Treball de fi de Carrera

ENGINYERIA TÈCNICA EN INFORMÀTICA DE SISTEMES

Facultat de Matemàtiques
Universitat de Barcelona

SimpleOpenCL: desenvolupament i documentació d'una llibreria que facilita la programació paral·lela en OpenCL

Rafael García Ortega

Director: Oscar Amoros Huguet
Realitzat a: Departament de Matemàtica Aplicada i Anàlisi. UB
Barcelona, 19 de setembre de 2013

Table of contents

Introduction

Computation always was related with big and expensive computers or clusters. But it changed the day that some people decide to use GPUs for general purpose computation instead of graphics rendering. GPUs are good for parallel computing because its high performance multi-core processors. It is like returns to the past, to coprocessors age but with GPUs.

Main vendors implemented low level hardware programming interfaces (I.e. AMD close-to-metal and Nvidia CUDA) to take advantage of modern GPUs. But the (big)drawback about these solutions is that applications created with theses SDKs are only compatibles with their vendor hardware. This is where OpenCL comes in.

OpenCL borns as an open standard. It let heterogeneous computing in a heterogeneous world. But programming with OpenCL is not easy at all. It is a low level API and after coding some applications you will see some patterns repeated.

SimpleOpenCL is a library that simplifies programming with OpenCL. It is an open source project with GPL v3 license.

The main idea behind this thesis is to analyze SimpleOpenCL for improving it and offer proposals for its future, but to achieve it I need to talk at first about OpenCL.

This thesis have an introduction about the current topic and it is organized with the following outline:

1. **OpenCL**
   Before going into SimpleOpenCL we need to understand the technologies behind it. This chapter begins with the origins of OpenCL (Open Computing Language), base of SimpleOpenCL, and then continues explaining the basics concepts.

2. **SimpleOpenCL**
   After look at the basic concepts we continue with SimpleOpenCL. It opens with an introductory text about it. Next, OpenCL and SimpleOpenCL flows are explained step by step to compare them and understand the advantages of using SimpleOpenCL. For finishing I show you the current infrastructure used by this open source project.

3. **Project contribution**
   Open source projects without contributions does not exist, so here is my contribution. This chapter is divided in three sections: Code review, documentation and proposals. Code review is the results of my analysis. Here, I enumerate and explain some issues that I have found.

Documentation is a testimonial section about my research in this field. Proposals is a section where I give some ideas for the project about infrastructure, code and documentation level.

4. **Conclusions**

Thoughts acquired by writing this thesis about this open source project.

5. **Future works**

Small note with things to do after the end of this thesis.

OpenCL

OpenCL (Open Computing Language) is an open standard for heterogeneous hardware general-purpose parallel programming.

OpenCL is also a framework(an API and a programming language) for writing programs and executing across central processing units (CPUs), graphic processing units (GPUs), coprocessors, and other parallel processors.

The OpenCL language is based on C99, removing some features and extended with built in data types, functions and vector operations. It is used for writing kernels (functions that execute on OpenCL devices).

The OpenCL API is used for coordinating parallel computation across heterogeneous processors. With this API define the platforms and then control them. It is a low level API.

Summarizing, OpenCL lets programmers write a single cross-platform program that uses all resources in the heterogeneous platform.

Image 1: OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

The origins of OpenCL

To discover the origins of OpenCL we should speak about high performance computing (HPC). When I thought about HPC, I always visualize in my mind a cluster or supercomputers. Something only affordable by universities or government organizations for scientific, military business, simulation... It makes sense because, before OpenCL, HPC was related with parallelism and homogeneity. To achieve computing power you had to buy a lot of computers an connect them, this meant only big budgets could access to it. Also the small amount of projects does not helps to invest resources for HPC.

But HPC situation changed when general purpose CPUs begins to include vector processing units. General purpose CPUs are always being improved because it is sold in mass. It is not an specific architecture for an HPC project is for real world usage with a return of investment (ROI). Another architecture and mainstream for mass market get the same direction, I'm speaking about Graphics Processing Units (GPUs).

GPUs, with the introduction of programmable shaders, were able to leave aside fixed function circuitry for creating visual effects and program and process it directly. Scientists saw its potential for matrix computations and started what was General-Purpose Computing on Graphics Processing Units (GPGPU).

Some companies wrote its owns APIs for programming theirs GPUs (i.e. Nvidia with CUDA, Ati with Close to Metal). But there was no standard.

Here, we were in an interesting situation. In one side we had CPUs with multiple cores driving performance increases, in other side we had GPUs increasing general-purpose parallel programming. CPUs use multi-processor programming and GPUs graphics APIs and shading languages.

From the main companies point of view we had:

- AMD and Ati were merged, need to do the same with their products
- Nvidia as GPU vendor wanted to steal market share from CPU
- Intel as CPU vendor wanted to steal market share from GPU
- Apple was tired of them and pushed vendors to standardize.

Apple defined and submitted the initial OpenCL proposal to the Khrono Group. It was the intersection point between CPUs and GPUs.

OpenCL became an important standard on release by virtue of the market coverage of the companies behind it (Processor vendors, system OEMs, middleware vendors, application developers...).

As a curiosity, OpenCL 1.0 version was released on 8 December 2008. At the this moment OpenCL 2.0 is the latest version of OpenCL specification but may change because it is not yet finished.

OpenCL basic concepts

OpenCL is an open standard for parallel programming a heterogeneous collection of CPUs, GPUs and other processors. OpenCL target is expert programmers therefore it provides a low-level hardware abstraction(i.e. Hardware details are exposed).

To understand OpenCL's global architecture we will review the following hierarchy of models and then discover the main ideas behind OpenCL:

- Platform model
- Execution model
- Memory model
- Programming model

Platform model

“The platform model is an abstraction describing how OpenCL views the hardware.” from OpenCL 2.0 provisional specification.

To understand the platform model of OpenCL lets have a look to the next image:

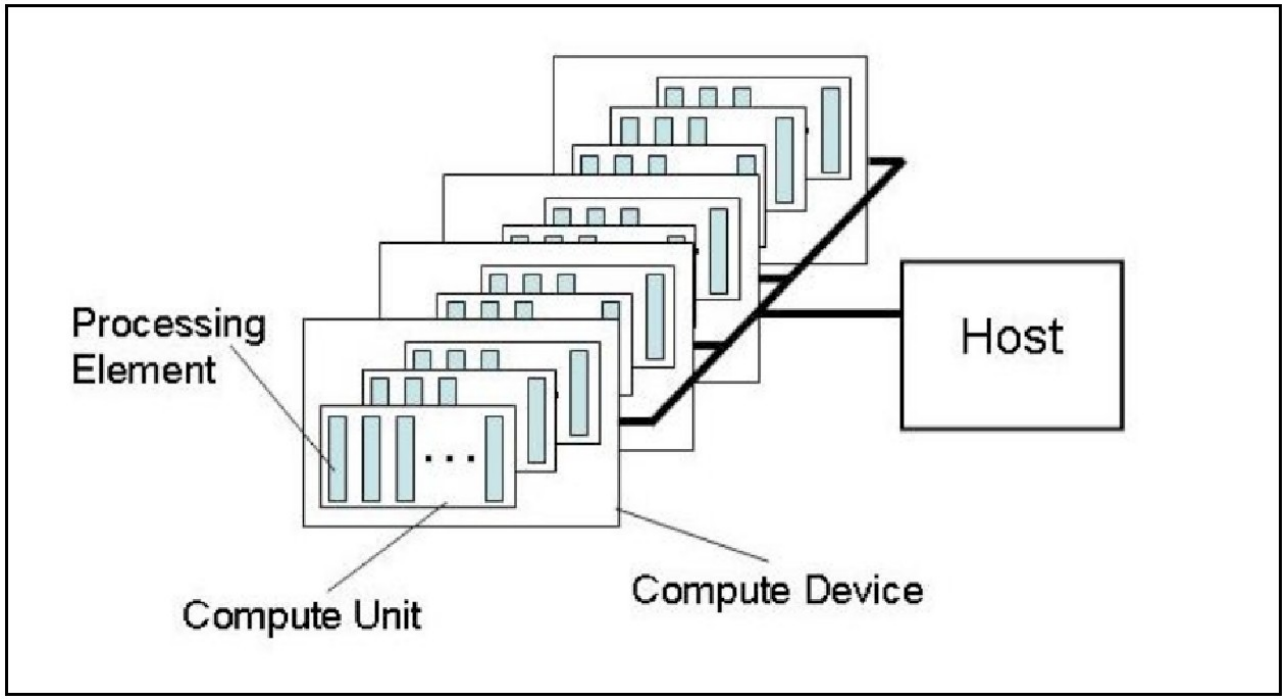


Image 2: platform model courtesy of Khronos Group

We have a host connected a compute device. Compute device contain compute units. These compute units is divided within processing elements.

Doing an analogy with our real world:

- Host = CPU where OpenCL main program run on.
- Compute device = Graphic card with OpenCL support.
- Compute unit = Graphic card multiprocessor, some kind of core.
- Processing element = It is some kind SIMD or SPMD unit with ALU(s) and SFU(s).

Based on our previous analogy and platform model platform we extract:

- A host is connected to one or more OpenCL devices.
- OpenCL device is divided in several compute units responsible of one or more execution flows.
- Compute unit contains one or more processing elements. Computations are made within processing elements.

Device code is written in OpenCL C (C99 variant). A platform have a compiler to convert an OpenCL C source program into a executable program object. The compiler could be online or offline. With standard APIs, at runtime, you could use online compiler to generate executable program objects. Outside of host program you can generate executable program objects with offline compilers but it depends on platform specific methods. Host program could load and execute previously compiled program objects.

Devices also could have special purpose functionality as a built-in function. The platform provides methods for enumerating and invoking these built-in functions.

Execution model

To understand OpenCL execution model first we should see the anatomy of an OpenCL application.
An OpenCL application consists on host code and device code. Host code is written with C/C++ and executed on the host. Device code is written in OpenCL C and executed on the device.
This partition offers us two different units of execution: kernels that execute on OpenCL devices and a host program that executes on host. We can say that kernels are where work is associated with a computation is done. This work occurs through work-items that are executed in groups (work-groups).

The host manage the context where kernel is executed. Context could be considered the spine of any OpenCL application, it drives the communication with/between devices. Contexts provides a container for devices and their memories and command queues. Context includes the following resources:

- Devices: Devices exposed by OpenCL
- Kernel objects: OpenCL functions and values to run on devices
- Program objects: Source and executable of programs that implement kernels (basically C

functions).

- Memory objects: Variables visibles shared between host and devices. Kernel instances use memory objects when are executed.

The host, trough OpenCL API, interact with devices using a command-queue. Each command-queue is associated a single device. There are three types of commands:

- Kernel-enqueue commands: Enqueue a command for execution on a device.
- Memory commands: Transfer data between host, devices and memory objects, map or unmap memory objects the host address space.
- Synchronization commands: Explicit synchronization points that define order constraints between commands.

Besides host command-queue, running kernel can enqueue commands to a device-side command queue. Regardless type command-queue, each command passes through six stages: queued, submitted, ready, running, ended and complete (for more detail read OpenCL 2.0 specification, pages 25-28).

When command is submitted for execution an index space is defined. A single kernel instance at a point in the index space is called work-item. The device manage work-items associated to a given kernel-instance in groups, these groups are called work-groups.

Work-items have a global ID based on their coordinates within index space. This index space is called NDRange. NDRange is a n-dimensional(1,2 or 3 dimensions) and describes the space of work-items. Work-item and work-group have the same dimensionality (if work-item space is n-dimensional, work-group space will be also n-dimensional).

Work-items also have an ID local to the work-group. It made a work-item identifiable via global index or work-group index plus local index within a work-group.

In relation with synchronization OpenCL 2.0 specification says: “There is no safe and portable way to synchronize across the independent execution of workgroups since once in the work-pool, they can execute in any order.”. However synchronization between work-items is possible within work-groups. High level synchronization methods are defined and included in OpenCL(ie. Barriers and memory fences).

Some implementations of OpenCL supports subgroups (work-group decomposition). Subgroups existence are highly implementation dependent. It means you should have extremly care when writing code that uses subgroups if you want a portable OpenCL application.

To finish with execution model lets see the three types of kernels supported:

- OpenCL kernels: Usually written with OpenCL C and compiled with OpenCL C compiler. OpenCL kernels are managed by OpenCL API as kernel-objects. All OpenCL

implementations must support OpenCL kernels and OpenCL C.

- Native kernels: are C functions which are called by OpenCL. Native kernels support is optional for OpenCL implementations but OpenCL API offer functions to query/determine if it is supported.
- Built-in kernels: Are used to expose fixed functions tied to a device.

Memory model

The following image shows you the relation between memory regions and platform model:

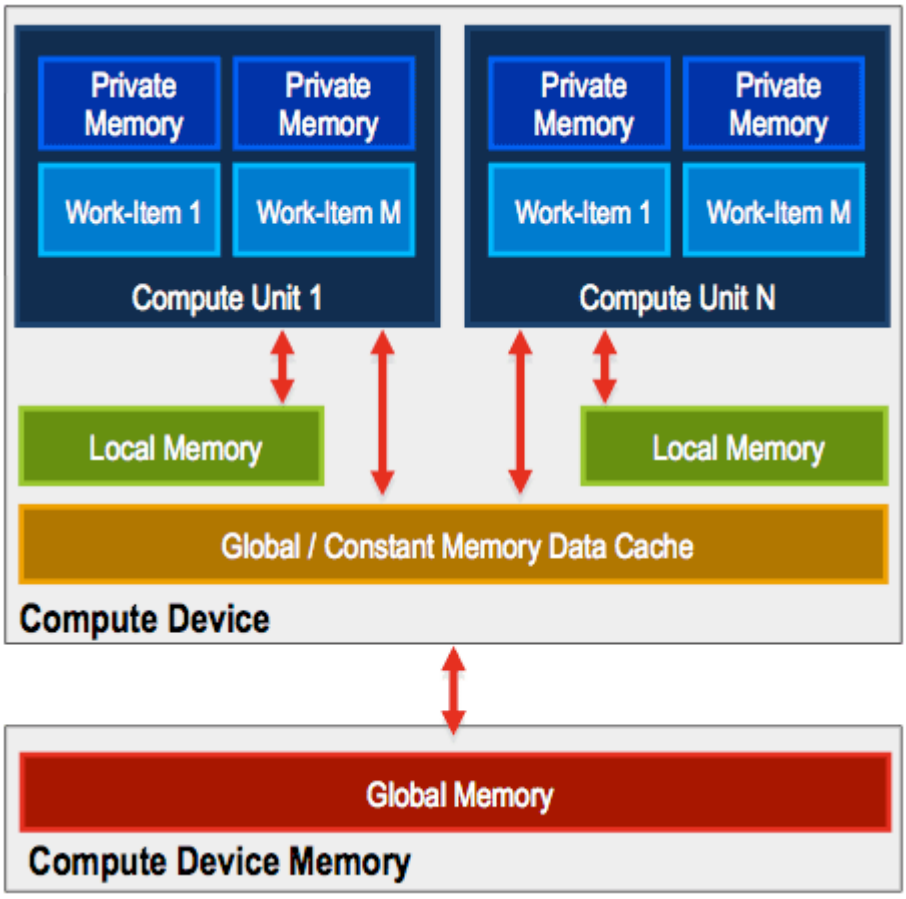


Image 3: memory model courtesy of Khronos Group

First, we have memory divided into two parts independents:

- Host memory: Memory directly available to the host. Host memory is managed outside of OpenCL. OpenCL API manage memory between host and devices.
- Device memory: Memory directly available to kernels executing on OpenCL devices.

Device memory consists of four memory regions:

- Global memory: This memory region permits read/write access to all work-items in all work-groups running on any device within a context. It is generally the largest capacity

memory but it should be considered the slowest. Read and writes could be cached, depending on the device.

- Constant memory: It is a read only memory section. Also visible to all work-groups.
- Local memory: A memory shared local to a work-group. It is faster than global memory.
- Private memory: A memory used within work-item, similar to registers.

As you can see in previous image, local and private region memories are always inside a compute device (are associated with particular devices). Global and constant memories, according to OpenCL 2.0 provisional specification (not the image), are shared between devices within a given context.

Memory objects are the contents of global memory. There are three distinct classes:

- Buffer: It is used to share data between devices.
- Image: An image memory is an opaque structure/object (2D or 3D) and can only be accesed by OpenCL API functions. In previous version of OpenCL kernels have been disallowed from both reading and writing a single image but OpenCL 2.0 provides synchronization and fence operations that let synchronize their code to safely allow a kernel to read and write a single image.
- Pipe: Pipe memory object uses producer-consumer pattern. A kernel instance connects to write endpoint(insert data) while another kernel instance connects to reading endpoint(remove data).

Allocation and access to memory objects in the different address spaces varies between host and work-items. The following table shows different memory region and how objects are allocated and accessed by the host and kernel instances:

	Global	Constant	Local	Private
Host	Dynamic allocation	Dynamic allocation	Dynamic allocation	No allocation
	Read/write access to buffers and images but not pipes	Read/write access	No access	No access
Kernel	Static allocation for program scope variables	Static allocation	Static allocation Dynamic allocation for child kernel	Read/write access
	Read/write access	Read-only access	Read/write access No access to child's local memory	Read/write access

Table 1: According to OpenCL 2.0 specification courtesy of Khronos Group

At OpenCL 2.0 specification you can find more detailed information about memory model(shared virtual memory, consistency model, overview of atomic and fence operations and ordering rules) but it goes beyond the scope of this document.

SimpleOpenCL

To explain what is SimpleOpenCL in one line I'm going to quote his author: “SimpleOpenCL is a library created to reduce the amount of host code needed to write an OpenCL program.”.

SimpleOpenCL was born after some works done about scientific research by the author, Oscar Amoros, with OpenCL. He realized that lot of codes was duplicated between developments and could therefore be more DRY (“Don't repeat yourself “ principle). For this reason he began to develop SimpleOpenCL.

SimpleOpenCL is a library wrapper for the OpenCL architecture written in ANSI C. Having in mind always to reduce this duplicated code, mostly host code needed to run OpenCL C kernels on the GPU and CPU.

In the next section I would like to show you the SimpleOpenCL's good parts by comparing it with OpenCL used directly.

Application flows (OpenCL and SimpleOpenCL)

In previous chapter about OpenCL I only explained it from a theoretical point of view to understand main concepts but I didn't show any line of code. It was premeditated.

Let's to discover the different application flows needed to work with OpenCL and SimpleOpenCL. Every step will have code from the examples.

OpenCL application flow

1. Get a list of available platforms
2. Select device
3. Create Context
4. Create command queue
5. Create memory objects
6. Read kernel file
7. Create program object
8. Compile kernel
9. Create kernel object
10. Set kernel arguments
11. Execute kernel (Enqueue task)
12. Read memory object
13. Free objects

Based on this points we will analize the example OpenCL program that you find at SimpleOpenCL home page.

Get a list of available platforms

A platform is a host with, at least, one OpenCL device. We need to know the available platforms:

```
...
12          cl_platform_id platform;
13          cl_device_id device;
14          cl_uint platforms, devices;
15
16          // Fetch the Platform and Device IDs; we only want one.
17          error=clGetPlatformIDs(1, &platform, &platforms);
18          if (error != CL_SUCCESS) {
```

```
19                  printf("\n Error number %d", error);
20          }
…
```

clGetPlatformIDs() function in line 17 have the following signature:

```
cl_int clGetPlatformIDs( cl_uint num_entries,
                         cl_platform_id *platforms,
                         cl_uint *num_platforms)
```

Parameters:

- num_entries: The number of cl_platform_id entries that can be added to platforms, most of the time 1.
- platforms: Return a list of OpenCL platforms found.
- num_platforms: Returns the number of OpenCL platforms that can be used.

Select device

Next step is to obtain devices list within the platform. This is done in the next code segment:

```
...
13  cl_device_id device;
14  cl_uint platforms, devices;
...
21  error=clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device, &devices);
...
```

clGetDeviceIDs() in line 21 have the following signature:

```
cl_int clGetDeviceIDs(cl_platform_id platform,
                      cl_device_type device_type,
                      cl_uint num_entries,
                      cl_device_id *devices,
                      cl_uint *num_devices)
```

Parameters:

- platform: Refers to platform ID obtained in the previous step.
- device_type: To query which kind of OpenCL devices are available (Check OpenCL manual to know valid values).
- num_entries: The number of cl_device entries that can be added to devices.
- devices: Return a list of OpenCL devices.

- num_devices: Return the number of avalaible OpenCL devices (based on device_type parameter).

Create Context

The context is used by OpenCL as environment to manage command-queues, memory, executing kernels, …

To create a context:

```
...
29  cl_context context=clCreateContext(properties, 1, &device, NULL, NULL,
  &error);
30  if (error != CL_SUCCESS) {
31    printf("\n Error number %d", error);
32  }
...
```

clCreateContext() have the following signature:

```
cl_context clCreateContext(cl_context_properties *properties,
                           cl_uint num_devices,
                           const cl_device_id *devices,
                           void *pfn_notify (
                             const char *errinfo,
                             const void *private_info,
                             size_t cb,
                             void *user_data
                            ),
                           void *user_data,
                           cl_int *errcode_ret)
```

Parameters:

- properties: Specifies a list of conext property.
- num_devices: Numbers of devices to use.
- devices: A list of devices.
- pfn_notify: To register a callback function. This function will be used by the OpenCL implementation to report information on errores in this context.
- user_data: From OpenCL specification “Passed as the user_data argument when pfn_notify is called. user_data can be NULL.”.

- errcode_ret: Returns error code.

This function returns a context and errcode_ret is set to CL_SUCCESS if context is created successfully else returns a NULL and set the error code in errcode_ret.

Create command queue

Command queue is used to communicate with the device. Any command from the host to the device is performed through this command queue.
To create it:

```
...
33  cl_command_queue cq = clCreateCommandQueue(context, device, 0, &error);
34  if (error != CL_SUCCESS) {
35    printf("\n Error number %d", error);
36  }
...
```

No signature or parameters explained here because are obvious.

This function returns a command queue and errcode_ret is set to CL_SUCCESS if it is created successfully. Otherwise, it returns a NULL and set the error code in errcode_ret.

Create memory objects

Memory objects are created to allow the host to access the device memory. Kernels store all processed data in memory device but does not have the capability to access memory outside of the device.

```
...
// Allocate memory for the kernel to work with
61  cl_mem mem1, mem2;
62  mem1=clCreateBuffer(context, CL_MEM_READ_ONLY, worksize, NULL, &error);
63  if (error != CL_SUCCESS) {
64    printf("\n Error number %d", error);
65  }
66  mem2=clCreateBuffer(context, CL_MEM_WRITE_ONLY, worksize, NULL, &error);
67  if (error != CL_SUCCESS) {
68    printf("\n Error number %d", error);
69  }
```

```
...
```

clCreateBuffer() function have the following signature:

```
cl_mem clCreateBuffer (cl_context context,
                       cl_mem_flags flags,
                       size_t size,
                       void *host_ptr,
                       cl_int *errcode_ret)
```

Parameters:

- context: A valid OpenCL context.
- flags: It is used to define memory allocation and usage.
- size: The size in bytes of the memory object to be allocated.
- host_ptr: “A pointer to the buffer data that may already be allocated by the application. The size of the buffer that host_ptr points to must be ≥ size bytes.” (from OpenCL 1.2 specification).
- errcode_ret: Returns the error code.

This function returns a buffer object and errcode_ret is set to CL_SUCCESS if it is created successfully. Otherwise, it returns a NULL and set the error code in errcode_ret.

Read kernel file

Kernel can only be executed via host program, then the host program must read the kernel program. It can do it from an external file(or hardcoded in the host program) which must be compiled using an OpenCL compiler or binary.

Create program object

We need to create a program object and load the source code of the kernel(OpenCL C program) into it.

```
...
45  cl_program prog=clCreateProgramWithSource(context,
46                                             1, srcptr, &srcsize, &error);
...
```

clCreateProgramWithSource() function signature:

```
cl_program clCreateProgramWithSource (cl_context context,
```

```
                                        cl_uint count,
                                        const char **strings,
                                        const size_t *lengths,
                                        cl_int *errcode_ret)
```

Parameters:

- context: A valid OpenCL context.
- Count: Number of strings in strings parameters (i.e. number of files)
- strings: Our source code
- lengths: Size of each string (source code).
- errcode_ret: Returns the error code

This function returns a program object and errcode_ret is set to CL_SUCCESS if it is created successfully. Otherwise, it returns a NULL and set the error code in errcode_ret.

Compile kernel

Next step is to compile the program object using an OpenCL C compiler. This step is unnecessary if the program is created from a binary (clCreateProgramWithBinary() function).

```
51  error=clBuildProgram(prog, 0, NULL, "", NULL, NULL);
```

clBuildProgram() function signature:

```
cl_int clBuildProgram (cl_program program,
                       cl_uint num_devices,
                       const cl_device_id *device_list,
                       const char *options,
                       void (CL_CALLBACK *pfn_notify)(cl_program program,
  void *user_data),
                       void *user_data)
```

Parameters:

- program: Program object to be compiled.
- num_devices: Number of target devices.
- device_list: Target devices for which binary is created.
- options: Build options.
- ptn_notify: A callback function to register when the program executable has been build (successfully or not).
- user_data: Same as clCreateContext

This function returns CL_SUCCESS if is executed successfully. Otherwise, it returns an error code.

Create kernel object

After program object compilation, we create a kernel object. Each kernel object have a kernel function, hence it is necessary to specify a name for each function.

```
...
71  cl_kernel k_example=clCreateKernel(prog, "function_name", &error);
72  if (error != CL_SUCCESS) {
73    printf("\n Error number %d", error);
74  }
...
```

Signature or parameters not need to be explained here.

This function returns a kernel object and errcode_ret is set to CL_SUCCESS if it is created successfully. Otherwise, it returns a NULL and set the error code in errcode_ret.

Set kernel arguments

Kernel is a function implemented using OpenCL C, this function will need parameters. With the following functions you set kernel arguments.

```
...
75  error = clSetKernelArg(k_example, 0, sizeof(mem1), &mem1);
76  if (error != CL_SUCCESS) {
77    printf("\n Error number %d", error);
78  }
...
```

clSetKernelArg() function signature:

```
cl_int clSetKernelArg (cl_kernel kernel,
                       cl_uint arg_index,
                       size_t arg_size,
                       const void *arg_value)
```

Parameters:

- kernel: A valid kernel object.
- arg_index: Argument index (0 leftmost argument, n-1 last argument).
- arg_size: Size of the argument value.
- arg_value: Pointer to the argument passed.

This function returns CL_SUCCESS if is executed successfully. Otherwise, it returns an error code.

Execute kernel (Enqueue task)

To execute the kernel we put it into the command queue (enqueue the task) to be executed on the device. Here is where kernel function is called.

```
...
94  error=clEnqueueNDRangeKernel(cq, k_example, 1, NULL, &worksize,
  &worksize, 0, NULL, NULL);
95  if (error != CL_SUCCESS) {
96    printf("\n Error number %d", error);
97  }
...
```

clEnqueueNDRangeKernel() function signature:

```
cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
                               cl_kernel kernel,
                               cl_uint work_dim,
                               const size_t *global_work_offset,
                               const size_t *global_work_size,
                               const size_t *local_work_size,
                               cl_uint num_events_in_wait_list,
                               const cl_event *event_wait_list,
                               cl_event *event)
```

Parameters:

- command_queue: A valid command queue.
- Kernel: A valid kernel object.
- work_dim: Number of dimensions used to specify the global work-items.
- global_work_offset: The offset used to calculated the global ID of a work-item. It could be NULL then global IDs start at offset (0, 0, 0,...0).
- global_work_size: Number of global work-items that will execute the kernel function. Total number of global work-items is computed as global_work_size[0] *...*

global_work_size[work_dim - 1].

- local_work_size: Size of the work-group. Number of work-items that make up a work-groups.
- num_events_in_wait_list: Number of events to complete before this command can be executed.
- event_wait_list: Events to complete before this command can be executed.
- event: Return an event object that identifies this particular kernel execution instance.

This function returns CL_SUCCESS if is executed successfully. Otherwise, it returns an error code.

Read memory object

After processing data on the kernel, the result should be transferred to the host-side.

```
...
98  // Read the result back into buf2
99  error=clEnqueueReadBuffer(cq, mem2, CL_FALSE, 0, worksize, buf2, 0, NULL,
  NULL);
100  if (error != CL_SUCCESS) {
101    printf("\n Error number %d", error);
102  }
...
```

clEnqueueReadBuffer() function signature:

```
cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
                            cl_mem buffer,
                            cl_bool blocking_read,
                            size_t offset,
                            size_t size,
                            void *ptr,
                            cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list,
                            cl_event *event)
```

Parameters:

- command_queue: A valid command queue.
- buffer: A valid buffer object.
- blocking_read: Indicates if the read operations are blocking(CL_TRUE) or not blocking(CL_FALSE).

- offset: The offset in bytes in the buffer object to read from.
- size: Size in bytes of data being read.
- ptr: Pointer to buffer in host memory.
- num_events_in_wait_list: Number of events to complete before this command can be executed.
- event_wait_list: Events to complete before this command can be executed.
- event: Return an event object that identifies this particular read command.

This function returns CL_SUCCESS if is executed successfully. Otherwise, it returns an error code.

Free objects

In our example objects are not freed but it is needed to do it real-life applications.

Example:

```
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(memobj);
ret = clReleaseCommandQueue(command_queue);
ret = clReleaseContext(context);
```

I think you get the idea about how tiresome could be program a simple program with OpenCL. We are ready to know the application flow with SimpleOpenCL.

SimpleOpenCL application flow

1. Get the hardware
2. Get the software
3. Manage arguments and launch kernel
4. Free objects

Only 4 points, compared with the 13 points used before for the OpenCL application flow, SimpleOpenCL looks like very simple (and really it is).

Get the hardware

In this step we select the OpenCL devices to run kernel code. We have several functions to select the devices but we choose the funtion that selects GPU.

```
...
7  sclHard hardware;
...
16 hardware = sclGetGPUHardware( 0, &found );
...
```

sclGetGPUHardware() function have the following signature:

```
sclHard sclGetGPUHardware( int nDevice, int* found );
```

Parameters:

- nDevice: Allow to specify which device if there are more than one.
- found: Number of devices found.

It returns a sclHard struct. This struct contain platform, context, device id, queue, compute units, max pointer size, device type (GPU, CPU or other) and device number.

Check SimpleOpenCL manual to see other functions for selecting the devices.

Get the software

In this point we have the hardware and we need to load a .cl file with one or more kernels.
We load the program source, create a program object, build it and create the kernel with just one function:

```
...
8      sclSoft software;
18  software = sclGetCLSoftware( "example.cl", "example", hardware );
...
```

Function sclGetCLSoftware() have the following signature:

```
sclSoft sclGetCLSoftware( char* path, char* name, sclHard hardware );
```

Parameters:

- path: A valid path of source file.
- name: A name for the kernel function.
- hardware: The hardware that will execute it.

This function returns a sclSoft struct that contain the program, kernel and kernel function name. It represents the compiled OpenCL kernel code.

Manage arguments and launch kernel

In this step we are ready to execute the code in the device because we have the hardware and software ready.

To execute it we should do:

```
sclManageArgsLaunchKernel( hardware, software, global_size, local_size,
                           " %r %w ",
                           worksize, buf, worksize, buf2 );
```

SclManageArgsLaunchKernel() have the following signature:

```
cl_event sclManageArgsLaunchKernel( sclHard hardware,
                                    sclSoft software,
                                    size_t *global_work_size,
                                    size_t *local_work_size,
                                    const char* sizesValues,
                                    ... );
```

Perhaps sclManageArgsLaunchKernel() function is the less intuitive at first sight. But you get used quickly when you understand that it uses printf-like codification for sizeValues and "..." parameters.

Check SimpleOpenCL manual to see how kernel arguments are defined using "printf-like codification".

Free objects

SimpleOpenCL have some functions that freed used variables automatically (i.e. sclManageArgsLaunchKernel and sclGetGPUHardware). There are some functions dedicated to release software, hardware and mem objects too.

Example:

```
sclReleaseMemObject( outBuffs[i] );
sclReleaseMemObject( inBuffs[i] );
```

Face to face

To appreciate SimpleOpenCL abstractions better I'll put each flow side by side:

OpenCL	SimpleOpenCL
1. Get a list of available platforms	1. Get the hardware
2. Select device	2. Get the software
3. Create Context	4. Manage arguments and launch kernel
4. Create command queue	5. Free objects
5. Create memory objects	
6. Read kernel file	
7. Create program object	
8. Compile kernel	
9. Create kernel object	
10. Set kernel arguments	
11. Execute kernel (Enqueue task)	
12. Read memory object	
13. Free objects	

Table 2: Flow comparative (OpenCL/SimpleOpenCL)

Of course, it is also translated to source code. Source codes from SimpleOpenCL website:

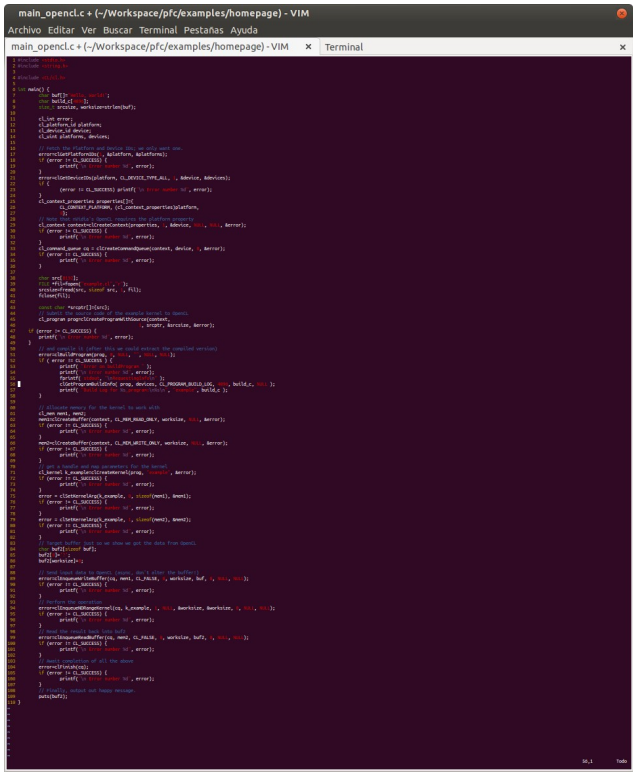
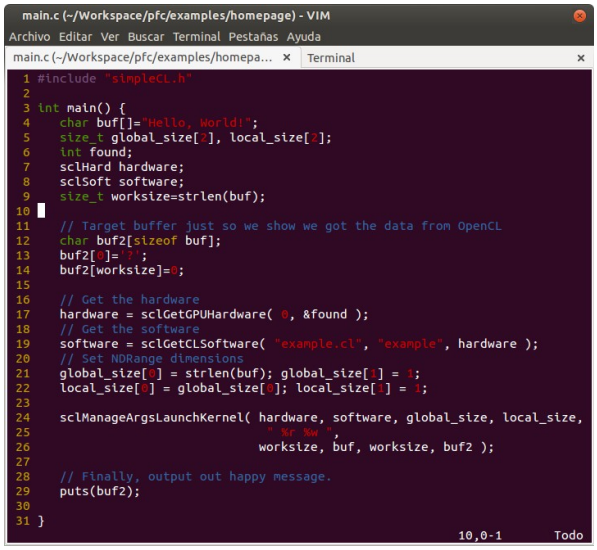
OpenCL	SimpleOpenCL
	

Table 3: Lines of code comparative (OpenCL/SimpleOpenCL)

My first program

I felt confidence to write my first program. It should be something simple but complete. I remembered a sample program that showed me all the information about host and OpenCL devices, and it also did some simple calculation (the index multiplied by a factor one hundred times).

I tried to replicate the functionality and the first part was easy. With SimpleOpenCL get the hardware and show the information is a trivial work. The second part, read the program(.cl file), build and compile it was easy too. But execute the commands to calculate the series does not worked.

The series is doing well the operations but first value is wrong, I guess it is something with a pointer or uninitialized values.

The problem was I did not read correctly the original program and I forget to initialize the values (index to by multiplied). After initialize it, result series where fine:

Output example:

```
$ ./main

Group 1 with 1 devices
Group 2 with 1 devices
 Device 0
 Platform name: AMD Accelerated Parallel Processing
 Vendor: Advanced Micro Devices, Inc.
 Device name: Redwood
 Device 1
 Platform name: AMD Accelerated Parallel Processing
 Vendor: Advanced Micro Devices, Inc.
 Device name: Intel(R) Core(TM) i7 CPU       Q 720  @ 1.60GHz

=== 0 OpenCL device(s) found on platform:
  -- 0 --
  DEVICE_NAME = Redwood
  DEVICE_VENDOR = Advanced Micro Devices, Inc.
  DEVICE_VERSION = OpenCL 1.2 AMD-APP (1214.3)
  DRIVER_VERSION = 1214.3
  DEVICE_MAX_COMPUTE_UNITS = 5
  DEVICE_MAX_CLOCK_FREQUENCY = 450
```

```
  DEVICE_GLOBAL_MEM_SIZE = 536870912
  -- 1 --
  DEVICE_NAME = Intel(R) Core(TM) i7 CPU       Q 720  @ 1.60GHz
  DEVICE_VENDOR = GenuineIntel
  DEVICE_VERSION = OpenCL 1.2 AMD-APP (1214.3)
  DRIVER_VERSION = 1214.3 (sse2)
  DEVICE_MAX_COMPUTE_UNITS = 8
  DEVICE_MAX_CLOCK_FREQUENCY = 933
  DEVICE_GLOBAL_MEM_SIZE = 8304603136
Result: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48
  50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98
  100 102 104 106 108 110 112 114 116 118 120 122 124 126 128 130 132 134 136
  138 140 142 144 146 148 150 152 154 156 158 160 162 164 166 168 170 172 174
  176 178 180 182 184 186 188 190 192 194 196 198
```

Note: You will find the source code in file src.zip

Working with SimpleOpenCL/OpenCL: Where is my memory?

After install AMD APP SDK (the AMD's implementation of OpenCL) in my laptop, I tried to run the examples. Most of the ran successfully, but there were some that did not work.

Example:

```
/opt/AMDAPP/samples/opencl/bin/x86_64$ ./GlobalMemoryBandwidth
Platform 0 : Advanced Micro Devices, Inc.
Platform found : Advanced Micro Devices, Inc.

Selected Platform Vendor : Advanced Micro Devices, Inc.
Device 0 : Redwood Device ID is 0x155c550
Error: clCreateBuffer failed due to inavailability of memory on this
  platform. (inputBufferExtra) Error code : CL_INVALID_BUFFER_SIZE
Location : GlobalMemoryBandwidth.cpp:330
```

Not enough memory? My GPU has 1GB.

Another clue about my memory problem came with the use of Darktable (an open source photography workflow application and RAW developer) with OpenCL disabled despite I had AMD APP SDK installed.

After digging Internet I read some user observed that Radeon cards report to have less memory than they physically own (512MB out of 1GB). I also read that Radeon cards have less memory overhead.

Then I looked for something to broke these limits and the response was in the environment variable: GPU_MAX_HEAP_SIZE. This variable represents the percentaje of memory to report.

Initially I set it to 100. AMD APP sample ran well, later I tried with Darktable equalizer module because it is a hot candidate to have problems allocating memory and worked but no well. My laptop run slow, laggy. The next try decremented GPU_MAX_HEAP_SIZE value to 70 then samples worked well and Darktable ran smooth.

The open source project infrastructure

SimpleOpenCL is hosted on Google Code a place that provides a collaborative development environment for free open source projects.

Website URL: http://code.google.com/p/simple-opencl/

Website have 5 sections (showed as tabs in the header):

- Main page: You will find project information (license, members, downloads,...) and a description to the project with an example.
- Downloads section: The place where you can download current version or previous ones. It shows you information about files to download (size, release dates, download counters,...)
- Wiki: A centralized point where you will find all the documentation (manual, specification, FAQ and Contact information).
- Issues: Google code provides a tracker where users could report bugs, petitions or send patches.
- Source: Explain how to download the repository to work with the source code. Repository used is Subversion. Inside this repository you will find source code and wiki pages.

At first sight, as with any Google page, there is no complexity and almost every developer have a Google account (not needed to register in a new service).

Project contribution

This topic will reflex my contributions to SimpleOpenCL project.

This chapter includes a description of code and documentation contributions, and also some proposals and changes that could be done.

All source code issues and comments relationated with it will found within “Issues” section at SimpleOpenCL project site. Documentation changes will be reflected within “Wiki” section.

Note: At the time of writing this thesis all this issues could not be addressed because some of them needs a deep discussion.

Code review

While coded the simple previous program I realized that I needed to know all the SimpleOpenCL API to code fluently. Furthermore it will help me to find issues.

Then I spent some days reading the code and trying to understand every function. I also annotated the issues I saw in “Issues” section at SimpleOpenCL website.

Mostly are issues that could be fixed directly but others requires changes in the architecture that should be commented with Oscar, the author of SimpleOpenCL, to arrive to the best solution.

During this section we will revise every issue. The title of each issue follow the next format:
issue <number>: <description>

Being:

- number: Number of the issue at SimpleOpenCL website
- description: Issue summary

Issue 3: Trying SimpleOpenCL homepage examples

With my laptop ready to play with SimpleOpenCL I wanted to refresh some of my C skills. Then I decided to being for the homepage examples.

I copied both examples from the homepage to local files and tried to compile them but I could not, it gives me an error:

```
example.c:13:9: error: ‘worksize’ undeclared (first use in this function)
```

The error was clear, worksize was not declared.

I reviewed the code and fixed it:

```
--- a/example.c
+++ b/example_fixed.c
@@ -6,7 +6,6 @@ int main() {
    int found;
    sclHard hardware;
    sclSoft software;
+   size_t worksize=strlen(buf);

    // Target buffer just so we show we got the data from OpenCL
    char buf2[sizeof buf];
```

After fix it locally I opened a ticket and began to think in a workflow for this project including Git on it.

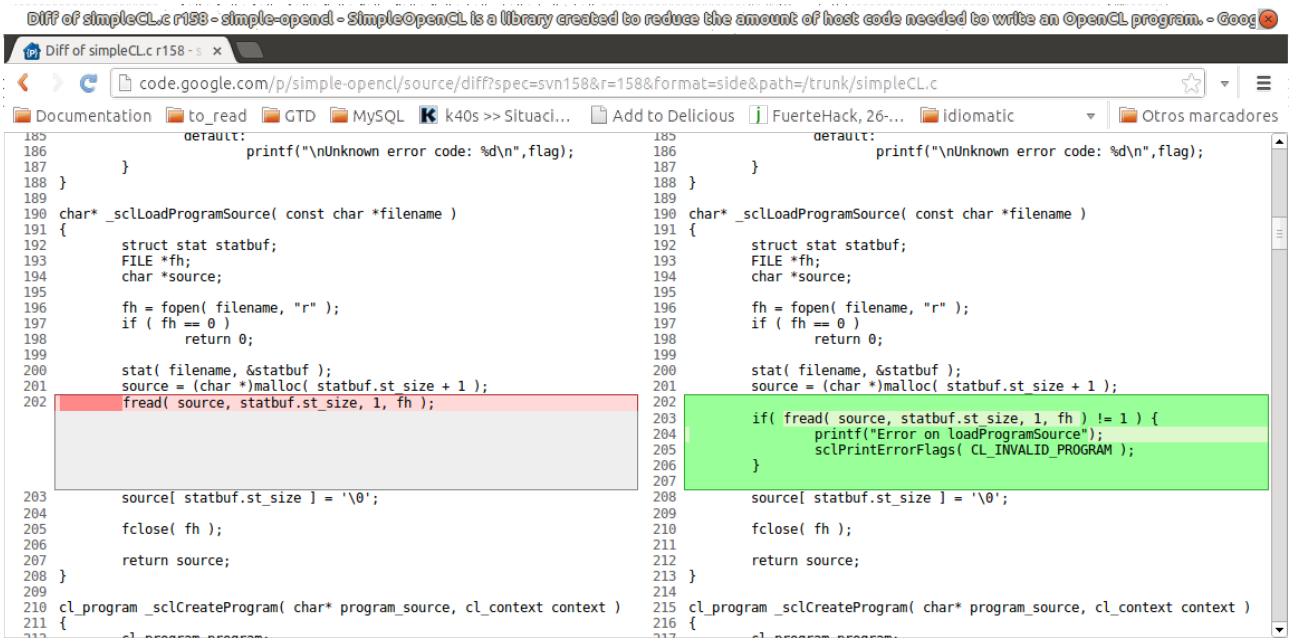
Issue 4: Compiling without warnings

When I fixed the source code and configured Subversion repository to work with/as Git, I compiled it and show me a warning:

```
simpleCL.c:202:7: aviso: se descarta el valor de devolución de ‘fread’, se
  declaró con el atributo warn_unused_result [-Wunused-result]
```

I was reading about fread function at a C++ reference webpage. Fread returns the number of elements successfully read and it is not being used at SimpleOpenCL.c file.

To fix it I only used the returned value to print an error message at runtime.




```
185                 default:
186                         printf("\nUnknown error code: %d\n",flag);
187         }
188 }
189
190 char* _sclLoadProgramSource( const char *filename )
191 {
192         struct stat statbuf;
193         FILE *fh;
194         char *source;
195
196         fh = fopen( filename, "r" );
197         if ( fh == 0 )
198                 return 0;
199
200         stat( filename, &statbuf );
201         source = (char *)malloc( statbuf.st_size + 1 );
202         fread( source, statbuf.st_size, 1, fh );
203         source[ statbuf.st_size ] = '\0';
204
205         fclose( fh );
206
207         return source;
208 }
209
210 cl_program _sclCreateProgram( char* program_source, cl_context context )
211 {
```

```
185                 default:
186                         printf("\nUnknown error code: %d\n",flag);
187         }
188 }
189
190 char* _sclLoadProgramSource( const char *filename )
191 {
192         struct stat statbuf;
193         FILE *fh;
194         char *source;
195
196         fh = fopen( filename, "r" );
197         if ( fh == 0 )
198                 return 0;
199
200         stat( filename, &statbuf );
201         source = (char *)malloc( statbuf.st_size + 1 );
202
203         if( fread( source, statbuf.st_size, 1, fh ) != 1 ) {
204                 printf("Error on loadProgramSource");
205                 sclPrintErrorFlags( CL_INVALID_PROGRAM );
206         }
207
208         source[ statbuf.st_size ] = '\0';
209
210         fclose( fh );
211
212         return source;
213 }
214
215 cl_program _sclCreateProgram( char* program_source, cl_context context )
216 {
```

Image 4: Issue 3 diff

Issue 5: _sclLoadProgramSource (called from sclGetCLSoftware) does not validate path

_sclLoadProgramSource doesn't validates passed path, continues the execution and failing in other places.

These kind of exceptions (errors) should be handled gracefully and print an error message to the users. But we have a problem with the term “gracefully”, what it mean for this library? It should be discussed later(check issue 14).

The fix proposed for this issue is to notify to the user with an error message. It will let to the user identify the problem better.

Issue 6: _sclBuildProgram uses fprintf not printf

This one is a minor change but gives consistency to the library. _sclBuildProgram uses printf to notify user about errors but in one line uses fprintf:

```
...
printf( "Error on buildProgram " );
sclPrintErrorFlags( err );
```

```
fprintf( stdout, "\nRequestingInfo\n" );
clGetProgramBuildInfo( program, devices, CL_PROGRAM_BUILD_LOG, 4096, build_c,
  NULL );
printf( "Build Log for %s_program:\n%s\n", pName, build_c );
...
```

It would be appropiate to change “fprintf” by a “printf” at this moment (I said it because issue 16 propose to use stderror to print errors).

Issue 7: Not initialized 'myEvent' variable is being used

In some functions 'myEvent' variable is defined and only initialized if DEBUG is defined . But 'myEvent' is always returned (initialized or not).

It could be others, but affected functions detected:

- sclLaunchKernel
- sclEnqueueKernel

A possible fix it is to initialize 'myEvent' with null value.
Other solution that requires a benchmark is use 'myEvent' because perfomance could be affected.

Issue 8: Check consistent use of "ifdef DEBUG"

"ifdef DEBUG" pattern is the chosen to handle errors but not everywhere is used in the same way.

Example:

```
...
#ifdef DEBUG
  cl_int err;

  ... /* Do something */

  if ( err != CL_SUCCESS ) {
        printf( "\n Do something Error\n" );
        sclPrintErrorFlags( err );
  }
```

```
  #else
    buffer = clCreateBuffer( hardware.context, mode, size, NULL, NULL );
  #endif
  ...
```

I have found functions that don't use it (could be more):
- sclReleaseMemObject
- sclPrintHardwareStatus
- sclGetAllHardware
- sclGetGPUHardware (commented lines)
- sclGetCPUHardware (commented lines)

Fix should be detect pieces of code that do not use “ifdeb DEBUG” pattern and implement it.

Issue 9: Typo error in sclGetAllHardware function

A minor change, fix a typo in a literal.

Original string:
"No OpenCL pla**n**tforms found."

Fixed string:
"No OpenCL platforms found."

Issue 10: Review scl*Hardware functions code

Based on the Don't Repeat Yourself principle (also knows as DRY), a deep review of sclGetAllHardware, sclGetGPUHardware and sclGetCPUHardware functions could help us to refactor these methods that share code. There is a lot of code to reuse.

It will do maintenance of the library easier.

Issue 11: Check returned value of sclMalloc and sclMallocWrite

SclMalloc and sclMallocWrite does not handle exceptions and could lead to bugs or errors in others

places at runtime execution.

Fix should be to handle exceptions but we have the previous problem about how to do it. A definition about “handle exceptions” is needed.

Issue 12: sclMallocWrite could call sclMalloc function

Both functions share same code.

Being dry we should refactor it and I find call sclMalloc from sclMallocWrite our best option.

Issue 13: sclSetArgsLaunchKernel and sclSetArgsEnqueueKernel could call to sclSetKernelArgs function

Problem here is the same thhat issue 12.

To fix this ticket I'll apply the same solution as issue 12, call sclSetKernelArgs function(which shares code with the others) from sclSetArgsLaunchKernel and sclSetArgsEnqueueKernel.

Issue 14: Show error only in DEBUG mode but handle it anyways

This issue needs a long discussion about how to handle errors . We need to define an standard form to do it.

Despite of this could be fine to show only error messages when DEBUG mode is enabled (but also needs a discussion).

I does not have a complete solution to it, but for now my idea is abort the execution in error case and release memory, I need to speak more with Oscar (SimpleOpenCL author).

Issue 15: Optimization: use pointers as parameters not structs

Some functions signature are defined passing sclHard and sclSoft structs values with its costs.

Maybe passing it by reference could improve performance, of course benchmark needed.

Issue 16: Use stderror for printing errors

Use standard output for printing information and warnings or errors is not a good practice. i.e. Using different channels let you, as user, log errors in one file and visualize information on the screen.

We can change every printf for error cases by fprintf and redirect the output to stderror.

Issue 17: Release objects (free memory)

All the documentation and specification read told me about release kernel, program, memory objects, command queue and context.

Reviewing the library I saw some places where it's done but others where it is not done.

It should be done always to avoid memory leaks but also in this case it will let us abort main program on error (it is not possible now because we are not sure if everything is released).

Documentation

SimpleOpenCL documentation is very good but is not completed. It is not a big problem because the library currently is small enough to read the code easily.

At website project you will find four wiki pages: Manual, specification, FAQ and ContactInformation. Let's ignore the last two.

“Manual” explains in a simple and concise way how to begin to work with the library. It tells you how from basics functions to select devices, load code execute and execute code to compile. It also provides another example. A good place to begin.

“Specification” is the place where you begin to code and needs to have a reference manual. It is divided four parts (actually more but these ones are the most important):

- SimpleOpenCL Types: Explain the structs related with the devices (hardware) and program and kernel objects (software).
- First level user functions: Explain high level functions to avoid deal with OpenCL complexity (memory management, host concepts, manage kernels...).
- Second level user functions: This functions are used for first level functions but could be used by developers. It gives you functions to work with memory objects, debug, event queries, queue management... With these level functions you have more power without add too much complexity.
- Third level user functions: This section is not done yet. But it is intended for explain functions to work at low level.

Proposals

At issues sections I enumerated some things to do at code level and wrote a proposal for fix them. But there are other kind of proposals with could not be grouped there.

This chapter is dedicated to these things that are not related with the source code directly, all these proposals are designed to help SimpleOpenCL project grows within the open source world.

Wikipage with contributing information

To encourage developers for contributing to SimpleOpenCL project could be easy to add a page with information and steps to contribute.

It should be a simple page with a concise content.

It could have these sections (without order and may have others):

- How to use Google code to report issues
- How to help resolve existing issues (send patches)
- How to download the repository
- Coding guidelines
- …

Git for the win

Some years ago I was working with Subversion but I discovered Git and I do not want to go back. The advantages of Git over Subversion are better explained at the Appendix B but my favorites are it is very fast and I do not need to be connected to work with it.

Git offer a tool to integrate(somehow) both SCMs, it is called git-svn. With git-svn I can work in the same way that I do with a Git repository but when I push my changes (push = in subversion world is a checkout) it goes to Subversion repository. I'm using it at this moment, works well but I personally prefer Git directly.

Google Code let you use Git repositories instead of Subversion. But for migrating from Subversion to Git it need a little planning because wiki pages are inside of the repository.

I think the steps are:

1. Go to “Admin” section, choose “source” sub-tab and select Git repository. With this change your new repository and wiki pages will be empty.
2. Move your Subversion history to Git with git-svn (basically clone your Subversion repository).
3. Push the changes to origin (“origin” is the name used for the main Git server).
4. To migrate the wiki we need to move its history to Git with git-svn.
5. Push the changes to the origin for the wiki (it is separated from the source the code)

Note: Migrate source code and wiki is more or less the same but goes to different repository (Subversion stored it in different directories instead of repositories).

Move from Google Code to GitHub

When I worked with projects hosted on Sourceforge or Google Code, it does not give too much reasons to work with it. Too much big-projects-old-school-management (download repository, open issue, fix, generate patch, send patch, … too much context changes and tasks).

Move to social coding! Let's move to a development environment that encourages discussion and sharing likes GitHub.

GitHub is Git based. It has a beautiful interface and the “social” aspect that makes easy and fun to collaborate with projects.

Some features, different from Google Code, are:

- Pull request system works great.
- Notifications system is very convenient.
- Show the code in the right way.
- Network graphs (to know what is going on in other people's branches).
- Social aspects
- Fork and pull request (Extremely easy to collaborate, I will develop this below).

To me the killer feature is “Fork and pull request”. Collaborate in a project only consists in do a fork of this project, make changes and send a pull request. It is as simple as that.

As a developer to begin collaborating within a open source project should be the most trivial task, GitHub achieved it. I collaborated in several projects just with minor changes because is fast and comfortable.

I think to move to GitHub should be a must because I guess will have more contributions to SimpleOpenCL project.

Note: I'm biased because I worked several years in Capistrano open source project hosted on GitHub. There are others services as BitBucket, but GitHub is the biggest in community and projects (i.e. Linux kernel is hosted there).

Conclusions

In my opinion, SimpleOpenCL is not ready for production environments or big projects for possible memory leaks, some details in its implementation(error handling) and it needs a little work on documentation.

But for education purposes I think is great. I can imagine a mathematicians, biologist and other people using it for his complex computations without have to handle with OpenCL. This persons could(and should) create community and evolve this project.

SimpleOpenCL is a library with a clear goal, abstract the developer from OpenCL complexities. This library already achieve it but it has a long way to walk.

Future works

There are future works to do for SimpleOpenCL project without timeline or close in the time.

Some ideas:

- Benchmarks to test library performance using different level functions or just to be sure that changes does not affect to performance.
- Add test to the projects. I was looking for a test library for C code, there are several and seems easy to use. It will let you make changes in the library without broke other parts. The drawback to add tests to the project is too much hardware oriented. Test hardware is not easy you could mock-up responses but it is not easy and need a deep analysis.
- Could be interesting have a function that get the faster device. The current function works but it is not reliable at all, only count compute units without take into account ALUs. i.e. a Nvidia GPU could have 192 ALUs and 2 cores and a Intel CPU 4 ALUs and 4 cores and this function will return as fast the CPU.
- Avoid performance problems due to create and destroy buffers every time a kernel is launched too much times.

Appendix A: Version control systems

Version control systems, also known as revision control or source code management, are tools or practices mainly dedicated to maintain a history of changes over the source code.

Version control tracks the complete development life of a project, for example: exact changes which have occurred between a period (commit, dates, releases, …), who made those changes, simplify release notes generation … Allow explore changes which resulted in each step of the version control history is very useful to understand who made a change, when it happened and why it happened.

Version control also exists to help the collaboration between users (Note: I don't want say “developers” because its use in the past was only for them but now its used by system administrators, graphic designers and other professionals). It prevents one user from accidentally overwrite changes from another, allowing many users to work in the same code without stepping on someone else's work (In case of conflict it offers tools to solve it).

Version control systems is a must nowadays.

Basic concepts

Repository

The set of files that are under version control system are commonly called “Repository”. It is a place where store system's data (different versions of files and directories, updated data, changes,...).

The repository usually stores information in the form of a filesystem tree or database system. Depending on the kind of version control system the repository could be centralized, i.e. stored in a server, or distributed among the different working copies.

Repository's working is similar a file server with steroids. Clients connect to the repository, and then read or write resources(files). Each resource modified/written at the repository is available for all the users. Users, by reading the repository, get the updated information from the repository. All theses modifications are stored as historic, having a new version of the resource for each change.

Tracking changes

As you can see version control systems is mostly based around this concept “Tracking changes”. In most cases, you specify a set of files or directories(resources) that should have their changes tracked in the repository. Since you begin to modify resources it will track each change.

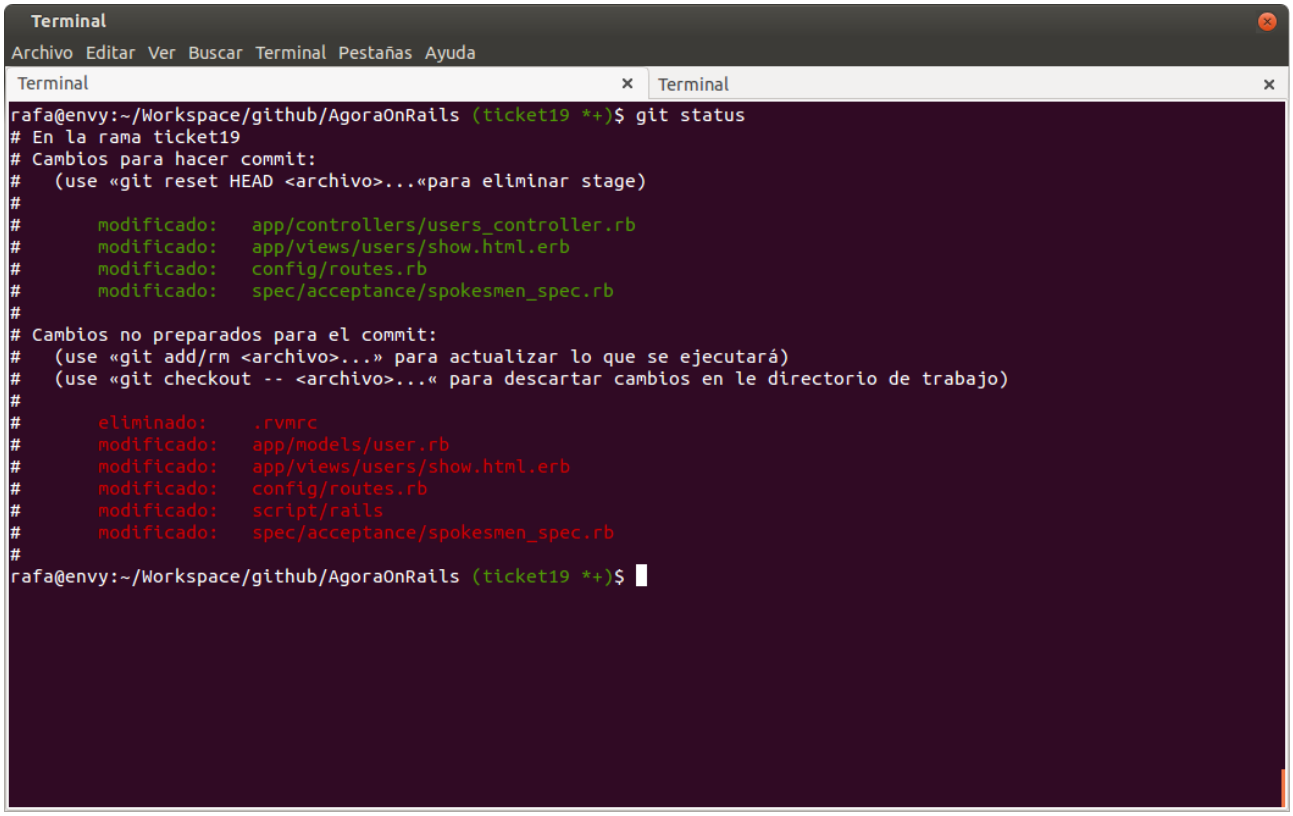


Image 5: "git status" command with pending changes

Committing

The action of recording at the repository every tracked change is to commit.

When you remove a directory, modify a a directory or set of files, add a new file(to be tracked), … anything else that might alter the state of the tracked resource or resources, the version control system will wait for you to submit your change in a single transaction called commit.

Revisions and changesets

Revisions are unique IDs for changesets. Revision could be an incremental number (1, 2, 3, 4, 5, …) or a unique hash (9566413, 852d31f, 852d31f7d95dd69c55505ec71f0ae02e34651003, …) depending on version control system. By knowing revision IDs we can view or reference it later.

A changeset from wikipedia: "... is a set of changes which should be treated as an indivisible group (i.e. an atomic package); ... ". In a version control system a changeset usually contains information about who made the commit, when the changeset was made, resources affected, a comment and the lines modified (lines of code for source files).

Viewing past revisions and changesets is a valuable tool to see how your project is evolving. Version control systems has a more or less confortable way to view the history or log for each

revision and changeset in the repository.

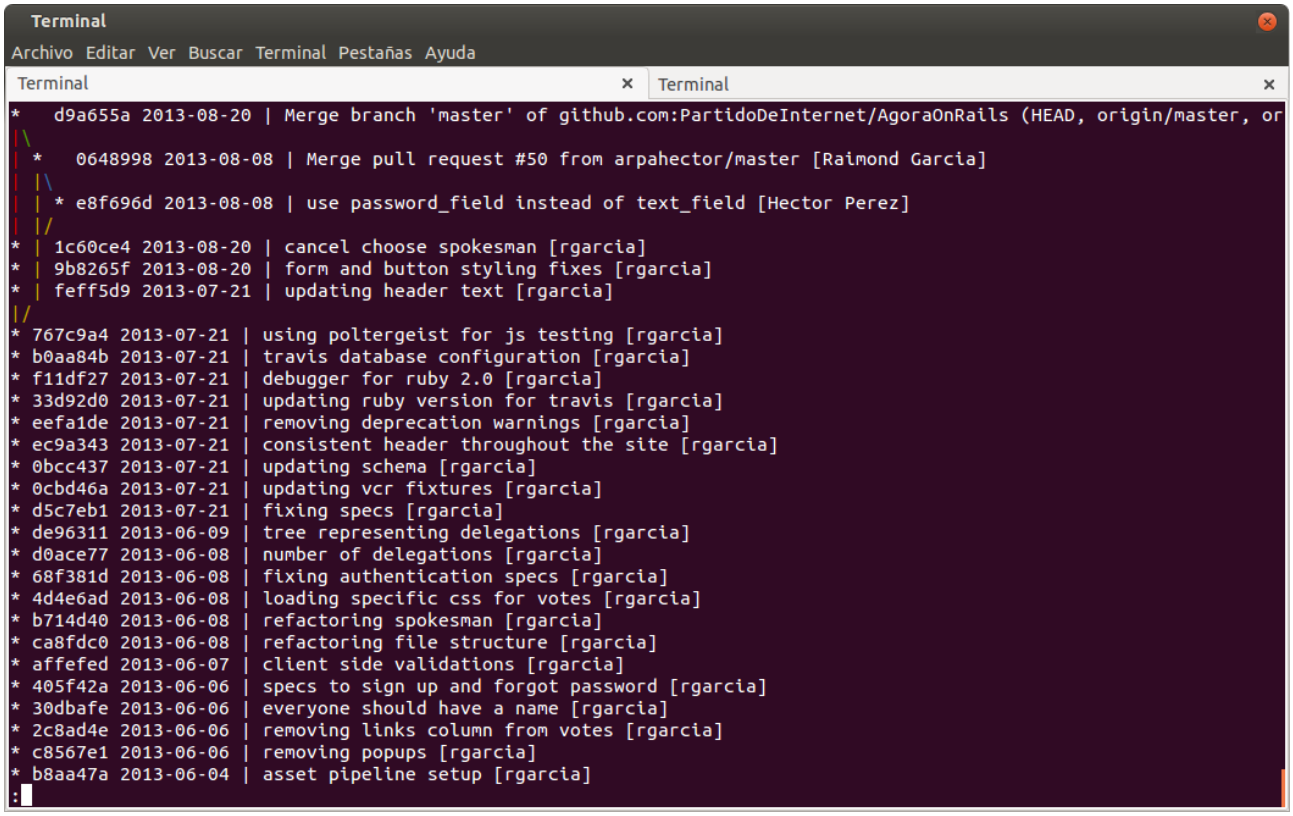


Image 6: Git history example (Note: I'm not rgarcia committer, he's Raimond)

Getting updates

It is important that you have the latest changes in your working copy (the repository local copy) along the time. Getting the latest changes is as simple as doing a pull or update from another computer (usually a server).

When an update is requested, only changes since your last request are downloaded from the repository.

Conflicts

Changes could be very similar or affects the same lines in a resource (i.e. modify same function in a C file) and version control system might not know how to determine which is the correct change. For these cases version control system provide you several ways to resolve conflicts.

It's a good thing that current version control systems do a good work resolving conflicts automatically.

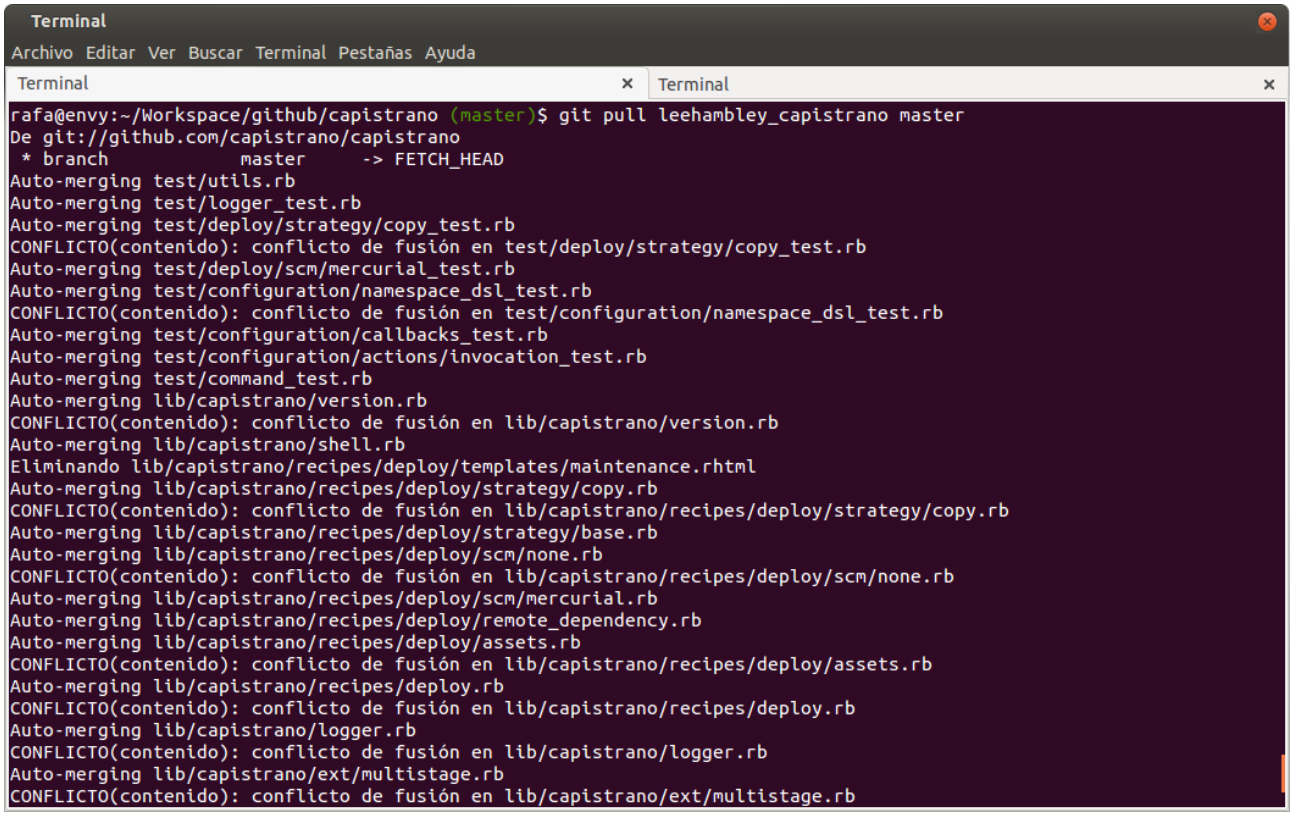
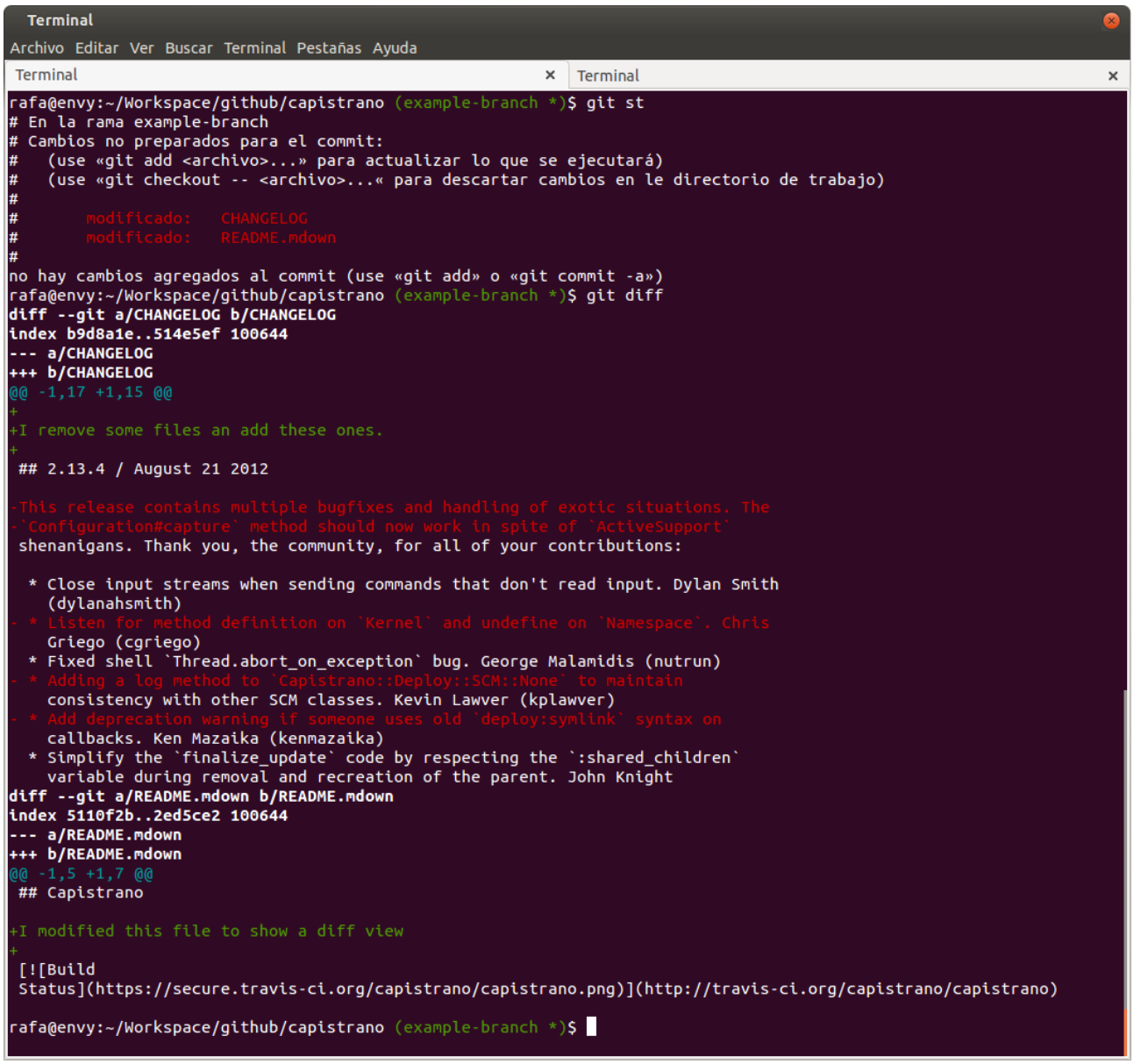


Image 7: Problematic merge

View differences

Viewing the differences is also called diffing. Sometimes it is useful to check what changed between revisions. Current version control systems let you diff sequential revisions, but also two revisions from anywhere in the history.



Branching

A branch is a snapshot, clone or fork within your own repository. Your branch will have an ancestor commit in your repository and will diverge from that commit with your new changes(commits).

It helps you a lot when you need to work in a experimental features and you don't want to impact in main branch (commonly called master or trunk) because this code maybe never will be production ready.

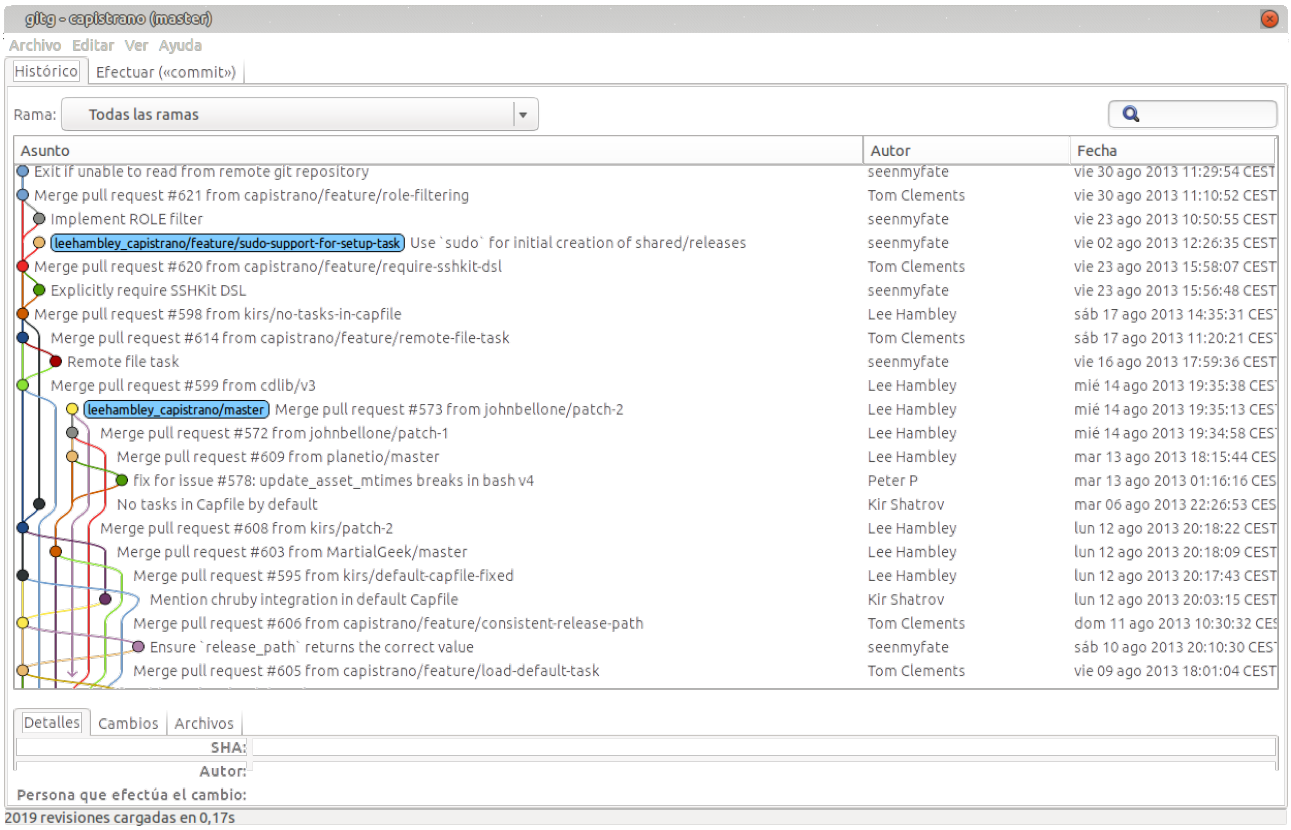


Image 8: branches visualization with gitg (Gnome GUI client to view git repositories)

Merging

Merge is the action that you do when you want to incorporate your branched changes in master (or another branch).

Version control system will try to add committed changes from the branch into the desired branch automatically but sometimes conflicts could happen and you should fix before being included.

Types of version control systems

The two main categories of version control systems are centralized and distributed.

Centralized version control systems are used these days but feels as something old fashioned or legacy but are not dead.

On the other hand you have distributed version control system as the new kid on the block but in fact are the evolution of the centralized systems.

Examples of this control systems are Subversion or CVS (Concurrent Version System).
Examples: Git, Mercurial, Bazaar, …

Centralized version control

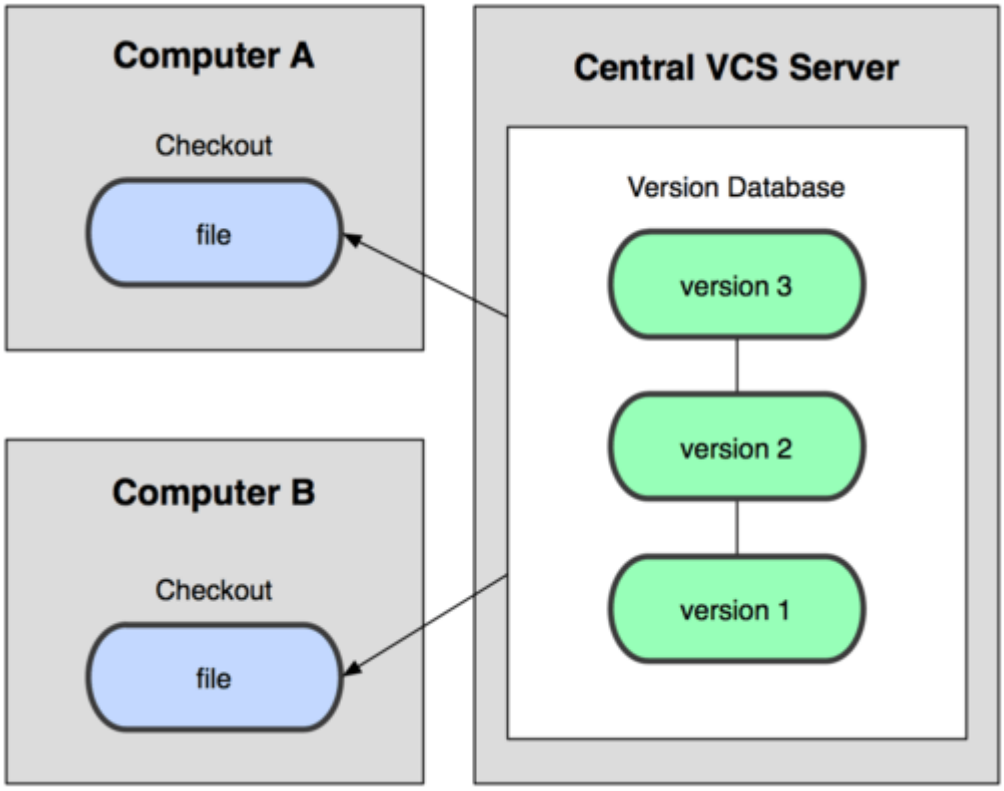


Image 9: Server and clients - Courtesy of Git SCM (MIT license)

The best way of thinking about centralized version control systems are in a client-server relationship. As I said previously it works as a file server. Server stores the trunk repository and clients connecto to the server. All changes must be sent and received from this centralized

repository.

Main benefits are:

- Easy to understand
- More control over users and access
- More GUI & IDE clients
- Simple to get started

Drawbacks:

- Dependent on access to the server
- You should manage a server (maintenance tasks, backups, updates,...)
- Slower because every command connects to the server
- Branching and merging tools are difficult to use and inefficients

Distributed version control

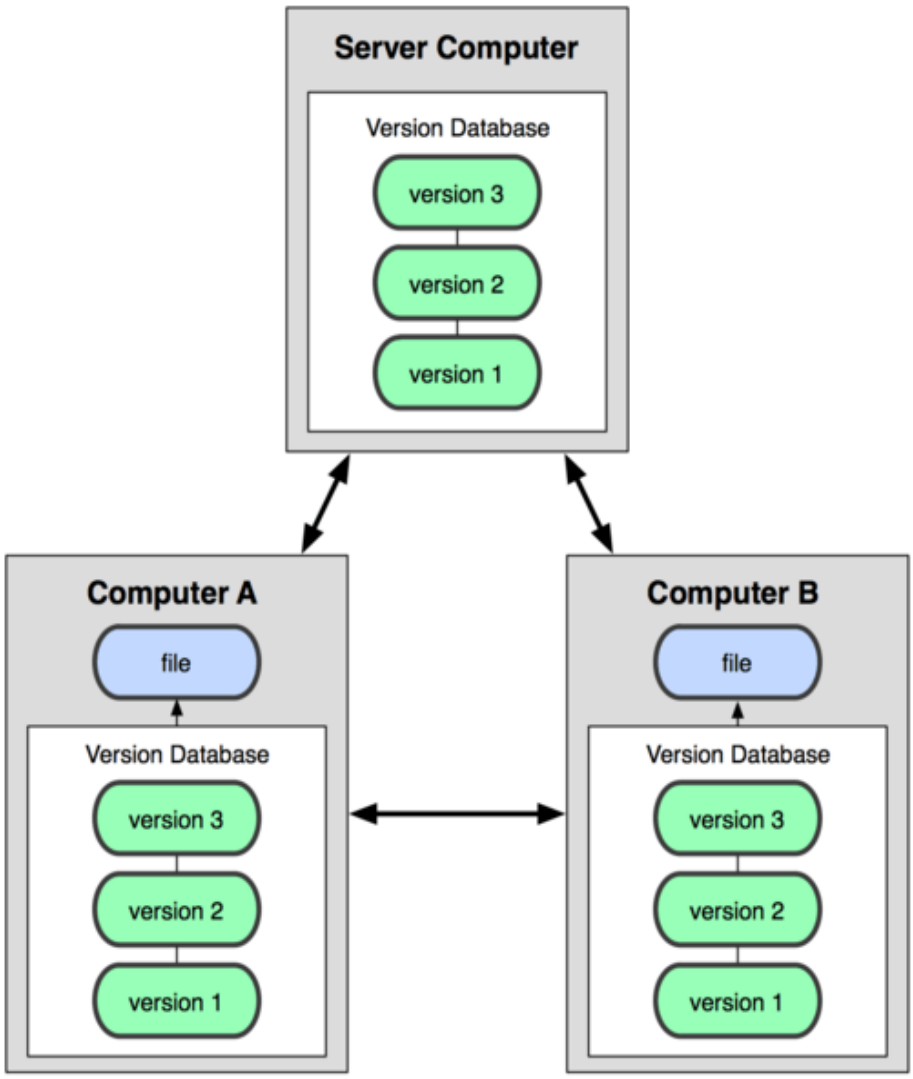


Image 10: Distributed version control - Courtesy of Git SCM (MIT license)

In distributed version control systems each user have a complete copy of the repository (a working copy as we called it previously) with files and history of changes. It is a network of individual repositories.

Despite a server is not needed several services appeared for sharing changes between repositories(i.e. Github or Beanstalk).

Main benefits are:

- Powerful and detailed change tracking
- No server necessary (All actions are done locally except share/push changes)
- Branching and merging more reliable
- It is fast

Drawbacks:

- It is harder to understand this model
- Not so many GUI clients
- Revisions are hashes, not incremental numbers
- Initial cloning could be slower

Appendix B: Git

From his page(http://git-scm.com/):

“Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is easy to learn and has a tiny footprint with lightning fast performance. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows.”

Git was initially designed by Linus Torvalds thinking on speed. That was because Linus was looking for a replacement for the current SCM used at Linux kernel project (a lot of files and people collaborating).

Features

Branching and merging

Git encourages you to work with multiple local branches. To support non-linear development(to have parallel branches), branches management is a simple and a day-to-day task.

It let you do things like:

- Context switching: Create branch to try ideas, back to master branch to apply a patch, back again to the experimental branch.
- Role based codelines: Have different branches for different environments (development, production, testing, …)
- Feature based workflow: Create new branches for new features and merge them when they are ready
- Disposable experimentation: Create a branch to experiment in, abandoning the work with nobody seeing it.

Small and fast

Nearly all operations are done locally, giving it a big advantage against centralized systems. Git is written in C language, reducing overhead of runtimes associated with high-level languages. Speed and performance has been the primary goal of its design from the start.

Distributed

You don't do a “checkout” of the current repository, you do a “clone”. Each cloned repository is a valid backup of your repository hosted in a main server.

Due to the distributed nature Git can adopt any workflow with relative ease.

Data assurance

Directly from Git website: “The data model that Git uses ensures the cryptographic integrity of every bit of your project. Every file and commit is checksummed and retrieved by its checksum when checked back out.”

Staging area

The intermediate area where commits can be reviewed or rewritten before to be pushed to main server.

Free and open source with a big community

Git is released under the GPLv2 license. You are free to inspect the code or collaborate with the project at any time.

Recommended to read Community section at Git website.

Getting started

The idea of this section it's to have a reference guide to survive with a basic usage of Git.

I have deliberately omitted the installation because it is very well documented here:
http://git-scm.com/book/en/Getting-Started-Installing-Git

Configuring / Setting up

After installing Git you need to set your identity: **username and email**. This information will be used for Git commits.

```
$ git config --global user.name "Foo Bar"
$ git config --global user.email foo@bar.com
```

We are using "--global" parameter to set it globally else it only will be set for the current repository.

To get more information about "git config" you could read the manual with "man git-config" or "git help config".
Every git command have its own manual page and could be read with "git help <command>" or "man git-<command>".

Other configuration sets that I consider important are your editor (used i.e. when you write a commit message or work with the history) and the diff tool to resolve merge conflicts.

To configure your editor:

```
$ git config --global core.editor vim
```

If you don't configure any editor system's default will be used.

To configure your diff tool:

```
$ git config --global merge.tool vimdiff
```

Git accepts kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, and opendiff as valid merge tools. It also can accept custom diff tools but it is out of the scope of this document.

To list/check all configuration variables:

```
$ git config -list
user.name=Foo Bar
```

```
user.email=foo@bar.com
merge.tool=vimdiff
...
```

To check a configuration variable:

```
$ git config user.name
Foo Bar
```

Getting a repository

You have two choices to get a Git repository: Use your current project or directory and import it to Git or clone an existing repository from another server.

Initializing a repository from the current project:

```
$ git init
```

It will create a .git subdirectory that contains all your necessary repository files. At this point, nothing is tracked yet.

Cloning an existing repository:

```
$ git clone <url>
```

You will get a copy of an existing Git repository. If you are used to Subversion will notice that the command is clone and not checkout, it is an important distinction. With git clone you get a copy of nearly all data that the server has, everything is pulled down.

Repository status

The command for checking the status of your files is “git status” and depending on the file status the output will be one or another.

If you run the command in a empty repository (after a “git init” in an empty directory):

```
$ git status
# On branch master nothing to commit
(working directory clean)
```

That means there are no tracked files modified or untracked files.

Tracking files

Let's to create a README file and track it:

```
$ touch README   #create a README file
$ git add README
```

You can see README is now tracked and staged:

```
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# new file: README
#
```

Note that README is at “Changes to be committed” section.

Stage changes

Imagine that we modify a example.rb and it's a tracked file and the we run “git status” command, the output will be:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   example.rb
#
```

It means that README is tracked but not committed and example.rb has been modified but it is not staged (ready to be committed).

To stage “example.rb” file we use “git add”:

```
$ git add example.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   example.rb
```

Note: If you modify a file after stage it, you have to stage it again to have the latest version of the file.

Commit

Now we have all our changes staged we can **commit our changes**:

```
$ git commit
```

This will open our editor (set previously or from $EDITOR variable) for writing the commit message.

I normally commit with "git commit -v" that show a diff between the HEAD of the commit and what would be committed.

You can use "-a" parameter to commit to include all tracked and modified files.

There are a lot of post related how to write a good commit message, some of them:

- http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html
- http://robots.thoughtbot.com/post/48933156625/5-useful-tips-for-a-better-commit-message

Ignoring files

Another essential feature for a good SCM is the possibility of ignore files.
Sometimes our IDEs, editor or the current project generates a lot files(i.e. logs) that must not be in the repository (not be added automatically or show as untracked) and should be ignored.

You can create a file with a pattern match list and call it as ".gitignore". For example:

```
$ cat .gitignore
coverage
```

```
doc
pkg
*.swp
*.patch
*.gem
*.sh
.DS_Store
.bundle/*
Gemfile.lock
```

Undoing changes

If you did something wrong in a file/directory you can **drop local changes** with the command:

```
$ git checkout <filename>
```

Undone commits and leave changes staged:

```
$ git reset --soft HEAD^
```

If instead you want **to drop the latest commit(!)**:

```
$ git reset --hard HEAD^
```

Notes:

- *"^" means previous revision. It could be write as "~1".*
- *"^^" means two previous revisions. It could be write as "~2".*
- *... and so on.*

Understanding it, you can drop ten last commits:

```
$ git reset --hard HEAD~10
```

Branches

We will cover the basics of the "git branch" command: create, list and delete branches.

Creating branches and switching to it:

```
$ git branch <name>
$ git checkout <name>
```

Create a branche a switch into it is a very common pattern and exists a shortcut:

```
$ git checkout -b <name>
```

Once you have done your work in a branch you need to incorporate it with you master branch. Then we switch to master and **merge the branch**:

```
$ git checkout master
$ git merge <branch-to-be-merged>
```

In the same way you can merge a branch with another, it is not mandatory to use master.

Listing branches:

```
$ git branch
experimental
feature1
feature2
* master
test-zoom
```

Branch marked with "*" is the current branch.

Deleting branches:

```
$ git branch -d <name>
```

Tags

A known concept it to create tags for software releases.

You can create a new tag(referencing to current revision):

```
$ git tag 1.0.2
```

Or if you want specify a revision:

```
$ git tag 1.0.3 <commit-id>
```

Note: Commit-id is the revision hash, it don't need to be the complete string with 5-10 characters are enough.

Pull changes

You can update your working copy (local repository) with “git pull” command. This command execute to actions: fetch new commits from the remote repository and merge them in the current branch.

```
$ git pull
```

Note: By default, if you cloned the repository or set upstream , pull will get the updates from origin/master (server pointed from origin and branch master).

Pushing changes

You can update the remote repository with “git push” command.

```
$ git push
```

Note: Push command have the same default behavior as pull, pushing changes from the working copy to origin/master.

Basic workflow

With all these previous commands we are ready to use a simple and basic workflow for daily working:

1. pull all the changes from the remote repository
   ```
   $ git pull
   ```

2. create a new branch for your feature/bug/experiment
   ```
   $ git checkout -b branch-name
   ```

3. DO YOUR STUFF
It's better to Keep it in small and atomics commits

4. Add your changes and new files
   ```
   $ git add .
   ```

5. See the changes you're going to commit
   ```
   $ git status
   ```
and/or
   ```
   $ git diff
   ```

6. Make the commit with a good and detailed message
   ```
   $ git commit -v
   ```

7. Go to step 3 if needed else continue to step 8

8. Switch back to master branch when all your stuff is done
   ```
   $ git checkout master
   ```

9. Update master branch with all your changes
   ```
   $ git merge branch-name
   ```

10. Go to step 2 if needed else continue to step 11 (last)

11. Send your changes to the remote repository
   ```
   $ git push
   ```

Of course you can use a more elaborated one, there are a lot of workflows explained on Internet. I encourage you to look for them and give a try.

Appendix C: Installing OpenCL SDK

Installing OpenCL framework is not hard (because it is well documented on the Internet) but depends on your graphic card vendor and operating system, in my case AMD (I have an AMD GPU ATI Mobility Radeon™ HD 5650) and Ubuntu.

Installation procedure is written as a tutorial.

Tutorial

1. Take a look at your hardware to make sure it's compatible

To find what kind of graphic card my laptop had, I did:

```
$ lspci -v | less | grep VGA
```

I found my hardware and checked online to make sure it was compatible.

2. Install dependencies (just libglu1-mesa-dev in most cases)
Ubuntu install packages => g++,libglu1-mesa-dev,g++-multilib,ia32-libs,libtifiles2-5
(From ReadMe.txt)

3. Download and untar the AMD-APP SDK
Download SDK and drivers(if needed) from here:
http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/downloads/

Note: I downloaded AMD-APP-SDK-v2.8.1.0-lnx64.tgz

I recommend decompress it under a directory:

```
$ tar xzvf AMD-APP-SDK-v2.8.10-lnx64.tgz
```

4. Run the installation shell script and restart

Run the Install-AMD-APP.sh file to install APP.

```
$ sudo Install-AMD-APP.sh
============================================================
64-bit Operating System Found..
```

```
Starting Installation of AMD APPSDK v2.8.1.0 ....
 SDK package name is :AMD-APP-SDK-v2.8.1.0-RC-lnx64.tgz
Checking Latest Version Info.....
Continuando en segundo plano, pid 22913.
......

 **You are installing latest version of APPSDK 2.8.1214.3
 Continuing Installation...
================================================================
Current directory path is  : /home/foo/AMD-APP-SDK-decompressed
Untar command executed succesfully, The SDK package available
Untar command executed succesfully, The ICD package available
Copying files to /opt/AMDAPP/ ...
SDK files copied successfully at /opt/AMDAPP/
Installing AMD APP CPU runtime under /opt/AMDAPP/lib
Updating Environment variables...
32-bit path is :/opt/AMDAPP/lib/x86
64-bit path is :/opt/AMDAPP/lib/x86_64
Environment variables updated successfully
AMD APPSDK v2.8.1.0 installation  Completed
>> Reboot required to reflect the changes
================================================================
*****Please refer 'AMD_APPSDK_v2.8.1.0.log' in the same directory*****
*****Refer 'README.txt' for FAQ/help in the same directory********
```

Note: Here installation could be considered finished but we go another step forward and test it.

5. Copy samples and required files

From installation path copy samples files and other required files:

```
$ cp -R /opt/AMDAPP/make target/path
$ cp -R /opt/AMDAPP/samples target/path
```

6. Compile and run the test apps

Compile files from samples directory

```
$ make
```

When compilation is complete you can run sample executables from bin directory:

```
$ cd samples/opencl/bin/x86_64
```

7. Installation and testing done!

Bibliography - OpenCL

Links

http://www.khronos.org/opencl/

http://es.wikipedia.org/wiki/OpenCL

http://en.wikipedia.org/wiki/OpenCL

http://software.intel.com/en-us/vcsource/tools/opencl

https://developer.nvidia.com/opencl

http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/introductory-tutorial-to-opencl/

http://indico.cern.ch/getFile.py/access?sessionId=0&resId=0&materialId=0&confId=138427

http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf

http://my.safaribooksonline.com/book/programming/9780132488006/platforms-contexts-and-devices/ch03lev1sec3

Papers

SimpleOpenCL: making OpenCL programable by Oscar Amoros

Bibliography - SimpleOpenCL

Links

http://code.google.com/p/simple-opencl/
http://code.google.com/p/simple-opencl/wiki/Manual
http://code.google.com/p/simple-opencl/wiki/Specification
http://code.google.com/p/simple-opencl/issues/list
http://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/basic-program-flow/
http://en.wikipedia.org/wiki/Don't_repeat_yourself
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clGetPlatformIDs.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clGetDeviceIDs.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateContext.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateCommandQueue.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateBuffer.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateProgramWithSource.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clBuildProgram.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateKernel.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clSetKernelArg.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clEnqueueNDRangeKernel.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clEnqueueReadBuffer.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clReleaseKernel.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clReleaseProgram.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clReleaseMemObject.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clReleaseCommandQueue.html
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clReleaseContext.html

Bibliography - Version control systems

Links

https://en.wikipedia.org/wiki/Software_configuration_management
http://sourceforge.net/apps/trac/sourceforge/wiki/What%20is%20Source%20Code%20Management
http://www.slideshare.net/ctankersley/source-code-management-basics
http://svnbook.red-bean.com/en/1.7/svn.basic.version-control-basics.html
http://guides.beanstalkapp.com/version-control/intro-to-version-control.html
http://en.wikipedia.org/wiki/Changeset
http://en.wikipedia.org/wiki/Distributed_revision_control

Bibliography - Git

Links

http://git-scm.com/about
http://en.wikipedia.org/wiki/Git_(software)
http://es.wikipedia.org/wiki/Git
http://git-scm.com/community
http://en.wikipedia.org/wiki/GitHub
http://git-scm.com/book/en/Getting-Started-Git-Basics
http://gitref.org/index.html
http://www.slideshare.net/hbalagtas/simple-daily-workflow-with-git
http://marklodato.github.io/visual-git-guide/index-en.html