Porting CUDA to OpenCL

From Documentation

The data-parallel programming model in OpenCL shares some commonalities with CUDA programming model, making it relatively straightforward to convert programs from CUDA to OpenCL.

Contents

Hardware Terminology

CUDA	OpenCL
SM (Stream Multiprocessor)	CU (Compute Unit)
Thread	Work-item
Block	Work-group
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory

Private memory (local memory in CUDA) used within a work item that is similar to registers in a GPU multiprocessor or CPU core. Variables inside a kernel function not declared with an address space qualifier, all variables inside non-kernel functions, and all function arguments are in the __private or private address space. Application performance can plummet when too much private memory is used on some devices – like GPUs because it is spilled to slower memory. Depending on the device, private memory can be spilled to cache memory. GPUs that do not have cache memory will spill to global memory causing significant performance drops.

Qualifiers for Kernel Functions

CUDA	OpenCL
__global__ function	__kernel function
__device__ function	No annotation necessary
__constant__ variable declaration	__constant variable declaration
__device__ variable declaration	__global variable declaration
__shared__ variable declaration	__local variable declaration

Kernels Indexing

CUDA	OpenCL
gridDim	get_num_groups()
blockDim	get_local_size()
blockIdx	get_group_id()
threadIdx	get_local_id()
blockIdx * blockDim + threadIdx	get_global_id()
gridDim * blockDim	get_global_size()

CUDA is using threadIdx.x to get the id for the first dimension while OpenCL is using get_local_id(0).

Kernels Synchronization

CUDA	OpenCL
__syncthreads()	barrier()
__threadfence()	No direct equivalent
__threadfence_block()	mem_fence()
No direct equivalent	read_mem_fence()
No direct equivalent	write_mem_fence()

API Calls

CUDA	OpenCL
cudaGetDeviceProperties()	clGetDeviceInfo()
cudaMalloc()	clCreateBuffer()
cudaMemcpy()	clEnqueueRead(Write)Buffer()
cudaFree()	clReleaseMemObj()
kernel<<<...>>>()	clEnqueueNDRangeKernel()

Example Code

A simple vector-add code will be given here to introduce the basic workflow of OpenCL program. An simple OpenCL program contains a source file *main.c* and a kernel file *kernel.cl*.

main.c

```c
#include <stdio.h>
#include <stdlib.h>

#ifdef __APPLE__ //Mac OSX has a different name for the header file
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

#define MEM_SIZE (128)//suppose we have a vector with 128 elements
#define MAX_SOURCE_SIZE (0x100000)

int main()
{
    //In general Intel CPU and NV/AMD's GPU are in different platforms
    //But in Mac OSX, all the OpenCL devices are in the platform "Apple"
    cl_platform_id platform_id = NULL;
    cl_device_id device_id = NULL;
    cl_context context = NULL;
    cl_command_queue command_queue = NULL; //"stream" in CUDA
    cl_mem memobj = NULL;//device memory
    cl_program program = NULL; //cl_prgram is a program executable created from the source or binary
    cl_kernel kernel = NULL; //kernel function
    cl_uint ret_num_devices;
    cl_uint ret_num_platforms;
    cl_int ret; //accepts return values for APIs

    float mem[MEM_SIZE]; //alloc memory on host(CPU) ram

    //OpenCL source can be placed in the source code as text strings or read from another file.
    FILE *fp;
    const char fileName[] = "./kernel.cl";
    size_t source_size;
    char *source_str;
    cl_int i;

    // read the kernel file into ram
    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 );

    //initialize the mem with 1,2,3...,n
    for( i = 0; i < MEM_SIZE; i++ ) {
        mem[i] = i;
    }

    //get the device info
    ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
    ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, &ret_num_devices);

    //create context on the specified device
```

```
    context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);

    //create the command_queue (stream)
    command_queue = clCreateCommandQueue(context, device_id, 0, &ret);

    //alloc mem on the device with the read/write flag
    memobj = clCreateBuffer(context, CL_MEM_READ_WRITE, MEM_SIZE * sizeof(float), NULL, &ret);

    //copy the memory from host to device, CL_TRUE means blocking write/read
    ret = clEnqueueWriteBuffer(command_queue, memobj, CL_TRUE, 0, MEM_SIZE * sizeof(float), mem, 0, NULL, NULL);

    //create a program object for a context
    //load the source code specified by the text strings into the program object
    program = clCreateProgramWithSource(context, 1, (const char **)&source_str, (const size_t *)&source_size, &ret);

    //build (compiles and links) a program executable from the program source or binary
    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    //create a kernel object with specified name
    kernel = clCreateKernel(program, "vecAdd", &ret);

    //set the argument value for a specific argument of a kernel
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj);

    //define the global size and local size (grid size and block size in CUDA)
    size_t global_work_size[3] = {MEM_SIZE, 0, 0};
    size_t local_work_size[3]  = {MEM_SIZE, 0, 0};

    //Enqueue a command to execute a kernel on a device ("1" indicates 1-dim work)
    ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global_work_size, local_work_size, 0, NULL, NULL);

    //copy memory from device to host
    ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0, MEM_SIZE * sizeof(float), mem, 0, NULL, NULL);

    //print out the result
    for(i=0; i<MEM_SIZE; i++) {
        printf("mem[%d] : %.2f\n", i, mem[i]);
    }

    //clFlush only guarantees that all queued commands to command_queue get issued to the appropriate device
    //There is no guarantee that they will be complete after clFlush returns
    ret = clFlush(command_queue);
    //clFinish blocks until all previously queued OpenCL commands in command_queue are issued to the associated device and have completed.
    ret = clFinish(command_queue);
    ret = clReleaseKernel(kernel);
    ret = clReleaseProgram(program);
    ret = clReleaseMemObject(memobj);//free memory on device
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);

    free(source_str);//free memory on host

    return 0;
}
```

kernel.cl

```
__kernel void vecAdd(__global float* a)
{
    int gid = get_global_id(0);// in CUDA = blockIdx.x * blockDim.x + threadIdx.x

    a[gid] += a[gid];
}
```

Atomic operations on floating point numbers

CUDA has atomicAdd() for floating numbers, but OpenCL doesn't have it. The only atomic function that can work on floating number is atomic_cmpxchg(). According to Atomic operations and floating point numbers in OpenCL (http://simpleopencl.blogspot.ca/2013/05/atomic-operations-and-floats-in-opencl.html), you can serialize the memory access like it is done in the next code:

```
float sum=0;
void atomic_add_global(volatile global float *source, const float operand) {
    union {
        unsigned int intVal;
        float floatVal;
    } newVal;
    union {
        unsigned int intVal;
        float floatVal;
    } prevVal;

    do {
        prevVal.floatVal = *source;
        newVal.floatVal = prevVal.floatVal + operand;
    } while (atomic_cmpxchg((volatile global unsigned int *)source, prevVal.intVal, newVal.intVal) != prevVal.intVal);
}
```

First function works on global memory the second one work on the local memory.

```
float sum=0;
void atomic_add_local(volatile local float *source, const float operand) {
    union {
        unsigned int intVal;
        float floatVal;
    } newVal;

    union {
        unsigned int intVal;
```

```
        float floatVal;
    } prevVal;

    do {
        prevVal.floatVal = *source;
        newVal.floatVal = prevVal.floatVal + operand;
    } while (atomic_cmpxchg((volatile local unsigned int *)source, prevVal.intVal, newVal.intVal) != prevVal.intVal);
}
```

A faster approch is based on the discuss in CUDA developer forums [1] (https://devtalk.nvidia.com/default/topic/458062/atomicadd-float-float-atomicmul-float-float-/)

```
inline void atomicAdd_f(__global float* address, float value)
{
    float old = value;

    while ((old = atomic_xchg(address, atomic_xchg(address, 0.0f)+old))!=0.0f);

}
```