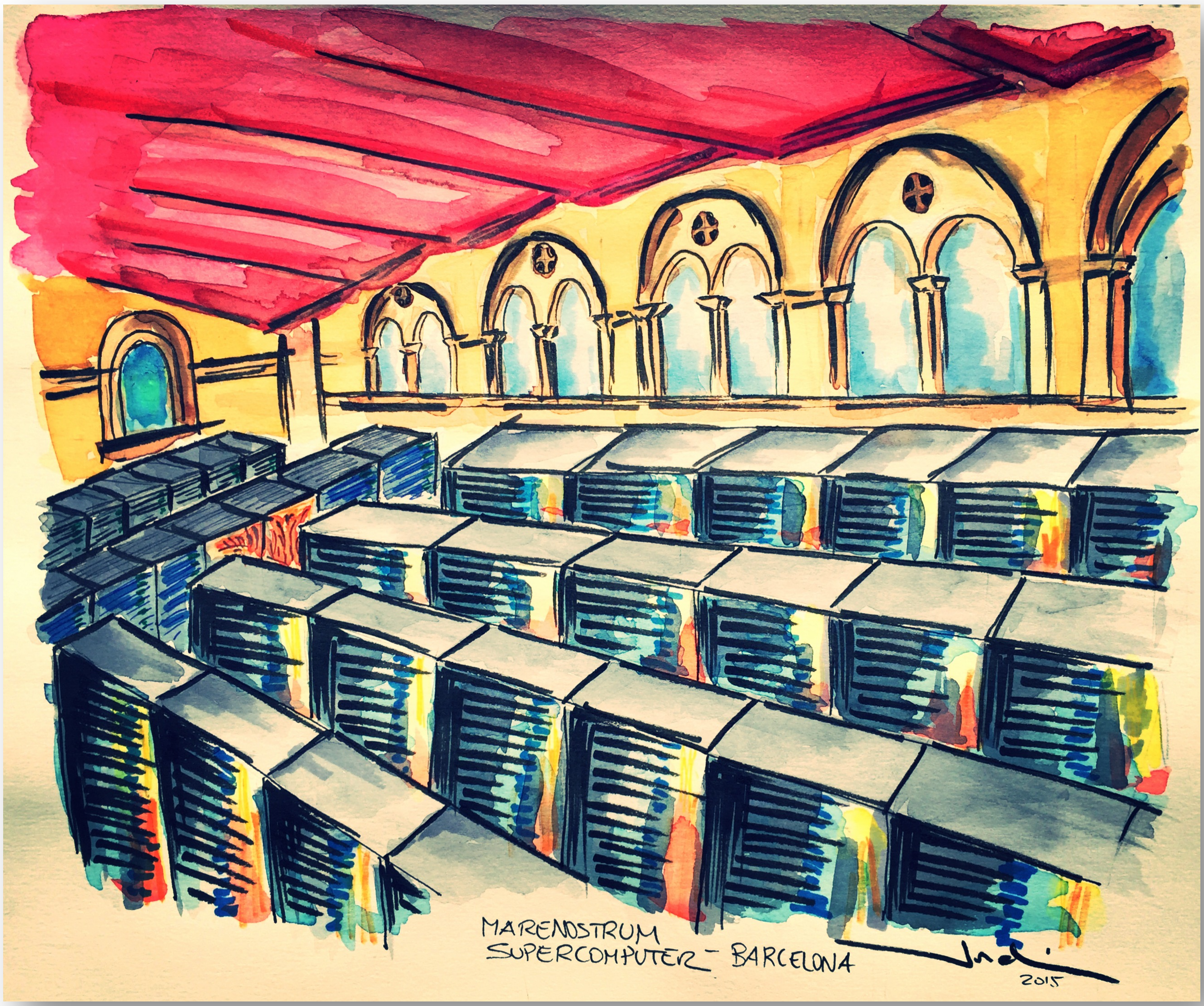
Hands-on 8: Getting Started with Heterogeneous Computing at Marenostrum III



SUPERCOMPUTERS ARCHITECTURE

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS

Barcelona, Fall 2015

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Hands-on 8

Getting Started with Heterogeneous Computing at Marenostrum III

SUPERCOMPUTERS ARCHITECTURE

Master in innovation and research in informatics

(Specialization High Performance Computing)

UPC Barcelona Tech & Barcelona Supercomputing Center

Version 2.0 - 30 November 2015

Authors: Alejandro Fernández, alejandro.fernandez@bsc.es

Jordi Torres, jordi.torres@bsc.es

Table of Contents

1 Hands-on description

This Hands-On will get you started with the basics of accelerator-based programming, which consists of a set of architectural and software solutions devoted to the efficient exploitation of specialized computation accelerators such as GPUs or Intel’s Xeon Phi.

First, we will present the OpenCL programming model, which is an open standard for heterogeneous programming. Then, we will see how to use the MareNostrum nodes equipped with the Xeon Phi accelerators to run simple examples. Afterwards, we will see a simple example and you will be asked to perform simple guided modifications. Finally, you will create a job script to submit your programs to the MIC queue and some optional work will be given for those more interested in OpenCL.

2 OpenCL basics

2.1 Why OpenCL?

OpenMP is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. OpenCL (Open Computing Language) greatly improves speed and responsiveness for a wide spectrum of applications in numerous market categories from gaming to scientific software by enabling access to massively parallel and specialized computing devices.

As we discussed in the theory class, OpenCL is implemented by hardware vendors for each platform and is commonly **deployed as a shared object library** that applications use from their corresponding APIs. The most common OpenCL API is written in C and is the most common way to execute OpenCL code in accelerators.

The source code of OpenCL applications is split between accelerator devices and the host server. For example, for GPUs, the host code is run on the CPUs like a regular non-accelerated application. Then, once the application starts running on the CPUs, the host application calls the device-specific compiler to compile the device code. Afterwards, the host sends the data to the devices (remember accelerators usually do not have access to the main memory of the machine, they have their own). Once the accelerator has the data and the host has compiled the application for it, the program binary is loaded into the device and starts running.

Although OpenCL presents many advantages regarding the use of accelerators, its nature implies an important deal of complexity in the application's code. This means that **porting a regular CPU-based application to an OpenCL application is by far not straightforward** but rather implies an important refactoring of the application and probably the algorithms it implements. **Nevertheless, later in this lab we will present OmpSs BSC programming model, which leverages a significant amount of complexity, making heterogeneous programming more accessible and productive.**

NOTE: The support of OpenCL on the Xeon Phi accelerators of MN3 is still experimental. Therefore, we will run the OpenCL programs using the regular Xeon CPUs, but just replacing CL_DEVICE_TYPE_CPU by CL_DEVICE_TYPE_ ACCELERATOR would make the code run in the Xeon Phi once there is stable support for it (we hope soon! ☺).

2.2 Using MareNostrum III's MIC nodes

To access the MIC you first need to request one of the host nodes. You may request interactive access through queue 'mic-interactive' as follows:

```
% bsub -Is -q mic-interactive -W 01:00 /bin/bash
```

Note: MareNostrum III has Intel Xeon Phi accelerators accessible only for users granted to have access to them. Plesase check if you are able to see the queues 'mic' and 'mic-interactive' in the output of bsc_queues.

There are 42 nodes with Xeon Phi, 6 nodes available for interactive usage and 36 nodes for off-line executions. These nodes have:

- 2x Intel SandyBridge-EP E5–2670/1600 20M 8-core at 2.6 GHz
- 2x Xeon Phi 5110 P
- 8x8GB DDR3–1600 DIMM (4GB/core)

Right now the communication network between the nodes is Infiniband, but between the MICs themselves is 10GE.

Once the interactive session starts, your shell will switch to a MIC node, which contains 2 Xeon CPUs and 2 Xeon Phi accelerators. In the case of a Xeon Phi, you can also open the shell in the accelerator itself and use it as a regular multicore, **but we will run our applications on the CPUs and use the Xeon Phis as accelerators via OpenCL**.

Now let's try a program that extracts information of the OpenCL platforms and devices, which is useful for identifying specific properties of your platform. You can download the code from https://opencl-book-samples.googlecode.com/svn-history/r18/trunk/src/Chapter_3/OpenCLInfo/OpenCLInfo.cpp (also included as a Annex in this hands-on).

Then, you can compile with:

```
% g++ OpenCLInfo.cpp –o OpenCLInfo -lOpenCL
```

Additionally, you will need to use the latest OpenCL library to your library path in order to have support for the Xeon Phi. You can do this by setting the LD_LIBRARY_PATH variable in your system by editing the .bashrc file in your home directory:

```
% cd
% echo "export LD_LIBRARY_PATH=/opt/intel/opencl-1.2-3.0.67279/lib64:$LD_LIB
  RARY_PATH" >> .bashrc
```

Exercise 1: Download the OpenCL Info program and execute it on a MIC host. How many devices are in the platform? Are all of them Xeon Phi accelerators? How much memory (in GB) does each Xeon Phi have? Which is the maximum workgroup size of a Xeon Phi?

2.3 Getting started

As discussed in class, the workflow of a typical OpenCL application is as follows:

1. Select an OpenCL platform
2. Select one or more devices on the platform
3. Create queues to submit commands to each device
4. Enqueue commands on the queues

Exercise 2: Given the following OpenCL application, where is the accelerator code compiled? When is data moved to/from the accelerator? When are the accelerator programs executed?

```
#include <CL/cl.h>
#include <stdio.h>
#include <string.h>

// Application constants
const size_t local = 1024;
const size_t global = local*128;

const char code_init[] = "__kernel void init(__global float *out) { \
                const uint index = get_global_id(0); \
                out[index] = 1.0f; \
        }";
const char code_add[] = "__kernel void add(__global float *inout, \
        __global float *in) { \
                const uint index = get_global_id(0); \
                inout[index] = inout[index] + in[index]; \
        }";

int main(int argc, char** argv) {
      cl_platform_id platform;
      cl_device_id devices[1];

      clGetPlatformIDs(1, &platform, NULL);
      clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, devices, NULL);


      cl_context_properties properties[] = {
        CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0};
      cl_context context = clCreateContext(
        properties, 1, devices, NULL, NULL, NULL);
      cl_command_queue cq1 = clCreateCommandQueue(
        context, devices[0], 0, NULL);

      uint buffSizeInBytes = global*sizeof(float);
      float buf[global];
      cl_mem device_buf1 = clCreateBuffer(
        context, CL_MEM_READ_WRITE, buffSizeInBytes, NULL, NULL);
      cl_mem device_buf2 = clCreateBuffer(
        context, CL_MEM_READ_WRITE, buffSizeInBytes, NULL, NULL);

      size_t initsrcsize = strlen(code_init);
      const char *initsrcptr[] = {code_init};
      size_t addsrcsize = strlen(code_add);
      const char *addsrcptr[] = {code_add};

      cl_program initprog=clCreateProgramWithSource(
        context, 1, initsrcptr, &initsrcsize, NULL);
      clBuildProgram(initprog, 0, NULL, "", NULL, NULL);
      cl_program addprog=clCreateProgramWithSource(
        context, 1, addsrcptr, &addsrcsize, NULL);
      clBuildProgram(addprog, 0, NULL, "", NULL, NULL);
      cl_kernel initK = clCreateKernel(initprog, "init", NULL);
      cl_kernel addK = clCreateKernel(addprog, "add", NULL);
```

```
    clSetKernelArg(initK, 0, sizeof(cl_mem), &device_buf1);
    clEnqueueNDRangeKernel(
      cq1, initK, 1, NULL, &global, &local, 0, NULL, NULL);
    clSetKernelArg(initK, 0, sizeof(cl_mem), &device_buf2);
    clEnqueueNDRangeKernel(
      cq1, initK, 1, NULL, &global, &local, 0, NULL, NULL);
    printf("Initializing buffers...\n");
    clFinish(cq1);

    clSetKernelArg(addK, 0, sizeof(cl_mem), &device_buf1);
    clSetKernelArg(addK, 1, sizeof(cl_mem), &device_buf2);
    clEnqueueNDRangeKernel(
      cq1, addK, 1, NULL, &global, &local, 0, NULL, NULL);
    clEnqueueReadBuffer(cq1,
      device_buf1, CL_FALSE, 0, buffSizeInBytes, buf, 0, NULL, NULL);
    printf("Adding buffers...\n");
    clFinish(cq1);

    return 0;
}
```

Exercise 3: Compile and run the program of Exercise 2. How many elements are added in parallel when the “add” kernel is run? Hint: Even if the maximum workgroup size is 1024 in both the Xeon CPU and the Xeon Phis, it may have less compute units.

3 Programming with OpenCL

Now you are going to develop an OpenCL application that calculates the dot product of two vectors.

3.1 Dot product in OpenCL

Exercise 4

Using the code in Exercise 2, create an OpenCL program that calculates the element-wise product with a kernel. You can follow this steps:

1. Copy the initialization code of Exercise 2, replace the “add” kernel with a “prod” kernel.
2. Read the resulting buffer from the OpenCL device.
3. Add all the elements of the buffer
4. Print the result on the terminal

Exercise 5 (OPTIONAL)

Instead of adding all the elements in the CPU (step 3), perform a parallel reduction on the device. You can take a look at the CUDA version here: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

4 Submitting OpenCL batch jobs

Exercise 6 (OPTIONAL) Take a look at the User Guide of MareNostrum III: http://www.bsc.es/support/MareNostrum3-ug.pdf . Create a job script to submit the program you developed in Exercise 4.

5 Report Lab

You have one week to deliver a report with the answers to the exercises and the program you wrote in Exercise 4 (and 5) with its script (Exercise 6).

6 Annex

Code donwsloaded from https://opencl-book-samples.googlecode.com/svn-history/r18/trunk/src/Chapter_3/OpenCLInfo/OpenCLInfo.cpp.

```
//
// Book:      OpenCL(R) Programming Guide
// Authors:   Aaftab Munshi, Benedict Gaster, Dan Ginsburg, Timothy Mattson
// Publisher: Addison-Wesley Professional

// OpenCLInfo.cpp
//
//    This is a simple example that demonstrates use of the clGetInfo* functions,
//    with particular focus on platforms and their associated devices.

#include <iostream>
#include <fstream>
#include <sstream>

#if defined(_WIN32)
#include <malloc.h> // needed for alloca
#endif // _WIN32

#if defined(linux) || defined(__APPLE__) || defined(__MACOSX)
# include <alloca.h>
#endif // linux

#ifdef __APPLE__
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif

///
// Display information for a particular platform.
// Assumes that all calls to clGetPlatformInfo returns
// a value of type char[], which is valid for OpenCL 1.1.
//
void DisplayPlatformInfo(
        cl_platform_id id,
        cl_platform_info name,
        std::string str)
{
        cl_int errNum;
        std::size_t paramValueSize;

        errNum = clGetPlatformInfo(
                id,
                name,
                0,
                NULL,
                &paramValueSize);
        if (errNum != CL_SUCCESS)
        {
                std::cerr << "Failed to find OpenCL platform " << str << "." << std::endl;
                return;
        }

        char * info = (char *)alloca(sizeof(char) * paramValueSize);
        errNum = clGetPlatformInfo(
                id,
                name,
                paramValueSize,
                info,
                NULL);
        if (errNum != CL_SUCCESS)
        {
                std::cerr << "Failed to find OpenCL platform " << str << "." << std::endl;
```

```
		return;
	}

	std::cout << "\t" << str << ":\t" << info << std::endl;
}

template<typename T>
void appendBitfield(T info, T value, std::string name, std::string & str)
{
	if (info & value)
	{
		if (str.length() > 0)
		{
			str.append(" | ");
		}
		str.append(name);
	}
}

///
// Display information for a particular device.
// As different calls to clGetDeviceInfo may return
// values of different types a template is used.
// As some values returned are arrays of values, a templated class is
// used so it can be specialized for this case, see below.
//
template <typename T>
class InfoDevice
{
public:
	static void display(
		cl_device_id id,
		cl_device_info name,
		std::string str)
	{
		cl_int errNum;
		std::size_t paramValueSize;

		errNum = clGetDeviceInfo(
			id,
			name,
			0,
			NULL,
			&paramValueSize);
		if (errNum != CL_SUCCESS)
		{
			std::cerr << "Failed to find OpenCL device info " << str << "." <<
std::endl;
			return;
		}

		T * info = (T *)alloca(sizeof(T) * paramValueSize);
		errNum = clGetDeviceInfo(
			id,
			name,
			paramValueSize,
			info,
			NULL);
		if (errNum != CL_SUCCESS)
		{
			std::cerr << "Failed to find OpenCL device info " << str << "." <<
std::endl;
			return;
		}

		// Handle a few special cases
		switch (name)
		{
		case CL_DEVICE_TYPE:
			{
				std::string deviceType;
```

```
                              appendBitfield<cl_device_type>(
                                     *(reinterpret_cast<cl_device_type*>(info)),
                                     CL_DEVICE_TYPE_CPU,
                                     "CL_DEVICE_TYPE_CPU",
                                     deviceType);

                              appendBitfield<cl_device_type>(
                                     *(reinterpret_cast<cl_device_type*>(info)),
                                     CL_DEVICE_TYPE_GPU,
                                     "CL_DEVICE_TYPE_GPU",
                                     deviceType);

                              appendBitfield<cl_device_type>(
                                     *(reinterpret_cast<cl_device_type*>(info)),
                                     CL_DEVICE_TYPE_ACCELERATOR,
                                     "CL_DEVICE_TYPE_ACCELERATOR",
                                     deviceType);

                              appendBitfield<cl_device_type>(
                                     *(reinterpret_cast<cl_device_type*>(info)),
                                     CL_DEVICE_TYPE_DEFAULT,
                                     "CL_DEVICE_TYPE_DEFAULT",
                                     deviceType);

                              std::cout << "\t\t" << str << ":\t" << deviceType <<
std::endl;
                       }
                       break;
               case CL_DEVICE_SINGLE_FP_CONFIG:
                       {
                              std::string fpType;

                              appendBitfield<cl_device_fp_config>(
                                     *(reinterpret_cast<cl_device_fp_config*>(info)),
                                     CL_FP_DENORM,
                                     "CL_FP_DENORM",
                                     fpType);

                              appendBitfield<cl_device_fp_config>(
                                     *(reinterpret_cast<cl_device_fp_config*>(info)),
                                     CL_FP_INF_NAN,
                                     "CL_FP_INF_NAN",
                                     fpType);

                              appendBitfield<cl_device_fp_config>(
                                     *(reinterpret_cast<cl_device_fp_config*>(info)),
                                     CL_FP_ROUND_TO_NEAREST,
                                     "CL_FP_ROUND_TO_NEAREST",
                                     fpType);

                              appendBitfield<cl_device_fp_config>(
                                     *(reinterpret_cast<cl_device_fp_config*>(info)),
                                     CL_FP_ROUND_TO_ZERO,
                                     "CL_FP_ROUND_TO_ZERO",
                                     fpType);

                              appendBitfield<cl_device_fp_config>(
                                     *(reinterpret_cast<cl_device_fp_config*>(info)),
                                     CL_FP_ROUND_TO_INF,
                                     "CL_FP_ROUND_TO_INF",
                                     fpType);

                              appendBitfield<cl_device_fp_config>(
                                     *(reinterpret_cast<cl_device_fp_config*>(info)),
                                     CL_FP_FMA,
                                     "CL_FP_FMA",
                                     fpType);


#ifdef CL_FP_SOFT_FLOAT
                              appendBitfield<cl_device_fp_config>(
                                     *(reinterpret_cast<cl_device_fp_config*>(info)),
                                        CL_FP_SOFT_FLOAT,
                                        "CL_FP_SOFT_FLOAT",
                                        fpType);
#endif

                                std::cout << "\t\t" << str << ":\t" << fpType << std::endl;
                        }
                case CL_DEVICE_GLOBAL_MEM_CACHE_TYPE:
                        {
                                std::string memType;

                                appendBitfield<cl_device_mem_cache_type>(

        *(reinterpret_cast<cl_device_mem_cache_type*>(info)),
                                        CL_NONE,
                                        "CL_NONE",
                                        memType);
                                appendBitfield<cl_device_mem_cache_type>(

        *(reinterpret_cast<cl_device_mem_cache_type*>(info)),
                                        CL_READ_ONLY_CACHE,
                                        "CL_READ_ONLY_CACHE",
                                        memType);

                                appendBitfield<cl_device_mem_cache_type>(

        *(reinterpret_cast<cl_device_mem_cache_type*>(info)),
                                        CL_READ_WRITE_CACHE,
                                        "CL_READ_WRITE_CACHE",
                                        memType);

                                std::cout << "\t\t" << str << ":\t" << memType << std::endl;
                        }
                        break;
                case CL_DEVICE_LOCAL_MEM_TYPE:
                        {
                                std::string memType;

                                appendBitfield<cl_device_local_mem_type>(

        *(reinterpret_cast<cl_device_local_mem_type*>(info)),
                                        CL_GLOBAL,
                                        "CL_LOCAL",
                                        memType);

                                appendBitfield<cl_device_local_mem_type>(

        *(reinterpret_cast<cl_device_local_mem_type*>(info)),
                                        CL_GLOBAL,
                                        "CL_GLOBAL",
                                        memType);

                                std::cout << "\t\t" << str << ":\t" << memType << std::endl;
                        }
                        break;
                case CL_DEVICE_EXECUTION_CAPABILITIES:
                        {
                                std::string memType;

                                appendBitfield<cl_device_exec_capabilities>(

        *(reinterpret_cast<cl_device_exec_capabilities*>(info)),
                                        CL_EXEC_KERNEL,
                                        "CL_EXEC_KERNEL",
                                        memType);

                                appendBitfield<cl_device_exec_capabilities>(

        *(reinterpret_cast<cl_device_exec_capabilities*>(info)),
                                        CL_EXEC_NATIVE_KERNEL,
                                        "CL_EXEC_NATIVE_KERNEL",
                                        memType);
                        std::cout << "\t\t" << str << ":\t" << memType << std::endl;
                    }
                    break;
                case CL_DEVICE_QUEUE_PROPERTIES:
                    {
                        std::string memType;

                        appendBitfield<cl_device_exec_capabilities>(

        *(reinterpret_cast<cl_device_exec_capabilities*>(info)),
                            CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
                            "CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE",
                            memType);

                        appendBitfield<cl_device_exec_capabilities>(

        *(reinterpret_cast<cl_device_exec_capabilities*>(info)),
                            CL_QUEUE_PROFILING_ENABLE,
                            "CL_QUEUE_PROFILING_ENABLE",
                            memType);

                        std::cout << "\t\t" << str << ":\t" << memType << std::endl;
                    }
                    break;
                default:
                    std::cout << "\t\t" << str << ":\t" << *info << std::endl;
                    break;
            }
        }
};

///
// Simple trait class used to wrap base types.
//
template <typename T>
class ArrayType
{
public:
        static bool isChar() { return false; }
};

///
// Specialized for the char (i.e. null terminated string case).
//
template<>
class ArrayType<char>
{
public:
        static bool isChar() { return true; }
};

///
// Specialized instance of class InfoDevice for array types.
//
template <typename T>
class InfoDevice<ArrayType<T> >
{
public:
        static void display(
                cl_device_id id,
                cl_device_info name,
                std::string str)
        {
                cl_int errNum;
                std::size_t paramValueSize;

                errNum = clGetDeviceInfo(
                        id,
                        name,
                        0,
                        NULL,
                        &paramValueSize);
                if (errNum != CL_SUCCESS)
                {
                        std::cerr
                                << "Failed to find OpenCL device info "
                                << str
                                << "."
                                << std::endl;
                        return;
                }

                T * info = (T *)alloca(sizeof(T) * paramValueSize);
                errNum = clGetDeviceInfo(
                        id,
                        name,
                        paramValueSize,
                        info,
                        NULL);
                if (errNum != CL_SUCCESS)
                {
                        std::cerr
                                << "Failed to find OpenCL device info "
                                << str
                                << "."
                                << std::endl;
                        return;
                }

                if (ArrayType<T>::isChar())
                {
                        std::cout << "\t" << str << ":\t" << info << std::endl;
                }
                else if (name == CL_DEVICE_MAX_WORK_ITEM_SIZES)
                {
                        cl_uint maxWorkItemDimensions;

                        errNum = clGetDeviceInfo(
                                id,
                                CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS,
                                sizeof(cl_uint),
                                &maxWorkItemDimensions,
                                NULL);
                        if (errNum != CL_SUCCESS)
                        {
                                std::cerr
                                        << "Failed to find OpenCL device info "
                                        << "CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS."
                                        << std::endl;
                                return;
                        }

                        std::cout << "\t" << str << ":\t" ;
                        for (cl_uint i = 0; i < maxWorkItemDimensions; i++)
                        {
                                std::cout << info[i] << " ";
                        }
                        std::cout << std::endl;
                }
        }
};

///
//  Enumerate platforms and display information about them
//  and their associated devices.
//
void displayInfo(void)
{
    cl_int errNum;
    cl_uint numPlatforms;
    cl_platform_id * platformIds;
    cl_context context = NULL;
```

```
        // First, query the total number of platforms
    errNum = clGetPlatformIDs(0, NULL, &numPlatforms);
    if (errNum != CL_SUCCESS || numPlatforms <= 0)
    {
                std::cerr << "Failed to find any OpenCL platform." << std::endl;
                return;
    }

        // Next, allocate memory for the installed plaforms, and qeury
        // to get the list.
        platformIds = (cl_platform_id *)alloca(sizeof(cl_platform_id) * numPlatforms);
        // First, query the total number of platforms
    errNum = clGetPlatformIDs(numPlatforms, platformIds, NULL);
    if (errNum != CL_SUCCESS)
    {
                std::cerr << "Failed to find any OpenCL platforms." << std::endl;
                return;
    }

        std::cout << "Number of platforms: \t" << numPlatforms << std::endl;
        // Iterate through the list of platforms displaying associated information
        for (cl_uint i = 0; i < numPlatforms; i++) {
                // First we display information associated with the platform
                DisplayPlatformInfo(
                        platformIds[i],
                        CL_PLATFORM_PROFILE,
                        "CL_PLATFORM_PROFILE");
                DisplayPlatformInfo(
                        platformIds[i],
                        CL_PLATFORM_VERSION,
                        "CL_PLATFORM_VERSION");
                DisplayPlatformInfo(
                        platformIds[i],
                        CL_PLATFORM_VENDOR,
                        "CL_PLATFORM_VENDOR");
                DisplayPlatformInfo(
                        platformIds[i],
                        CL_PLATFORM_EXTENSIONS,
                        "CL_PLATFORM_EXTENSIONS");

                // Now query the set of devices associated with the platform
                cl_uint numDevices;
                errNum = clGetDeviceIDs(
                        platformIds[i],
                        CL_DEVICE_TYPE_ALL,
                        0,
                        NULL,
                        &numDevices);
                if (errNum != CL_SUCCESS)
                {
                        std::cerr << "Failed to find OpenCL devices." << std::endl;
                        return;
                }

                cl_device_id * devices = (cl_device_id *)alloca(sizeof(cl_device_id) *
numDevices);
                errNum = clGetDeviceIDs(
                        platformIds[i],
                        CL_DEVICE_TYPE_ALL,
                        numDevices,
                        devices,
                        NULL);
                if (errNum != CL_SUCCESS)
                {
                        std::cerr << "Failed to find OpenCL devices." << std::endl;
                        return;
                }

                std::cout << "\tNumber of devices: \t" << numDevices << std::endl;
                // Iterate through each device, displaying associated information
                for (cl_uint j = 0; j < numDevices; j++)
                {
                        InfoDevice<cl_device_type>::display(
                               devices[j],
                               CL_DEVICE_TYPE,
                               "CL_DEVICE_TYPE");

                        InfoDevice<cl_uint>::display(
                               devices[j],
                               CL_DEVICE_VENDOR_ID,
                               "CL_DEVICE_VENDOR_ID");

                        InfoDevice<cl_uint>::display(
                               devices[j],
                               CL_DEVICE_MAX_COMPUTE_UNITS,
                               "CL_DEVICE_MAX_COMPUTE_UNITS");

                        InfoDevice<cl_uint>::display(
                               devices[j],
                               CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS,
                               "CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS");

                        InfoDevice<ArrayType<size_t> >::display(
                               devices[j],
                               CL_DEVICE_MAX_WORK_ITEM_SIZES,
                               "CL_DEVICE_MAX_WORK_ITEM_SIZES");

                        InfoDevice<std::size_t>::display(
                               devices[j],
                               CL_DEVICE_MAX_WORK_GROUP_SIZE,
                               "CL_DEVICE_MAX_WORK_GROUP_SIZE");

                        InfoDevice<cl_uint>::display(
                               devices[j],
                               CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR,
                               "CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR");

                        InfoDevice<cl_uint>::display(
                               devices[j],
                               CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT,
                               "CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT");

                        InfoDevice<cl_uint>::display(
                               devices[j],
                               CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT,
                               "CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT");

                        InfoDevice<cl_uint>::display(
                               devices[j],
                               CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG,
                               "CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG");

                        InfoDevice<cl_uint>::display(
                               devices[j],
                               CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT,
                               "CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT");

                        InfoDevice<cl_uint>::display(
                               devices[j],
                               CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE,
                               "CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE");

#ifdef CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF

                        InfoDevice<cl_uint>::display(
                               devices[j],
                               CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF,
                               "CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF");

                        InfoDevice<cl_uint>::display(
                               devices[j],
                               CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR,
                               "CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR");
```

```
                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT,
                              "CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT");

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_NATIVE_VECTOR_WIDTH_INT,
                              "CL_DEVICE_NATIVE_VECTOR_WIDTH_INT");

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG,
                              "CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG");

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT,
                              "CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT");

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE,
                              "CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE");

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF,
                              "CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF");
#endif

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_MAX_CLOCK_FREQUENCY,
                              "CL_DEVICE_MAX_CLOCK_FREQUENCY");

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_ADDRESS_BITS,
                              "CL_DEVICE_ADDRESS_BITS");

                      InfoDevice<cl_ulong>::display(
                              devices[j],
                              CL_DEVICE_MAX_MEM_ALLOC_SIZE,
                              "CL_DEVICE_MAX_MEM_ALLOC_SIZE");

                      InfoDevice<cl_bool>::display(
                              devices[j],
                              CL_DEVICE_IMAGE_SUPPORT,
                              "CL_DEVICE_IMAGE_SUPPORT");

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_MAX_READ_IMAGE_ARGS,
                              "CL_DEVICE_MAX_READ_IMAGE_ARGS");

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_MAX_WRITE_IMAGE_ARGS,
                              "CL_DEVICE_MAX_WRITE_IMAGE_ARGS");

                      InfoDevice<std::size_t>::display(
                              devices[j],
                              CL_DEVICE_IMAGE2D_MAX_WIDTH,
                              "CL_DEVICE_IMAGE2D_MAX_WIDTH");

                      InfoDevice<std::size_t>::display(
                              devices[j],
                              CL_DEVICE_IMAGE2D_MAX_WIDTH,
                              "CL_DEVICE_IMAGE2D_MAX_WIDTH");

                      InfoDevice<std::size_t>::display(
                              devices[j],
                              CL_DEVICE_IMAGE2D_MAX_HEIGHT,
                              "CL_DEVICE_IMAGE2D_MAX_HEIGHT");

                      InfoDevice<std::size_t>::display(
                              devices[j],
                              CL_DEVICE_IMAGE3D_MAX_WIDTH,
                              "CL_DEVICE_IMAGE3D_MAX_WIDTH");

                      InfoDevice<std::size_t>::display(
                              devices[j],
                              CL_DEVICE_IMAGE3D_MAX_HEIGHT,
                              "CL_DEVICE_IMAGE3D_MAX_HEIGHT");

                      InfoDevice<std::size_t>::display(
                              devices[j],
                              CL_DEVICE_IMAGE3D_MAX_DEPTH,
                              "CL_DEVICE_IMAGE3D_MAX_DEPTH");

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_MAX_SAMPLERS,
                              "CL_DEVICE_MAX_SAMPLERS");

                      InfoDevice<std::size_t>::display(
                              devices[j],
                              CL_DEVICE_MAX_PARAMETER_SIZE,
                              "CL_DEVICE_MAX_PARAMETER_SIZE");

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_MEM_BASE_ADDR_ALIGN,
                              "CL_DEVICE_MEM_BASE_ADDR_ALIGN");

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE,
                              "CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE");

                      InfoDevice<cl_device_fp_config>::display(
                              devices[j],
                              CL_DEVICE_SINGLE_FP_CONFIG,
                              "CL_DEVICE_SINGLE_FP_CONFIG");

                      InfoDevice<cl_device_mem_cache_type>::display(
                              devices[j],
                              CL_DEVICE_GLOBAL_MEM_CACHE_TYPE,
                              "CL_DEVICE_GLOBAL_MEM_CACHE_TYPE");

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE,
                              "CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE");

                      InfoDevice<cl_ulong>::display(
                              devices[j],
                              CL_DEVICE_GLOBAL_MEM_CACHE_SIZE,
                              "CL_DEVICE_GLOBAL_MEM_CACHE_SIZE");

                      InfoDevice<cl_ulong>::display(
                              devices[j],
                              CL_DEVICE_GLOBAL_MEM_SIZE,
                              "CL_DEVICE_GLOBAL_MEM_SIZE");

                      InfoDevice<cl_ulong>::display(
                              devices[j],
                              CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE,
                              "CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE");

                      InfoDevice<cl_uint>::display(
                              devices[j],
                              CL_DEVICE_MAX_CONSTANT_ARGS,
                               "CL_DEVICE_MAX_CONSTANT_ARGS");

                       InfoDevice<cl_device_local_mem_type>::display(
                               devices[j],
                               CL_DEVICE_LOCAL_MEM_TYPE,
                               "CL_DEVICE_LOCAL_MEM_TYPE");

                       InfoDevice<cl_ulong>::display(
                               devices[j],
                               CL_DEVICE_LOCAL_MEM_SIZE,
                               "CL_DEVICE_LOCAL_MEM_SIZE");

                       InfoDevice<cl_bool>::display(
                               devices[j],
                               CL_DEVICE_ERROR_CORRECTION_SUPPORT,
                               "CL_DEVICE_ERROR_CORRECTION_SUPPORT");

#ifdef CL_DEVICE_HOST_UNIFIED_MEMORY
                       InfoDevice<cl_bool>::display(
                               devices[j],
                               CL_DEVICE_HOST_UNIFIED_MEMORY,
                               "CL_DEVICE_HOST_UNIFIED_MEMORY");
#endif

                       InfoDevice<std::size_t>::display(
                               devices[j],
                               CL_DEVICE_PROFILING_TIMER_RESOLUTION,
                               "CL_DEVICE_PROFILING_TIMER_RESOLUTION");

                       InfoDevice<cl_bool>::display(
                               devices[j],
                               CL_DEVICE_ENDIAN_LITTLE,
                               "CL_DEVICE_ENDIAN_LITTLE");

                       InfoDevice<cl_bool>::display(
                               devices[j],
                               CL_DEVICE_AVAILABLE,
                               "CL_DEVICE_AVAILABLE");

                       InfoDevice<cl_bool>::display(
                               devices[j],
                               CL_DEVICE_COMPILER_AVAILABLE,
                               "CL_DEVICE_COMPILER_AVAILABLE");

                       InfoDevice<cl_device_exec_capabilities>::display(
                               devices[j],
                               CL_DEVICE_EXECUTION_CAPABILITIES,
                               "CL_DEVICE_EXECUTION_CAPABILITIES");

                       InfoDevice<cl_command_queue_properties>::display(
                               devices[j],
                               CL_DEVICE_QUEUE_PROPERTIES,
                               "CL_DEVICE_QUEUE_PROPERTIES");

                       InfoDevice<cl_platform_id>::display(
                               devices[j],
                               CL_DEVICE_PLATFORM,
                               "CL_DEVICE_PLATFORM");

                       InfoDevice<ArrayType<char> >::display(
                               devices[j],
                               CL_DEVICE_NAME,
                               "CL_DEVICE_NAME");

                       InfoDevice<ArrayType<char> >::display(
                               devices[j],
                               CL_DEVICE_VENDOR,
                               "CL_DEVICE_VENDOR");

                       InfoDevice<ArrayType<char> >::display(
                               devices[j],
                               CL_DRIVER_VERSION,
                               "CL_DRIVER_VERSION");

                       InfoDevice<ArrayType<char> >::display(
                               devices[j],
                               CL_DEVICE_PROFILE,
                               "CL_DEVICE_PROFILE");

                       InfoDevice<ArrayType<char> >::display(
                               devices[j],
                               CL_DEVICE_VERSION,
                               "CL_DEVICE_VERSION");

#ifdef CL_DEVICE_OPENCL_C_VERSION
                       InfoDevice<ArrayType<char> >::display(
                               devices[j],
                               CL_DEVICE_OPENCL_C_VERSION,
                               "CL_DEVICE_OPENCL_C_VERSION");
#endif

                       InfoDevice<ArrayType<char> >::display(
                               devices[j],
                               CL_DEVICE_EXTENSIONS,
                               "CL_DEVICE_EXTENSIONS");


                       std::cout << std::endl << std::endl;
               }
       }
}

///
//     main() for OpenCLInfo example
//
int main(int argc, char** argv)
{
    cl_context context = 0;

       displayInfo();

    return 0;
 }
```