# CUDA转换为OpenCL

https://www.sharcnet.ca/help/index.php/Porting\_CUDA\_to\_OpenCL

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.

也就是说，OpenCL也可以方便地转换为CUDA。

## 目录

* [1Hardware Terminology](https://www.sharcnet.ca/help/index.php/Porting_CUDA_to_OpenCL#Hardware_Terminology)
* [2Qualifiers for Kernel Functions](https://www.sharcnet.ca/help/index.php/Porting_CUDA_to_OpenCL#Qualifiers_for_Kernel_Functions)
* [3Kernels Indexing](https://www.sharcnet.ca/help/index.php/Porting_CUDA_to_OpenCL#Kernels_Indexing)
* [4Kernels Synchronization](https://www.sharcnet.ca/help/index.php/Porting_CUDA_to_OpenCL#Kernels_Synchronization)
* [5API Calls](https://www.sharcnet.ca/help/index.php/Porting_CUDA_to_OpenCL#API_Calls)
* [6Example Code](https://www.sharcnet.ca/help/index.php/Porting_CUDA_to_OpenCL#Example_Code)
* [7Atomic operations on floating point numbers](https://www.sharcnet.ca/help/index.php/Porting_CUDA_to_OpenCL#Atomic_operations_on_floating_point_numbers)

## 硬件术语

|  |  |
| --- | --- |
| **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() |

## 代码示例

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

#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];

}

## 浮点数的原子操作

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))!=

http://simpleopencl.blogspot.com/2013/05/atomic-operations-and-floats-in-opencl.html