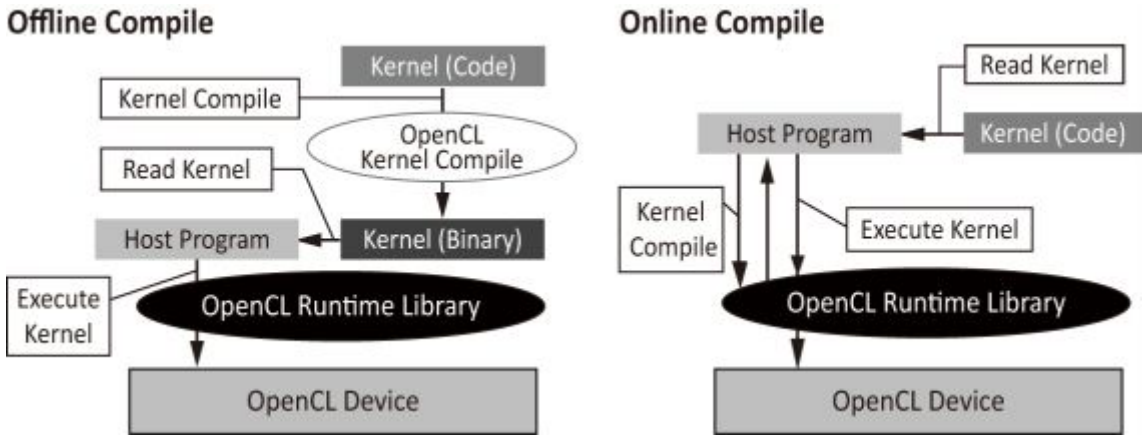
The OpenCL Programming Book

4.2 Online/Offline Compilation

In OpenCL, a kernel can be compiled either online or offline (Figure 4.1).

Figure 4.1: Offline and Online Compilation



The basic difference between the 2 methods is as follows:

- Offline: Kernel binary is read in by the host code

- Online: Kernel source file is read in by the host code

In "offline-compilation", the kernel is pre-built using an OpenCL compiler, and the generated binary is what gets loaded using the OpenCL API. Since the kernel binary is already built, the time lag between starting the host code and the kernel getting executed is negligible. The problem with this method is that in order to execute the program on various platforms, multiple kernel binaries must be included, thus increasing the size of the executable file.

In "online-compilation", the kernel is built from source during runtime using the OpenCL runtime library. This method is commonly known as JIT (Just in time) compilation. The advantage of this method is that the host-side binary can be distributed in a form that's not device-dependent, and that adaptive compiling of the kernel is possible. It also makes the testing of the kernel easier during development, since it gets rid of the need to compile the kernel each time. However, this is not suited for embedded systems that require real-time processing. Also, since the kernel code is in readable form, this method may not be suited for commercial applications.

The OpenCL runtime library contains the set of APIs that performs the above operations. In a way, since OpenCL is a programming framework for heterogeneous environments, the online compilation support should not come as a shock. In fact, a stand-alone OpenCL compiler is not available for the OpenCL environment by NVIDIA, AMD, and Apple. Hence, in order to create a kernel binary in these environments, the built kernels has to be written to a file during runtime by the host program. FOXC on the other hand includes a stand-alone OpenCL kernel compiler, which makes the process of making a kernel binary intuitive.

We will now look at sample programs that show the two compilation methods. The first code shows the online compilation version.

List 4.2: Online Compilation version

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.
4.  #ifdef __APPLE__
5.  #include <OpenCL/opencl.h>
6.  #else
7.  #include <CL/cl.h>
8.  #endif
9.
10. #define MEM_SIZE (128)
```

» Mersenne Twister

› Notes

```
#define MAX_SOURCE_SIZE (0x100000)

int main()
{
        cl_platform_id platform_id = NULL;
        cl_device_id device_id = NULL;
        cl_context context = NULL;
        cl_command_queue command_queue = NULL;
        cl_mem memobj = NULL;
        cl_program program = NULL;
        cl_kernel kernel = NULL;
        cl_uint ret_num_devices;
        cl_uint ret_num_platforms;
        cl_int ret;

        float mem[MEM_SIZE];

        FILE *fp;
        const char fileName[] = "./kernel.cl";
        size_t source_size;
        char *source_str;
        cl_int i;

        /* Load kernel source code */
        fp = fopen(fileName, "r");
        if (!fp) {


        }
        source_str = (char *)malloc(MAX_SOURCE_SIZE);
        source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
        fclose(fp);

        /*Initialize Data */
        for (i = 0; i < MEM_SIZE; i++) {

        }

        /* Get platform/device information */
```

```
50.         ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
51.         ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, &r
    et_num_devices);
52.
53.         /* Create OpenCL Context */
54.         context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
55.
56.         /* Create Command Queue */
57.         command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
58.
59.         /* Create memory buffer*/
60.         memobj = clCreateBuffer(context, CL_MEM_READ_WRITE, MEM_SIZE *
    sizeof(float), NULL, &ret);
61.
62.         /* Transfer data to memory buffer */
63.         ret = clEnqueueWriteBuffer(command_queue, memobj, CL_TRUE, 0, MEM_SIZE * si
    zeof(float), mem, 0, NULL, NULL);
64.
65.         /* Create Kernel program from the read in source */
66.         program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
     (const size_t *)&source_size, &ret);
67.
68.         /* Build Kernel Program */
69.         ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
70.
71.         /* Create OpenCL Kernel */
72.         kernel = clCreateKernel(program, "vecAdd", &ret);
73.
74.         /* Set OpenCL kernel argument */
75.         ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj);
76.
77.         size_t global_work_size[3] = {MEM_SIZE, 0, 0};
78.         size_t local_work_size[3] = {MEM_SIZE, 0, 0};
79.
80.         /* Execute OpenCL kernel */
81.         ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global_work_si
    ze, local_work_size, 0, NULL, NULL);
82.
83.         /* Transfer result from the memory buffer */
```

```
84.         ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0, MEM_SIZE * siz
    eof(float), mem, 0, NULL, NULL);
85.
86.         /* Display result */
87.         for (i=0; i < MEM_SIZE; i++) {
88.
89.         }
90.
91.         /* Finalization */
92.         ret = clFlush(command_queue);
93.         ret = clFinish(command_queue);
94.         ret = clReleaseKernel(kernel);
95.         ret = clReleaseProgram(program);
96.         ret = clReleaseMemObject(memobj);
97.         ret = clReleaseCommandQueue(command_queue);
98.         ret = clReleaseContext(context);
99.
100.        free(source_str);
101.
102.        return 0;
103.        }
```

The following code shows the offline compilation version (List 4.3).

List 4.3 Offline compilation version

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.
4.  #ifdef __APPLE__
5.  #include <OpenCL/opencl.h>
6.  #else
7.  #include <CL/cl.h>
8.  #endif
9.
10. #define MEM_SIZE (128)
11. #define MAX_BINARY_SIZE (0x100000)
12.
```

```
13. int main()
14. {
15.         cl_platform_id platform_id = NULL;
16.         cl_device_id device_id = NULL;
17.         cl_context context = NULL;
18.         cl_command_queue command_queue = NULL;
19.         cl_mem memobj = NULL;
20.         cl_program program = NULL;
21.         cl_kernel kernel = NULL;
22.         cl_uint ret_num_devices;
23.         cl_uint ret_num_platforms;
24.         cl_int ret;
25.
26.         float mem[MEM_SIZE];
27.
28.         FILE *fp;
29.         char fileName[] = "./kernel.clbin";
30.         size_t binary_size;
31.         char *binary_buf;
32.         cl_int binary_status;
33.         cl_int i;
34.
35.         /* Load kernel binary */
36.         fp = fopen(fileName, "r");
37.         if (!fp) {
38.
39.
40.         }
41.         binary_buf = (char *)malloc(MAX_BINARY_SIZE);
42.         binary_size = fread(binary_buf, 1, MAX_BINARY_SIZE, fp);
43.         fclose(fp);
44.
45.         /* Initialize input data */
46.         for (i = 0; i < MEM_SIZE; i++) {
47.
48.         }
49.
50.         /* Get platform/device information */
51.         ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
```

```
52.         ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, &r
    et_num_devices);
53.
54.         /* Create OpenCL context*/
55.         context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
56.
57.         /* Create command queue */
58.         command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
59.
60.         /* Create memory buffer */
61.         memobj = clCreateBuffer(context, CL_MEM_READ_WRITE, MEM_SIZE *
    sizeof(float), NULL, &ret);
62.
63.         /* Transfer data over to the memory buffer */
64.         ret = clEnqueueWriteBuffer(command_queue, memobj, CL_TRUE, 0, MEM_SIZE * si
    zeof(float), mem, 0, NULL, NULL);
65.
66.         /* Create kernel program from the kernel binary */
67.         program = clCreateProgramWithBinary(context, 1, &device_id, (const size_t
    *)&binary_size,
68.         (const unsigned char **)&binary_buf, &binary_status, &ret);
69.
70.         /* Create OpenCL kernel */
71.         kernel = clCreateKernel(program, "vecAdd", &ret);
72.         printf("err:%d\n", ret);
73.
74.         /* Set OpenCL kernel arguments */
75.         ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj);
76.
77.         size_t global_work_size[3] = {MEM_SIZE, 0, 0};
78.         size_t local_work_size[3] = {MEM_SIZE, 0, 0};
79.
80.         /* Execute OpenCL kernel */
81.         ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global_work_si
    ze, local_work_size, 0, NULL, NULL);
82.
83.         /* Copy result from the memory buffer */
84.         ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0, MEM_SIZE * siz
    eof(float), mem, 0, NULL, NULL);
```

```
85.
86.         /* Display results */
87.         for (i=0; i < MEM_SIZE; i++) {
88.
89.         }
90.
91.         /* Finalization */
92.         ret = clFlush(command_queue);
93.         ret = clFinish(command_queue);
94.         ret = clReleaseKernel(kernel);
95.         ret = clReleaseProgram(program);
96.         ret = clReleaseMemObject(memobj);
97.         ret = clReleaseCommandQueue(command_queue);
98.         ret = clReleaseContext(context);
99.
100.        free(binary_buf);
101.
102.        return 0;
103.        }
```

The kernel program performs vector addition. It is shown below in list 4.4.

List 4.4: Kernel program

```
1.  __kernel void vecAdd(__global float* a)
2.  {
3.          int gid = get_global_id(0);
4.          a[gid] += a[gid];
5.  }
```

We will take a look at the host programs shown in List 4.2 and List 4.3. The two programs are almost identical, so we will focus on their differences.

The first major difference is the fact that the kernel source code is read in the online compile version (List 4.5).

List 4.5: Online compilation version - Reading the kernel source code

```
        fp = fopen(fileName, "r");
        if (!fp) {
                fprintf(stderr, "Failed to load kernel.\n");
                exit(1);
        }
        source_str = (char *)malloc(MAX_SOURCE_SIZE);
        source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
        fclose(fp);
```

The source_str variable is a character array that merely contains the content of the source file. In order to execute this code on the kernel, it must be built using the runtime compiler. This is done by the code segment shown below in List 4.6.

List 4.6: Online compilation version - Create kernel program

```
        /* Create kernel program from the source */
        program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
(const size_t *)&source_size, &ret);
        /* Build kernel program */
        ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
```

The program is first created from source, and then built.

Next, we will look at the source code for the offline compilation version (List 4.7).

List 4.7: Offline compilation - Reading the kernel binary

```
        fp = fopen(fileName, "r");
        if (!fp) {
                fprintf(stderr, "Failed to load kernel.\n");
                exit(1);
        }
        binary_buf = (char *)malloc(MAX_BINARY_SIZE);
        binary_size = fread(binary_buf, 1, MAX_BINARY_SIZE, fp);
        fclose(fp);
```

The code looks very similar to the online version, since the data is being read into a buffer of type char. The difference is that the data on the buffer can be directly executed. This means that the kernel source code must be compiled beforehand using an OpenCL compiler. In FOXC, this can be done as follows.

```
> /path-to-foxc/bin/foxc -o kernel.clbin kernel.cl
```

The online compilation version required 2 steps to build the kernel program. With offline compilation, the clCreateProgramWithSource() is replaced with clCreateProgramWithBinary.

```
1.          program = clCreateProgramWithBinary(context, 1, &device_id, (const size_t
*)&binary_size,
2.                  (const unsigned char **)&binary_buf, &binary_status, &ret);
```

Since the kernel is already built, there is no reason for another build step as in the online compilation version.

To summarize, in order to change the method of compilation from online to offline, the following steps are followed:

1. Read the kernel as a binary

2. Change clCreateProgramWithSource() to clCreateProgramWithBinary()

3. Get rid of clBuildProgram()

This concludes the differences between the two methods. See chapter 7 for the details on the APIs used inside the sample codes.

Previous | Next

processors programming and the use of next generation memory technology, Fixstars provides the best performance and the highest capacity storage solutions to deliver high speed IO as well as power savings to accelerate customers' business in various fields.

9205 Research Drive, 1st Floor, Irvine, California 92618

View Map

Contact form

Follow us on Twitter