_NumDevices, v_DeviceIDs[v_SelectedPlatform],
        NULL, NULL, &status);
116
117 //====Create a command queue on the context====//
118 queue = clCreateCommandQueueWithProperties(context, v_DeviceIDs[
        v_SelectedPlatform][v_SelectedDevice], NULL, &status);
119
120 //====Build a program from a .cl source====//
121 //Read .cl file to char buffer as text
122 char* src;
123 src = new char[MAX_CL_SOURCE_SIZE];
124 size_t v_SizeOfSrc = fread(src, sizeof(char), MAX_CL_SOURCE_SIZE - 1, fp_CLSrc);
125 src[v_SizeOfSrc] = '\0';
126
127 //Create program object with the .cl source file
128 prog = clCreateProgramWithSource(context, 1, (const char**)&src, NULL, &status);
129
130 //Build program
131 status = clBuildProgram(prog, v_NumDevices, v_DeviceIDs[v_SelectedPlatform], NULL,
         NULL, NULL);
132
```

```
133  delete[] src;
134
135  //====Create kernels to execute====//
136  ker_CGH = clCreateKernel(prog, "simpleCGH", &status);
137
138  //====Create memory objects====//
139  bfd_CGH = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(cl_uchar)*
         cgh_width*cgh_height, NULL, &status);
140  bfd_ox = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * numPLS,
         NULL, &status);
141  bfd_oy = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * numPLS,
         NULL, &status);
142  bfd_oz = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * numPLS,
         NULL, &status);
143
144  //====Transfer the PLS data from the host ====//
145  status = clEnqueueWriteBuffer(queue, bfd_ox, CL_TRUE, 0, sizeof(cl_float) * numPLS,
         bfh_ox, 0, NULL, NULL);
146  status = clEnqueueWriteBuffer(queue, bfd_oy, CL_TRUE, 0, sizeof(cl_float) * numPLS,
         bfh_oy, 0, NULL, NULL);
147  status = clEnqueueWriteBuffer(queue, bfd_oz, CL_TRUE, 0, sizeof(cl_float) * numPLS,
         bfh_oz, 0, NULL, NULL);
148
149  //====Execute kernels====//
150  //Set arguments of the kernel
151  status = clSetKernelArg(ker_CGH, 0, sizeof(cl_mem), (void*)&bfd_CGH);
152  status = clSetKernelArg(ker_CGH, 1, sizeof(int), (void*)&numPLS);
153  status = clSetKernelArg(ker_CGH, 2, sizeof(cl_mem), (void*)&bfd_ox);
154  status = clSetKernelArg(ker_CGH, 3, sizeof(cl_mem), (void*)&bfd_oy);
155  status = clSetKernelArg(ker_CGH, 4, sizeof(cl_mem), (void*)&bfd_oz);
156
157  //Set the division unit for parallel execution
158  size_t globalSize[] = { (size_t)cgh_width, (size_t)cgh_height };
159  size_t localSize[] = { 256, 1 };
160
161  //Execute the kernel
162  status = clEnqueueNDRangeKernel(queue, ker_CGH, 2, NULL, globalSize, localSize, 0,
         NULL, NULL);
163
164  //====Transfer the CGH data from the device====//
165  status = clEnqueueReadBuffer(queue, bfd_CGH, CL_TRUE, 0, sizeof(cl_char)*
         cgh_width*cgh_height, bfh_CGH, 0, NULL, NULL);
166
167  //Wait for finish the last enqueued command
168  clFinish(queue);
169
170  //====Termination (Freeing memory)====//
171  fclose(fp_CLSrc);
172  clReleaseMemObject(bfd_CGH);
173  clReleaseMemObject(bfd_ox);
174  clReleaseMemObject(bfd_oy);
175  clReleaseMemObject(bfd_oz);
176
```

```
177   delete[] bfh_CGH;
178   delete[] bfh_ox;
179   delete[] bfh_oy;
180   delete[] bfh_oz;
181
182   return 0;
183 }
```

Listing 6.2 Simple CGH calculation employing OpenCL (device code; CGHspshelloworld.cl)

```
1 #define CNS_255_DIV_2_PI 40.58451049
2 #define CNS_2_PI_DIV_LAMBDA 11810498.7
3 #define CNS_PITCH 0.000008
4
5 __kernel void simpleCGH(__global uchar* dbf_CGH, const int numPLS, __global float*
        ox, __global float* oy, __global float*oz)
6 {
7   float x = get_global_id(0) * CNS_PITCH;
8   float y = get_global_id(1) * CNS_PITCH;
9   int width = get_global_size(0);
10  int dst_addr = get_global_id(0) + get_global_size(0) * get_global_id(1);
11
12  float2 c = (float2)(0.0, 0.0);
13
14  for (int i = 0; i < numPLS; i++)
15  {
16    float phase = CNS_2_PI_DIV_LAMBDA * sqrt(pow(ox[i]−x, 2) + pow(oy[i]−y, 2) +
          pow(oz[i], 2));
17    c += (float2)(cos(phase), sin(phase));
18  }
19
20  float arg = CNS_255_DIV_2_PI * atan2(c.y, c.x);
21  dbf_CGH[dst_addr] = convert_uchar((int)arg);
22 }
```

Fundings This work was supported by JSPS KAKENHI Grant Number 22H03616.

References

1. Official OpenCL website of khronos group https://www.khronos.org/opencl/ Cited 30 Oct. 2019.
2. Khonos group, The OpenCL Extension Specification https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_Ext.html. Cited 30 Oct. 2019
3. Khronos's github repository for OpenCL C++ bindings https://github.khronos.org/OpenCL-CLHPP/ Cited 30 Oct. 2019
4. A. Klöckener, PyOpenCL https://mathema.tician.de/software/pyopencl/. Cited 30 Oct. 2019
5. STREAM High Peformance Computing, OpenCL Wrappers https://streamhpc.com/knowledge/for-developers/opencl-wrappers/. Cited 30 Oct. 2019
6. NVIDIA, OpenCL Programming Guide for the CUDA Architecture Ver 4.2 http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf. Cited 30 Oct. 2019

7. Intel, Intel FPGA SDK for OpenCL Pro Edition Programming Guide 19.3 https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf. Cited 30 Oct. 2019
8. Qualcomm, Snapdragon Mobile Platform OpenCL General Programming and Optimization, Nov. 3, 2017 https://developer.qualcomm.com/software/adreno-gpu-sdk/tools. Cited 30 Oct. 2019
9. Shimobaba, T., Ito. T., Masuda, N., Ichihashi. Y., Takada, N.: Fast calculation of computer-generated-hologram on AMD HD5000 series GPU and OpenCL, Opt. Express **18**, 10, 9955–9960 (2010).