Hello Vector

OpenCL Tutorial
Part 1.2

Guillermo Marcus guillermo.marcus@gmail.com

Hello Vector

Takes two vectors, A and B, and adds them. Writes the result in a third vector C.

Parallelizes as one add per thread

Files

Makefile

Builds the executable files

oclHelloVector.cpp

Host code

oclHelloVector.ocl

GPU Kernel code

oclHelloVector.ocl

```
__kernel void helloVector(__global const int *A, __global const int *B, __global int *C)
{
	int i = get_global_id(0);

	C[i] = A[i] + B[i];
	return;
}
```

oclHelloVector.cpp

```
/**
* This is the Main file for the OpenCL Hello Vector Program.
*
* \author: Guillermo Marcus
* \date: 2011-11-09
*/

#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;

std::string loadProgram(const char *filename);

int main(int argc, char **argv)
{
	const int VECT_SIZE = 100000;
	int *A = new int[VECT_SIZE];
	int *B = new int[VECT_SIZE];
	int *C = new int[VECT_SIZE];
	bool error = false;

	// Fill test data
	for( int i=0; i < VECT_SIZE; i++ ) {
		A[i] = i;
		B[i] = VECT_SIZE - i;
		C[i] = 0;
	}

	try {
		// Check available platforms
		std::vector<Platform> platforms;
		Platform::get( &platforms );

		// Create context
		cl_context_properties cps[3] = {
			CL_CONTEXT_PLATFORM,
			(cl_context_properties)(platforms[0])(),
			0
		};
		Context context( CL_DEVICE_TYPE_GPU, cps );

		// Devices available in this platform
		std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

		// query name of device
		STRING_CLASS name;
		devices[0].getInfo( CL_DEVICE_NAME, &name );
		cout << "Will use this device: " << name << endl;

		// Read kernel source and build a program
		std::string kernel_source = loadProgram("oclHelloVector.ocl");

		Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

		Program program = Program(context, source);
		program.build(devices);

		// Create a kernel
		Kernel kernel(program, "helloVector");

		// Create command queue using the first device
		CommandQueue queue = CommandQueue( context, devices[0], 0 );

		// Create Memory buffers
		Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE *
sizeof(int) );

		// Copy buffers to device
		queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
		queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

		// Create kernel specification (ND range)
		NDRange global(VECT_SIZE);
		NDRange local(1);

		// Set kernel arguments
		kernel.setArg(0, bufA);
		kernel.setArg(1, bufB);
		kernel.setArg(2, bufC);

		// Run kernel
		Event event;
		queue.enqueueNDRangeKernel(	kernel,	NullRange, global, local );

		// Copy result buffer from device
		queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

	}
	catch (Error err) {
		cerr << "ERROR: " << err.what()	<< "(" << err.err() << ")" << endl;
		error = true;
	}

	// Compare results
	bool test = true;
	for( int i=0; i < VECT_SIZE; i++ ) {
		if (C[i] != VECT_SIZE)
			test = false;
	}
	if (test)
		cout << "Test PASSED!" << endl;
	else
		cout << "Test FAILED!" << endl;

	delete A;delete B;delete C;
	A = NULL;B = NULL;C = NULL;

	if (error)
		return EXIT_FAILURE;
	else
		return EXIT_SUCCESS;
}

std::string loadProgram(const char *filename) {
	ifstream src(filename);
	std::string src_code( istreambuf_iterator<char>(src), (istreambuf_iterator<char>()) );

	return src_code;
}
```

```
/**
* This is the Main file for the OpenCL Hello Vector Program.
*
* \author: Guillermo Marcus
* \date: 2011-11-09
*/

#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;

std::string loadProgram(const char *filename);

int main(int argc, char **argv)
{
	const int VECT_SIZE = 100000;
	int *A = new int[VECT_SIZE];
	int *B = new int[VECT_SIZE];
	int *C = new int[VECT_SIZE];
	bool error = false;

	// Fill test data
	for( int i=0; i < VECT_SIZE; i++ ) {
		A[i] = i;
		B[i] = VECT_SIZE - i;
		C[i] = 0;
	}

	try {
		// Check available platforms
		std::vector<Platform> platforms;
		Platform::get( &platforms );

		// Create context
		cl_context_properties cps[3] = {
			CL_CONTEXT_PLATFORM,
			(cl_context_properties)(platforms[0])(),
			0
		};
		Context context( CL_DEVICE_TYPE_GPU, cps );

		// Devices available in this platform
		std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

		// query name of device
		STRING_CLASS name;
		devices[0].getInfo( CL_DEVICE_NAME, &name );
		cout << "Will use this device: " << name << endl;

		// Read kernel source and build a program
		std::string kernel_source = loadProgram("oclHelloVector.ocl");

		Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

		Program program = Program(context, source);
		program.build(devices);

		// Create a kernel
		Kernel kernel(program, "helloVector");

		// Create command queue using the first device
		CommandQueue queue = CommandQueue( context, devices[0], 0 );

		// Create Memory buffers
		Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

		// Copy buffers to device
		queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
		queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

		// Create kernel specification (ND range)
		NDRange global(VECT_SIZE);
		NDRange local(1);

		// Set kernel arguments
		kernel.setArg(0, bufA);
		kernel.setArg(1, bufB);
		kernel.setArg(2, bufC);

		// Run kernel
		Event event;
		queue.enqueueNDRangeKernel( kernel, NullRange, global, local );

		// Copy result buffer from device
		queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );
	}
	catch (Error err) {
		cerr << "ERROR: " << err.what() << "(" << err.err() << ")" << endl;
		error = true;
	}

	// Compare results
	bool test = true;
	for( int i=0; i < VECT_SIZE; i++ ) {
		if (C[i] != VECT_SIZE)
			test = false;
	}
	if (test)
		cout << "Test PASSED!" << endl;
	else
		cout << "Test FAILED!" << endl;

	delete A;delete B;delete C;
	A = NULL;B = NULL;C = NULL;

	if (error)
		return EXIT_FAILURE;
	else
		return EXIT_SUCCESS;
}

std::string loadProgram(const char *filename) {
	ifstream src(filename);
	std::string src_code( istreambuf_iterator<char>(src), (istreambuf_iterator<char>()) );

	return src_code;
}
```

Host code Structure

Headers

Initialize OpenCL

- Create Buffers
- Copy data in host -> GPU
- Configure and Launch Kernel
- Copy data out GPU -> host

Finalize OpenCL

Initialization

Select **Platform**

A platform is any environment that implements OpenCL

Select **Device**(s) from Platform

A device that executes OpenCL

Create **Context** for Device(s)

Binds a device with memory and compiled program kernels

Create a **Queue**

Orders Kernel launches in a device

```
// Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );


// Create context
        cl_context_properties cps[3] = {
            CL_CONTEXT_PLATFORM,
            (cl_context_properties)(platforms[0])(),
            0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

// Devices available in this platform
        std::vector<Device> devices = context.
getInfo<CL_CONTEXT_DEVICES>();

// query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

// Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );
```

Initialization of the Kernels

Load a **Source**

- Source code for the Kernel(s)

Build for Device(s)

- Compile the Source into binaries

Create a **Kernel** for launch

- Only Kernel functions can be invoked from the host

```
// Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");
```

Create Buffers

We need to explicitly allocate the GPU Global Memory before using it.

This creates an object in the host that represents the buffer in the GPU.

```
// Create Memory buffers
      Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
      Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
      Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int)
);
```

Copy Data from host to GPU

We enqueue the Copy operation into the Command Queue we created earlier.

This is a **write** operation (we write to the GPU).

```
// Copy buffers to device
    queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
    queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );
```

Configure and Launch Kernel

```
// Create kernel specification (ND range)
    NDRange global(VECT_SIZE);
    NDRange local(1);

// Set kernel arguments
    kernel.setArg(0, bufA);
    kernel.setArg(1, bufB);
    kernel.setArg(2, bufC);

// Run kernel
    Event event;
    queue.enqueueNDRangeKernel(    kernel,    NullRange, global, local );
```

Copy data from GPU

```
// Copy result buffer from device
    queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );
```

Let's begin!

Now let's try this in the Amazon Cloud!

Login into the cloud with your account data
Compile the helloVector program
Try it!

Task 1

Let OpenCL define the number of threads per Work Group automatically.

To do that, change the **local** parameter of the enqueue Kernel for a NullRange.

This only works for unidimensional arrays

Solution

Task 2

Define the number of threads per Work Group manually.

Fix the number of threads por Work Group to be 64, and modify the host and the kernel code accordingly!

Solution

You have more time?

Change the data types to float, double .. see what happens!

Try doing something more complex with each thread, maybe a more complex function than just add?

Now do something that requires more data IO. Average the 3,5,10 nearby values in a vector position.