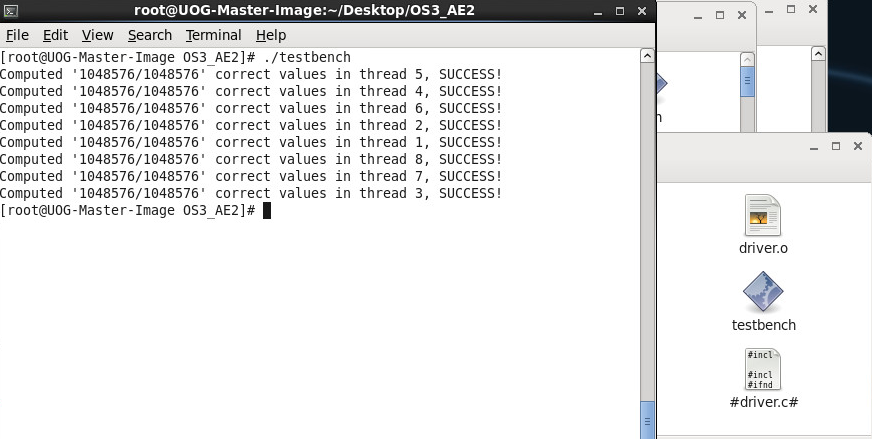
OS3 AE2 - Output



OS3 AE2 - Driver.c Implementation

#include "driver.h"

#include <stdio.h>

#ifndef OSX

#define CL\_USE\_DEPRECATED\_OPENCL\_1\_1\_APIS

#include <CL/cl.h>

#else

#include <OpenCL/opencl.h>

#endif

////////////////////////////////////////////////////////////////////////////////

CLObject\* init\_driver() {

CLObject\* ocl = (CLObject\*)malloc(sizeof(CLObject));

int err; // error code returned from api calls

unsigned int status[1] = { 0 }; // number of correct results returned

size\_t global; // global domain size for our calculation

size\_t local; // local domain size for our calculation

cl\_device\_id device\_id; // compute device id

cl\_context context; // compute context

cl\_command\_queue command\_queue; // compute command queue

cl\_program program; // compute program

cl\_kernel kernel; // compute kernel

cl\_mem input1, input2; // device memory used for the input array

cl\_mem output, status\_buf; // device memory used for the output array

FILE\* programHandle;

size\_t programSize;

char \*programBuffer;

cl\_uint nplatforms;

err = clGetPlatformIDs(0, NULL, &nplatforms);

if (err != CL\_SUCCESS) {

fprintf(stderr, "Error: Failed to get number of platform: %d!\n", err);

exit(EXIT\_FAILURE);

}

// Now ask OpenCL for the platform IDs:

cl\_platform\_id\* platforms = (cl\_platform\_id\*)malloc(sizeof(cl\_platform\_id) \* nplatforms);

err = clGetPlatformIDs(nplatforms, platforms, NULL);

if (err != CL\_SUCCESS) {

fprintf(stderr, "Error: Failed to get platform IDs: %d!\n", err);

exit(EXIT\_FAILURE);

}

#ifdef GPU

err = clGetDeviceIDs(platforms[0], CL\_DEVICE\_TYPE\_GPU, 1, &device\_id, NULL);

#else

err = clGetDeviceIDs(platforms[0], CL\_DEVICE\_TYPE\_CPU, 1, &device\_id, NULL);

#endif

if (err != CL\_SUCCESS)

{

fprintf(stderr, "Error: Failed to create a device group: %d!\n", err);

exit(EXIT\_FAILURE);

}

// Create a compute context

//

context = clCreateContext(0, 1, &device\_id, NULL, NULL, &err);

if (!context)

{

fprintf(stderr, "Error: Failed to create a compute context: %d!\n", err);

exit(EXIT\_FAILURE);

}

// Create a command command\_queue

//

command\_queue = clCreateCommandQueue(context, device\_id, 0, &err);

if (!command\_queue)

{

fprintf(stderr, "Error: Failed to create a command command\_queue: %d!\n", err);

exit(EXIT\_FAILURE);

}

// get size of kernel source

programHandle = fopen("./firmware.cl", "r");

fseek(programHandle, 0, SEEK\_END);

programSize = ftell(programHandle);

rewind(programHandle);

// read kernel source into buffer

programBuffer = (char\*)malloc(programSize + 1);

programBuffer[programSize] = '\0';

fread(programBuffer, sizeof(char), programSize, programHandle);

fclose(programHandle);

// create program from buffer

program = clCreateProgramWithSource(context, 1, (const char\*\*)&programBuffer, &programSize, &err);

free(programBuffer);

if (!program)

{

fprintf(stderr, "Error: Failed to create compute program: %d!\n", err);

exit(EXIT\_FAILURE);

}

// Build the program executable

//

err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

if (err != CL\_SUCCESS)

{

size\_t len;

char buffer[2048];

fprintf(stderr, "Error: Failed to build program executable: %d!\n", err);

clGetProgramBuildInfo(program, device\_id, CL\_PROGRAM\_BUILD\_LOG, sizeof(buffer), buffer, &len);

fprintf(stderr, "%s\n", buffer);

exit(EXIT\_FAILURE);

}

// Create the compute kernel in the program we wish to run

//

kernel = clCreateKernel(program, "firmware", &err);

if (!kernel || err != CL\_SUCCESS)

{

fprintf(stderr, "Error: Failed to create compute kernel: %d!\n", err);

exit(EXIT\_FAILURE);

}

ocl->context = context;

ocl->command\_queue = command\_queue;

ocl->kernel = kernel;

ocl->program = program;

ocl->device\_id = device\_id;

//===============================================================================================================================================================

// START of assignment code section

// create a mutex lock

pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;

ocl->device\_lock = lock;

// END of assignment code section

//===============================================================================================================================================================

return ocl;

}

int shutdown\_driver(CLObject\* ocl) {

int err = clReleaseProgram(ocl->program);

if (err != CL\_SUCCESS) {

fprintf(stderr, "Error: Failed to release Program: %d!\n", err);

exit(EXIT\_FAILURE);

}

err = clReleaseKernel(ocl->kernel);

if (err != CL\_SUCCESS) {

fprintf(stderr, "Error: Failed to release Kernel: %d!\n", err);

exit(EXIT\_FAILURE);

}

err = clReleaseCommandQueue(ocl->command\_queue);

if (err != CL\_SUCCESS) {

fprintf(stderr, "Error: Failed to release Command Queue: %d!\n", err);

exit(EXIT\_FAILURE);

}

err = clReleaseContext(ocl->context);

if (err != CL\_SUCCESS) {

fprintf(stderr, "Error: Failed to release Context: %d!\n", err);

exit(EXIT\_FAILURE);

}

//===============================================================================================================================================================

// START of assignment code section

// release device

err = clReleaseDevice(ocl->device\_id);

if (err != CL\_SUCCESS) {

fprintf(stderr, "Error: Failed to release device: %d!\n", err);

exit(EXIT\_FAILURE);

}

// destroy lock

pthread\_mutex\_destroy(&ocl->device\_lock);

// END of assignment code section

//===============================================================================================================================================================

free(ocl);

return 0;

}

////////////////////////////////////////////////////////////////////////////////

int run\_driver(CLObject\* ocl, unsigned int buffer\_size, int\* input\_buffer\_1, int\* input\_buffer\_2, int\* output\_buffer) {

long long unsigned int tid = ocl->thread\_num;

#if VERBOSE\_MT>2

printf("run\_driver thread: %llu\n", tid);

#endif

int err; // error code returned from api calls

int status[1] = { -1 }; // number of correct results returned

unsigned int max\_iters;

max\_iters = MAX\_ITERS;

size\_t global; // global domain size for our calculation

size\_t local; // local domain size for our calculation

cl\_mem input1, input2; // device memory used for the input array

cl\_mem output, status\_buf; // device memory used for the output array

// Get the maximum work group size for executing the kernel on the device

err = clGetKernelWorkGroupInfo(ocl->kernel, ocl->device\_id, CL\_KERNEL\_WORK\_GROUP\_SIZE, sizeof(local), &local, NULL);

if (err != CL\_SUCCESS) {

fprintf(stderr, "Error: Failed to retrieve kernel work group info! %d\n", err);

exit(EXIT\_FAILURE);

}

global = buffer\_size; // create as meany threads on the device as there are elements in the array

//===============================================================================================================================================================

// START of assignment code section

// You must make sure the driver is thread-safe by using the appropriate POSIX mutex operations

// You must also check the return value of every API call and handle any errors

// Create the buffer objects to link the input and output arrays in device memory to the buffers in host memory

input1 = clCreateBuffer(ocl->context, CL\_MEM\_READ\_ONLY, sizeof(float) \* buffer\_size, NULL, NULL);

input2 = clCreateBuffer(ocl->context, CL\_MEM\_READ\_ONLY, sizeof(float) \* buffer\_size, NULL, NULL);

output = clCreateBuffer(ocl->context, CL\_MEM\_WRITE\_ONLY, sizeof(float) \* buffer\_size, NULL, NULL);

status\_buf = clCreateBuffer(ocl->context, CL\_MEM\_WRITE\_ONLY, sizeof(float) \* 1, NULL, NULL);

if (!input1 || !input2 || !output || !status\_buf)

{

fprintf(stderr, "Error: Failed to allocate buffer objects to device memory!\n");

exit(EXIT\_FAILURE);

}

// Write the data in input arrays into the device memory

pthread\_mutex\_lock(&ocl->device\_lock);

err = clEnqueueWriteBuffer(ocl->command\_queue, input1, CL\_TRUE, 0, sizeof(float) \* buffer\_size, input\_buffer\_1, 0, NULL, NULL);

if (err != CL\_SUCCESS)

{

fprintf(stderr, "Error: Failed to write to input array %d to device memory!\n", err);

exit(EXIT\_FAILURE);

}

err = clEnqueueWriteBuffer(ocl->command\_queue, input2, CL\_TRUE, 0, sizeof(float) \* buffer\_size, input\_buffer\_2, 0, NULL, NULL);

if (err != CL\_SUCCESS)

{

fprintf(stderr, "Error: Failed to write to input array %d to device memory!\n", err);

exit(EXIT\_FAILURE);

}

// Set the arguments to our compute kernel

err = clSetKernelArg(ocl->kernel, 0, sizeof(cl\_mem), &input1);

err |= clSetKernelArg(ocl->kernel, 1, sizeof(cl\_mem), &input2);

err |= clSetKernelArg(ocl->kernel, 2, sizeof(cl\_mem), &output);

err |= clSetKernelArg(ocl->kernel, 3, sizeof(cl\_mem), &status\_buf);

err |= clSetKernelArg(ocl->kernel, 4, sizeof(unsigned int), &buffer\_size);

if (err != CL\_SUCCESS)

{

fprintf(stderr, "Error: Failed to set kernel arguments! %d\n", err);

exit(EXIT\_FAILURE);

}

// Execute the kernel, i.e. tell the device to process the data using the given global and local ranges

err = clEnqueueNDRangeKernel(ocl->command\_queue, ocl->kernel, 1, NULL, &global, &local, 0, NULL, NULL);

if (err != CL\_SUCCESS)

{

fprintf(stderr, "Error: Failed to execute kernel! %d\n", err);

exit(EXIT\_FAILURE);

}

pthread\_mutex\_unlock(&ocl->device\_lock);

// Wait for the command command\_queue to get serviced before reading back results. This is the device sending an interrupt to the host

clFinish(ocl->command\_queue);

// Check the status

err = clEnqueueReadBuffer(ocl->command\_queue, status\_buf, CL\_TRUE, 0, sizeof(float) \* 1, status, 0, NULL, NULL);

if (err != CL\_SUCCESS) {

fprintf(stderr, "Error: Failed to read status %d!\n", err);

exit(EXIT\_FAILURE);

}

// When the status is 0, read back the results from the device to verify the output

if (status[0] == 0) {

err = clEnqueueReadBuffer(ocl->command\_queue, output, CL\_TRUE, 0, sizeof(float) \* buffer\_size, output\_buffer, 0, NULL, NULL);

if (err != CL\_SUCCESS)

{

fprintf(stderr, "Error: Failed to read output array %d!\n", err);

exit(EXIT\_FAILURE);

}

}

// Shutdown and cleanup

err = clReleaseMemObject(input1);

if (err != CL\_SUCCESS) {

fprintf(stderr, "Error: Failed to release memory object: %d!/n", err);

exit(EXIT\_FAILURE);

}

err = clReleaseMemObject(input2);

if (err != CL\_SUCCESS) {

fprintf(stderr, "Error: Failed to release memory object: %d!/n", err);

exit(EXIT\_FAILURE);

}

err = clReleaseMemObject(output);

if (err != CL\_SUCCESS) {

fprintf(stderr, "Error: Failed to release memory object: %d!/n", err);

exit(EXIT\_FAILURE);

}

err = clReleaseMemObject(status\_buf);

if (err != CL\_SUCCESS) {

fprintf(stderr, "Error: Failed to release memory object: %d!/n", err);

exit(EXIT\_FAILURE);

}

// END of assignment code section

//===============================================================================================================================================================

return \*status;

}