Simple-OpenCL Documentation

October 3, 2016

Contents

Chapter 1

Readme

1.0.1 Motivation and description

SimpleOpenCL is a library written in ANSI C and born in the needs of scientific research test development. It has been originated while developing different OpenCL codes for Linux and Apple test machines, with single device performance and portability goals in mind. The main goal of SimpleOpenCL has been reducing the code needed to run the experiments on the GPU, but also supports managing CPU devices. As this is an open source project, we welcome any contribution, from code correction and functionality suggestions/contributions to documentation or even contribution system proposals.

The main author have created and uses this library for its projects and aims to share it with those who want to use OpenCL with C or C++ for experimental codes in any sector. Nevertheless we think this can be an interesting library for academical purposes. With SimpleOpenCL the teacher can for instance focus on teaching Device code. And if desired later on the teacher could either show the library code to teach basic OpenCL Host code or propose student's to do more technical improvements over it or both.

We are not aiming to provide a CUDA-like interface for OpenCL, but there will be some similarities. The main author proposes a CUDA-like library for OpenCL as a separate and more production oriented project.

1.1 History

SimpleOpenCL is a library written in ANSI C and born in the needs of scientific research test development. It has been originated while developing different OpenCL codes for Linux and Apple test machines, with single device performance and portability goals in mind. The main goal of SimpleOpenCL has been reducing the Host code needed to run OpenCL C kernels on the GPU, but also supports managing CPU devices.

In a sentence: **The only thing a programmer needs to know in order to use SimpleOpenCL is C, two SimpleOpenCL types, and a minimum of three functions**. Device code (OpenCL C kernels) is exactly the same as in OpenCL.

1.2 License

1.3 Contributions

As this is an open source project, we welcome any contribution, from code correction and functionality suggestions/contributions to documentation or even contribution system proposals.

The Oscar Amoros has created and uses this library for its projects and aims to share it with those who want to use OpenCL with C or C++ for experimental codes in any sector. Nevertheless we think this can be an interesting library for academical purposes. With SimpleOpenCL the teacher can for instance focus on teaching Device code. And if desired later on the teacher could either show the library code to teach basic OpenCL Host code or propose student's to do more technical improvements over it or both.

1.4 Example

A comparative example:

1.4.1 SimpleOpenCL code version

```
#include "simpleCL.h"

int main() {
   char buf[]="Hello, World!";
   size_t global_size[2], local_size[2];
   int found, worksize;
   sclHard hardware;
   sclSoft software;
```

```
  // Target buffer just so we show we got the data from OpenCL
  worksize = strlen(buf);
  char buf2[worksize];
  buf2[0]='?';
  buf2[worksize]=0;

  // Get the hardware
  hardware = sclGetGPUHardware( 0, &found );
  // Get the software
  software = sclGetCLSoftware( "example.cl", "example", hardware );
  // Set NDRange dimensions
  global_size[0] = strlen(buf); global_size[1] = 1;
  local_size[0] = global_size[0]; local_size[1] = 1;

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

  // Finally, output out happy message.
  puts(buf2);

}
```

1.4.2 Now the same code but in plain OpenCL WITHOUT using SimpleOpenCL

```
#include <stdio.h>
#include <string.h>

#include <CL/cl.h>

int main() {
  char buf[]="Hello, World!";
  char build_c[4096];
  size_t srcsize, worksize=strlen(buf);

  cl_int error;
  cl_platform_id platform;
  cl_device_id device;
  cl_uint platforms, devices;

  /* Fetch the Platforms, we only want one. */
  error=clGetPlatformIDs(1, &platform, &platforms);
  if (error != CL_SUCCESS) {
```

```
  printf("\n Error number %d", error);
}
      /* Fetch the Devices for this platform */
error=clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device,
    &devices);
if (error != CL_SUCCESS) {
            printf("\n Error number %d", error);
}
      /* Create a memory context for the device we want to use */
cl_context_properties properties[]={CL_CONTEXT_PLATFORM,
    (cl_context_properties)platform,0};
/* Note that nVidia's OpenCL requires the platform property */
cl_context context=clCreateContext(properties, 1, &device, NULL, NULL,
    &error);
if (error != CL_SUCCESS) {
  printf("\n Error number %d", error);
}
      /* Create a command queue to communicate with the device */
cl_command_queue cq = clCreateCommandQueue(context, device, 0, &error);
if (error != CL_SUCCESS) {
  printf("\n Error number %d", error);
}

      /* Read the source kernel code in exmaple.cl as an array of
          char's */
char src[8192];
FILE *fil=fopen("example.cl","r");
srcsize=fread(src, sizeof src, 1, fil);
fclose(fil);

const char *srcptr[]={src};
/* Submit the source code of the kernel to OpenCL, and create a
    program object with it */
cl_program prog=clCreateProgramWithSource(context,
                                        1, srcptr, &srcsize, &error);
      if (error != CL_SUCCESS) {
            printf("\n Error number %d", error);
      }

/* Compile the kernel code (after this we could extract the compiled
    version) */
error=clBuildProgram(prog, 0, NULL, "", NULL, NULL);
if ( error != CL_SUCCESS ) {
  printf( "Error on buildProgram " );
  printf("\n Error number %d", error);
  fprintf( stdout, "\nRequestingInfo\n" );
  clGetProgramBuildInfo( prog, devices, CL_PROGRAM_BUILD_LOG, 4096,
      build_c, NULL );
  printf( "Build Log for %s_program:\n%s\n", "example", build_c );
}
```

```
/* Create memory buffers in the Context where the desired Device is.
    These will be the pointer
     parameters on the kernel. */
cl_mem mem1, mem2;
mem1=clCreateBuffer(context, CL_MEM_READ_ONLY, worksize, NULL, &error);
if (error != CL_SUCCESS) {
  printf("\n Error number %d", error);
}
mem2=clCreateBuffer(context, CL_MEM_WRITE_ONLY, worksize, NULL,
    &error);
if (error != CL_SUCCESS) {
  printf("\n Error number %d", error);
}
/* Create a kernel object with the compiled program */
cl_kernel k_example=clCreateKernel(prog, "example", &error);
if (error != CL_SUCCESS) {
  printf("\n Error number %d", error);
}

     /* Set the kernel parameters */
error = clSetKernelArg(k_example, 0, sizeof(mem1), &mem1);
if (error != CL_SUCCESS) {
  printf("\n Error number %d", error);
}
error = clSetKernelArg(k_example, 1, sizeof(mem2), &mem2);
if (error != CL_SUCCESS) {
  printf("\n Error number %d", error);
}
/* Create a char array in where to store the results of the Kernel */
char buf2[sizeof buf];
buf2[0]='?';
buf2[worksize]=0;

/* Send input data to OpenCL (async, don't alter the buffer!) */
error=clEnqueueWriteBuffer(cq, mem1, CL_FALSE, 0, worksize, buf, 0,
    NULL, NULL);
if (error != CL_SUCCESS) {
  printf("\n Error number %d", error);
}
/* Tell the Device, through the command queue, to execute que Kernel */
error=clEnqueueNDRangeKernel(cq, k_example, 1, NULL, &worksize,
    &worksize, 0, NULL, NULL);
if (error != CL_SUCCESS) {
  printf("\n Error number %d", error);
}
/* Read the result back into buf2 */
error=clEnqueueReadBuffer(cq, mem2, CL_FALSE, 0, worksize, buf2, 0,
    NULL, NULL);
if (error != CL_SUCCESS) {
    printf("\n Error number %d", error);
  }
  /* Await completion of all the above */
  error=clFinish(cq);
  if (error != CL_SUCCESS) {
    printf("\n Error number %d", error);
  }
  /* Finally, output the result */
  puts(buf2);
}
```

And this is without printing the names of the error flags, that would add a large function.

1.4.3 Kernel Code

Kernel code for both cases in OpenCL C (note that SimpleOpenCL doesn't alter Device code at all)

```
__kernel void example( __global char* buf, __global char* buf2 ){
       int x = get_global_id(0);

       buf2[x] = buf[x];

}
```

1.5 Contact

Original Author Oscar Amoros oscaramoros@ub.edu

Chapter 2

Roadmap

2.1 SimpleOpenCL Version 0.010 27/02/2013

- v1 Incremented with increased functionality up to a certain goal or amount of changes (new functions)
- vv2 Incremented with bug corrections and functionality additions.
- date Set at the date of bug correction or any v1 or vv2 level modification.

2.2 Actual version 0.08

- Non finished but working version. A first finished version is scheduled for version 1.0
- It provides two structs to simplify the handling of OpenCL objects. They are conceptually grouped in hardware and software.
- It provides simplified Device memory allocation and copy functionalities.
- It also provides a function that creates a list of devices using the "clHard" struct for each device, and creates a context for that devices. The context will be the same for same type and same memory capacity devices. If any of those differ, then different contexts will be created for each device.
- Other functions select the desired devices from the list.
- In order to get information of OpenCL errors, there is a function that prints the OpenCL error flags returned by the OpenCL functions.
- Functions to load Device source code, compile it etc are present.
- Functions to enqueue or execute kernels, etc...

- A main 1.0 version goal is already implemented as "sclManageArgsLaunchKernel". It can with only host pointers, sclHard and sclSoft variables NDRange dimensions and a string containing the info of what to do with the pointers etc, execute the kernel and update the results on the host pointers. All in a single function call.

2.3 Next version

Goals for version 1.0:

- All hardware selection functions must use the sclGetAllHardware function first if a list of hardware is not passed as an argument and it has NULL value. Then, the functions must return the desired hardware following the function criteria expressed in its own name.

- The sclManageArgsLaunchKernel can/must be improved in the following ways:

1. Optional: a version that hides sclHard and sclSoft so the user only cares about which host pointers will be read and written from the device in OpenCL C code and which to use exclusively on the device. Nothing else. Think of when to initialize hardware? Possibly needing a global variable pointing to the sclHard and sclSoft objects to avoid repeating the software/hardware initialization process on each kernel execution.

2. Mandatory: the function must have the ability to schedule work across all the devices available. So maybe the function will need more info from the user to know which data can be partitioned, and which can not. The possibility of internally using something similar to GMAC would be wonderfull. Pagination of the pointers in my opinion is the most efficient automatic method to do that work, but repeating a work already done would be frustrating.

Chapter 3

simpleCl.h API

3.1 data types

```
*/
typedef struct {
  cl_platform_id platform;
  cl_context context;
  cl_device_id device;
  cl_command_queue queue;
  int nComputeUnits;
  unsigned long int maxPointerSize;
  int deviceType; /* deviceType 0 = GPU | deviceType 1 = CPU |
      deviceType 2 =
                   Accelerator | deviceType 3 = other */
  int devNum;
} sclHard;
typedef sclHard *ptsclHard;
typedef struct {
  cl_program program;
  cl_kernel kernel;
  char kernelName[98];
} sclSoft;
//
```

3.2 USER FUNCTIONS

3.2.1 Device memory allocation read and write

```
*/

cl_mem sclMalloc(sclHard hardware, cl_int mode, size_t size);
cl_mem sclMallocWrite(sclHard hardware, cl_int mode, size_t size,
                      void *hostPointer);
void sclWrite(sclHard hardware, size_t size, cl_mem buffer, void
    *hostPointer);
void sclRead(sclHard hardware, size_t size, cl_mem buffer, void
    *hostPointer);

/*
```

3.2.2 initialization of sclSoft structs

```
*/

void sclGetCLSoftware(const char *kernel_file, const char *kernel_name,
                      const sclHard hardware, sclSoft *Software);

/*
```

3.2.3 Release and retain OpenCL objects

```
void sclReleaseClSoft(sclSoft soft);
void sclReleaseClHard(sclHard hard);
void sclRetainClHard(sclHard hardware);
void sclReleaseAllHardware(sclHard *hardList, cl_int found);
void sclRetainAllHardware(sclHard *hardList, cl_int found);
void sclReleaseMemObject(cl_mem object);

/*
```

3.2.4 Debug functions

```
void sclPrintErrorFlags(cl_int flag);
void sclPrintHardwareStatus(sclHard hardware);
void sclPrintDeviceNamePlatforms(sclHard *hardList, cl_int found);

/*
```

3.2.5 Device execution

```
cl_event sclLaunchKernel(sclHard hardware, sclSoft software,
                      size_t *global_work_size, size_t
                          *local_work_size);
cl_event sclEnqueueKernel(sclHard hardware, sclSoft software,
                       size_t *global_work_size, size_t
                           *local_work_size);
cl_event sclSetArgsLaunchKernel(sclHard hardware, sclSoft software,
                             size_t *global_work_size,
                             size_t *local_work_size,
                             const char *sizesValues, ...);
cl_event sclSetArgsEnqueueKernel(sclHard hardware, sclSoft software,
                              size_t *global_work_size,
                              size_t *local_work_size,
                              const char *sizesValues, ...);
cl_event sclManageArgsLaunchKernel(sclHard hardware, sclSoft software,
                                size_t *global_work_size,
                                size_t *local_work_size,
                                const char *sizesValues, ...);
/*
```

3.2.6 Event queries

```
*/

cl_ulong sclGetEventTime(sclHard hardware, cl_event event);

/*
```

3.2.7 Queue management

```
*/

cl_int sclFinish(sclHard hardware);

/*
```

3.2.8 Kernel argument setting

```
*/

void sclSetKernelArg(sclSoft software, int argnum, size_t typeSize,
                   void *argument);
void sclSetKernelArgs(sclSoft software, const char *sizesValues, ...);
void _sclVSetKernelArgs(sclSoft software, const char *sizesValues,
                        va_list argList);

/*
```

3.2.9 Hardware init and selection

```
*/

void sclGetHardwareByType(const cl_device_type device_type, const int
    iDevice,
                          int *found, sclHard *hardware);
void sclGetHardware(const int nDevice, int *found, sclHard *GPUHardware);
void sclGetGPUHardware(const int nDevice, int *found, sclHard
    *GPUHardware);
void sclGetCPUHardware(const int nDevice, int *found, sclHard
    *CPUHardware);
void sclGetAcceleratorHardware(const int iDevice, int *found,
                               sclHard *AcceleratorHardware);
// void sclGetAllHardware(int *found, sclHard *hardwareList);
void sclGetFastestDevice(const sclHard *hardList, const cl_int found,
                         sclHard *fastest);

/*
```

3.3 INTERNAL FUNCITONS

3.3.1 debug

```
*/

void _sclWriteArgOnAFile(int argnum, void *arg, size_t size, const char
    *diff);

/*
```

3.3.2 cl software management

```
*/

void _sclBuildProgram(cl_program program, cl_device_id devices,
                      const char *pName);
```

```
cl_kernel _sclCreateKernel(sclSoft software);
cl_program _sclCreateProgram(char *program_source, cl_context context);
char *_sclLoadProgramSource(const char *filename);

/*
```

3.3.3 hardware management

```
*/

int _sclGetMaxComputeUnits(cl_device_id device);
unsigned long int _sclGetMaxMemAllocSize(cl_device_id device);
int _sclGetDeviceType(cl_device_id device);
void _sclSmartCreateContexts(sclHard *hardList, cl_int found);
void _sclCreateQueues(sclHard *hardList, cl_int found);

/*
```

Chapter 4

simpleCl.c API

```
// void sclReleaseAllHardware(sclHard *hardList, cl_int found) {
// //
```

```
// void sclGetAllHardware(int *found, sclHard *hardwareList) {
// //
```

```
void sclGetCLSoftware(const char *kernel_file, const char *kernel_name,
                      const sclHard hardware, sclSoft *Software) {
  //
```

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

4.0.1 kernel argument assignement sizeTypes string's letter meanings

```
  for (p = sizesValues; *p != '\0'; p++) {
    if (*p == '%') {
      switch (*++p) {
      case 'a': /* Single value non pointer argument: byte length, array
          pointer
                  */
        //
```

```
     case 'v': /* Buffer or image object void* argument: array pointer
         */
              //
```

```
     case 'N': /* Local memory object using NULL argument: byte length
         */
              //
```

```
     case 'w': /* output write_only cl_mem buffer: byte length, array
         pointer*/
              //
```

```
     case 'r': /* input read_only cl_mem buffer: byte size, array
         pointer */
              //
```

```
     case 'R': /* output read_write buffer: byte length, array pointer*/
              //
```

```
     case 'g': /* output read_write variable: bytesize */
       //
```