Implementing a Scheduler on GPU using OpenCL

Cassiano Kleinert Casagrande

Dr. B. Mora

Prifysgol Abertawe
Swansea University

Introduction

Graphics Processors can now be used as a computational platform. The parallel nature of modern Graphics Processing Units added to the fact that almost every modern computer has a GPU make them very useful on solving data-parallel problems. GPUs nowadays have more transistors then CPUs and most of those transistors are employed on increasing the number of Single Instruction Multiple Data(SIMD) units, while on CPUs almost two thirds of those transistors are used to implement cache memory and cache logic.

The goal of this project was to implement a scheduler on the GPU memory, each GPU working group will retrieve tasks from the scheduler and will process them. By using this task scheduler, there will be less need to transfer data between the CPU and the GPU, the number of kernel calls and idle working groups will also be lower.

This project used the OpenCL 1.2 framework to implement the scheduler on the GPU memory. The host code was written in C(ANSI C99) and it implemented a library that defines an Abstract Data Type. The library contains functions to create and release a struct called cltask, which holds the information needed by the host in order to use the task pool. There is also code written in the OpenCL language that has functions to manipulate(create, retrieve and finish) tasks in the Graphics Processor memory.

As a proof of concept, the library was used to sort an array of random numbers using the QuickSort algorithm. In this algorithm, each OpenCL working group retrieves the delimiters of a subarray from the scheduler and partitions it around a pivot, creating two new tasks that delimits the new partitions. When there are no tasks left on the task pool, the array is sorted.

General Purpose computing on Graphics Processing Units(GPGPU)

General Purpose computing on Graphics Processing Units(GPGPU) became possible in the early 2000s, when the GPUs hardware started supporting programable shaders in the vertex shader and pixel shader units. Graphics processors could be programmed in order to make changes in the shading on graphic scenes.

At first, the attempts of doing general purpose computations on GPUs used the existing Application Programming Interfaces(API) like OpenGL and DirectX and data should be somehow mapped into graphic primitives in order to be processed. This process of translation was complicated and error-prone.

In the following years, the hardware became flexible and several general purpose programming languages and APIs were created, like Sh, RapidMine and Brook. In 2007 NVIDIA launched the CUDA(Compute Unified Device Architecture), making possible that programmers used the Graphics Processors without the need to do this translation to graphics processing concepts.

OpenCL

In 2008, the OpenCL specification was released, OpenCL is an open framework for developing programs to execute in several platforms like: CPUs, GPUs and FPGAs. using an abstraction to low level hardware routines and using a memory and execution model. It was developed by several technical teams from Intel, IBM, AMD, NVIDIA, Apple and Qualcomm. The framework runs in a CPU, which is called “host”, that is connected to one or more “compute devices”.

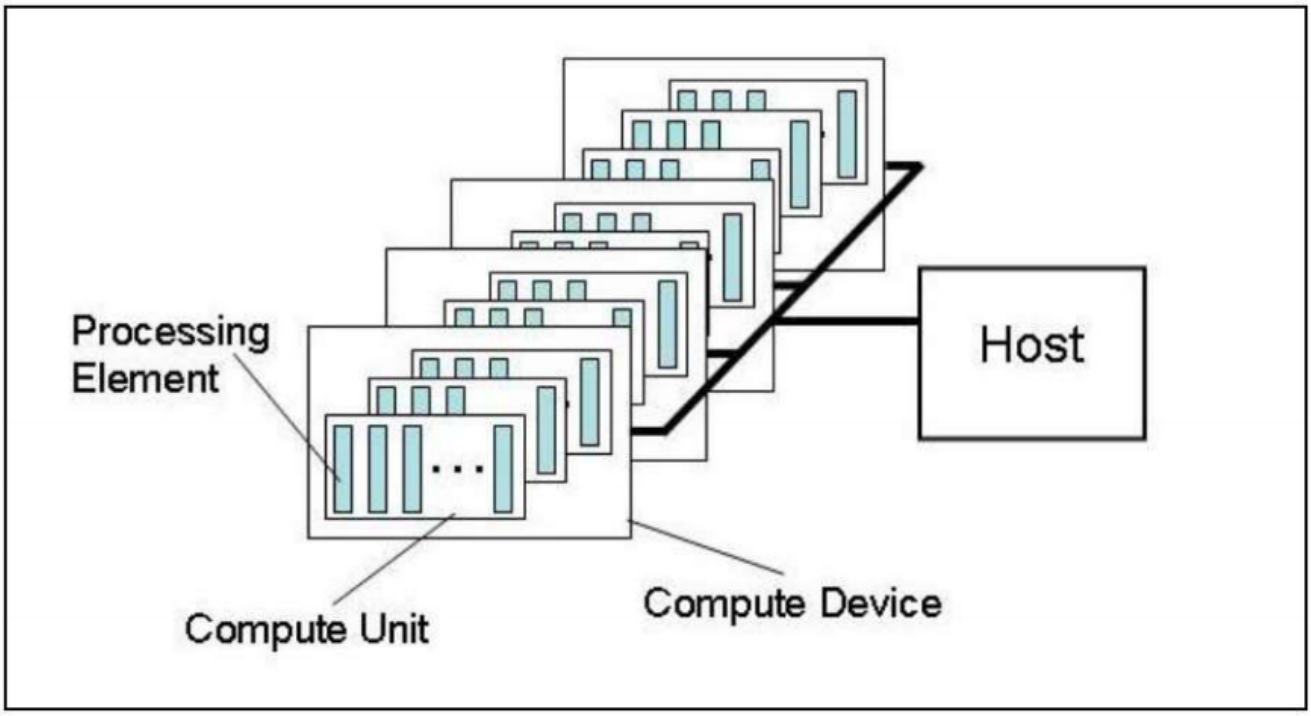

Figure 1: *OpenCL Platform Model (from [Khronos 2011])*

The OpenCL framework has a language similar to C99 that is compiled by the GPU driver compiler and executes in the compute devices. The OpenCL language has a few differences from C99:

1. Removal of recursive functions, variable length arrays and function pointers.

2. Addition of address space qualifiers, used on variable declaration to define which is the memory hierarchy scope of the variable(__global, __local, __constant and __private).

3. Addition of fixed length types like float4(vector of 4 single precision floating points) and their respective vectorized operations.

4. Addition of builtin functions to synchronize, control access to memory regions and control work-groups.

5. Removal of the standard C Library.

```
// This kernel computes FFT of length 1024. The 1024 length FFT is decomposed into
// calls to a radix 16 function, another radix 16 function and then a radix 4 function

__kernel void fft1D_1024 (__global float2 *in, __global float2 *out,
                          __local float *sMemx, __local float *sMemy) {
  int tid = get_local_id(0);
  int blockIdx = get_group_id(0) * 1024 + tid;
  float2 data[16];

  // starting index of data to/from global memory
  in = in + blockIdx;  out = out + blockIdx;

  globalLoads(data, in, 64); // coalesced global reads
  fftRadix16Pass(data);      // in-place radix-16 pass
  twiddleFactorMul(data, tid, 1024, 0);

  // local shuffle using local memory
  localShuffle(data, sMemx, sMemy, tid, (((tid & 15) * 65) + (tid >> 4)));
  fftRadix16Pass(data);               // in-place radix-16 pass
  twiddleFactorMul(data, tid, 64, 4); // twiddle factor multiplication

  localShuffle(data, sMemx, sMemy, tid, (((tid >> 4) * 64) + (tid & 15)));

  // four radix-4 function calls
  fftRadix4Pass(data);      // radix-4 function number 1
  fftRadix4Pass(data + 4);  // radix-4 function number 2
  fftRadix4Pass(data + 8);  // radix-4 function number 3
  fftRadix4Pass(data + 12); // radix-4 function number 4

  // coalesced global writes
  globalStores(data, out, 64);
}
```

Figure 2. Example of Fast Fourier Transform(FFT) on the OpenCL Language.[Apple 2012].

The OpenCL framework also defines one runtime API that is used by the host code. The API has functions to discover and manage the OpenCL platform and its devices. The API also has runtime functions to compile OpenCL Language code,

manage the compute devices, launch kernels on them and synchronize these operations through call queues and asynchronous events[Khronos 2011].

The execution model on OpenCL is an abstraction to the way that low level routines are executed in each device. This makes the OpenCL execution model to scale to different devices like GPUs, CPUs, embedded micro controllers and FPGAs. The OpenCL platform and runtime APIs are used to to select the compute devices and to submit kernel calls to them. Each kernel executes in several work-items grouped into independent work-groups. The results can then be read back by the host.

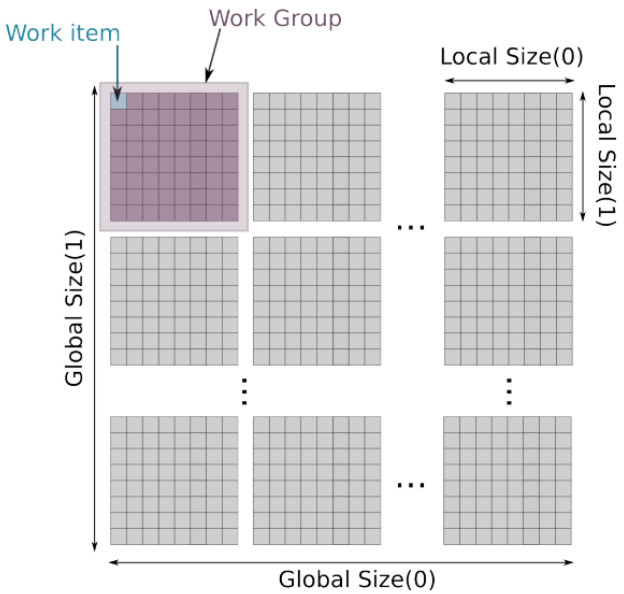

Figure 3. The OpenCL Memory Model(from [Khronos 2011])

Scheduler

Implementing a scheduler on the GPU allow the use of persistent working items executing tasks, instead of the usual way of OpenCL computing: send data to GPU, process it by launching a kernel, wait for it to finish and read it back from the GPU. On the persistent working item approach, the host launches a kernel where the working items continually retrieve tasks from the scheduler and execute them. The

host can then add more tasks to the scheduler on demand or the tasks can create others and wait for then to finish, entering in the dependent state.

The scheduler stores the tasks in an array in the GPU memory. Since several working items can try to access the scheduler at the same time, there is a need to control the access to the scheduler memory. Instead of blocking the whole scheduler each time a working item tries to use the scheduler, each task on the task pool has a mutex that controls its access.

Instead of retrieving the tasks from deterministic places like the beginning or the end of the array, like a stack or queue would, the scheduler tries to retrieve tasks from a random place on the array queue. The insertion of tasks is also made randomically searching for a empty place on the task array to ensure that the idle working groups do not try to retrieve tasks from the same place.

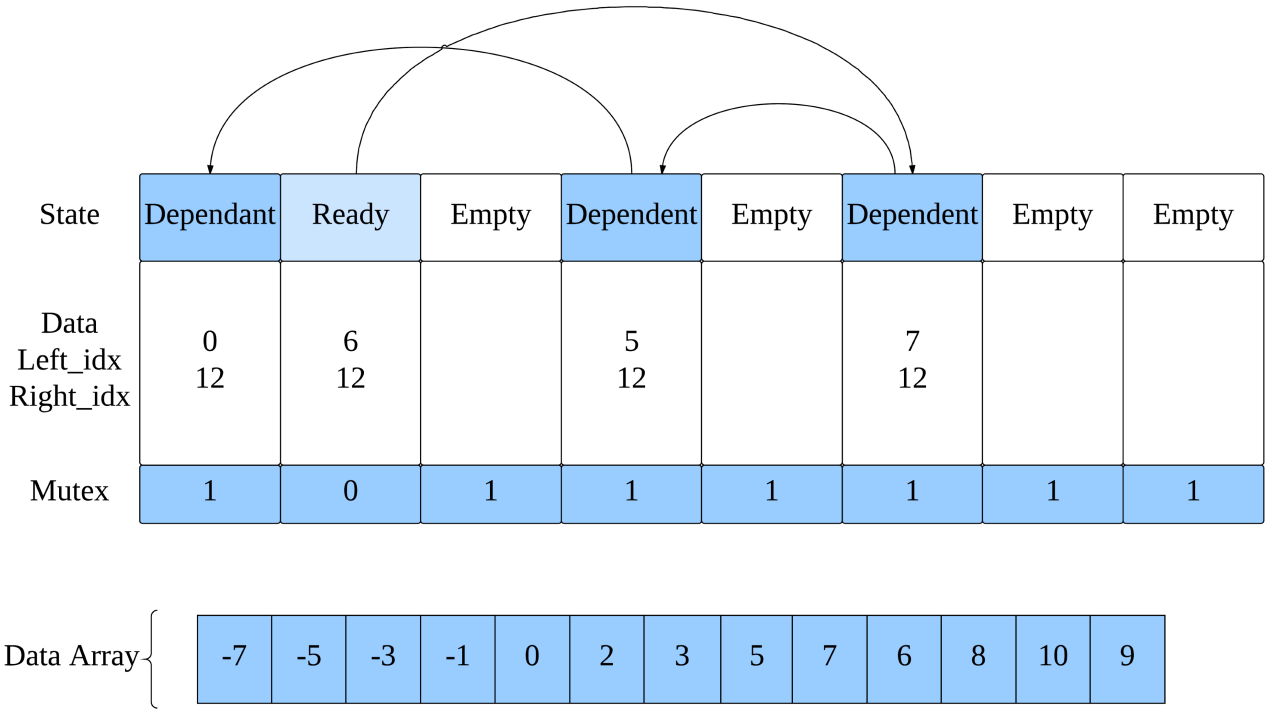

Figure 4. How the task data is stored in the scheduler.

Besides storing the task data in the array, the scheduler also has fields to store, for each task, a mutex to control access to this task, the current state of the task and the index to the position of the parent task in the array.

Sorting the Array

Quicksort is a sorting algorithm that uses the divide-and-conquer approach to sort an array. Good implementations usually have running time of O(n*log(n)) comparisons on the average case.

It was developed in 1960 by Tony Hoare[Hoare 1962] and is today extensively used to order arrays, having faster running times than other sorting algorithms.

To sort an array A[p, r], the algorithm uses the following steps:

Chose a pivot : There are several ways to do that, usually the chosen pivot is the median value of three random array elements. In our implementation the pivot is simply the leftmost element of the array.

Partition the array around the pivot: Arrange the array in a way that A[i]<A[pivot] if i<pivot and A[i]>=A[pivot] if i>pivot.

Sort the partitions: Call recursively the algorithm to sort the sub-arrays created by the partitioning process.

```
int pivot, i, j;
pivot = array[left];
i = left; j = right+1;
while(1)
{
        do ++i; while( array[i] <= pivot && i <= right );
        do --j; while( array[j] > pivot );
        if( i >= j ) break;
                SWAP(array[i], array[j]);
}
```

```
        SWAP(array[left], array[j]);
```

Figure 5. C code to partition the array A between the indexes left and right.

On the GPU implementation, each task stores the index of the leftmost and the rightmost elements of a subarray. Instead of making the recursive calls to the QuickSort method as it would normally happen in a CPU implementation, each recursive call is actually a new task that is going to be created in the task pool.

The sorting method also tests each subarray retrieved from the task pool to see if its size is less then 100, if it is, then instead of making another recursive call, this subarray is sorted using the method QuickSort as described in [Hoare 1962].

```
        While (task pool is not empty)
        {
                Retrieve one task
                Partition the Array
                Create new tasks containing the leftmost and rightmost indexes of each
subarray in the task pool
        }
```

Figure 6. Algorithm of QuickSort interacting with the task pool.

References

[Vinkler] M. Vinkler, J. Bittner, V. Havran, and M. Hapala, "Massively Parallel Hierarchical Scene Processing with Applications in Rendering", presented at Comput. Graph. Forum, 2013, pp.13-25.

[OpenCL] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Jeongdo Son, Satoshi Miki (2012). The OpenCL Programming Book. Japan: Fixstars.

[Khronos 2011] KHRONOS Group. (2011). OpenCL 1.2 Reference Pages. Available at: http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/. Last accessed 31th Aug 2014.

[Boydstun 2011] BOYDSTUN, K., (2011) . Introduction to Opencl (lecture). Avaliable at: http://www.tapir.caltech.edu/~kboyds/ OpenCL/opencl.pdf. Last accessed 3rd Sept 2014.

[Hoare 1962] Hoare, C. A. R. (1962). “Quicksort". Avaliable at: http://comjnl.oxfordjournals.org/content/5/1/10. Last accessed 3rd Sept 2014.

[Apple 2012] Avaliable at: https://developer.apple.com/library/mac/samplecode/OpenCL_FFT/Introduction/Intro.html. Last accessed 3rd Sept 2014.

Appendix

File common.h

```
#ifndef COMMON_H
#define COMMON_H

#define SUCCESS 0
#define FAILURE -1
#define TRUE 1
#define FALSE 0
#define ARRAY_SIZE(a) (sizeof(a) / sizeof(*a))

#ifdef __GNUC__
#define likely(x)       __builtin_expect((x),1)
#define unlikely(x)     __builtin_expect((x),0)
#define prefetch(x)                 __builtin_prefetch(x)
#else
#define likely(x)                   x
#define unlikely(x)                 x
#define prefetch(x)                 do {}while(0);
#endif




#endif
```

File cltask.h

```
#ifndef CLTASK_H
#define CLTASK_H

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

/*
*	This project implements a queue on the GPU memory, each GPU working group will dequeue jobs from
* this queue and will process them.
*/


struct cltask
{
	unsigned long size;
	unsigned long data_size;
	cl_context context;
	cl_command_queue queue;
	cl_mem buffer;
	cl_program program;
	cl_kernel init;
};

struct cltask_gpu_pool
{
	int mutex;
	int state;
	//Missing data field because it is going to be calculated on the runtime
};

struct cltask_gpu_task
{
	unsigned long data_size;
```

```
	struct cltask_gpu_pool tasks;
};

/*
*	Allocates space for the cltask struct, compiles the cltask.cl and creates the kernel.
*	@param context Contains the OpenCL context in which the scheduler will be used.
*	@param queue A command queue in the context.
*	@param device_id The device to create the scheduler.
*	@param size The maximum number of tasks that the scalonator can have(not yet implemented, the maximum
* number of tasks will always be 256).
*	@param data_size The number of bytes that each task can store(not yet implemented, the maximum number
* of bytes will always be 12).
*	@param task Pointer to the first task that will be added to the scheduler.
*	@return Returns a pointer to the cltask structure if successfull, returns NULL otherwise.
*/
struct cltask *cltask_init(cl_context context, cl_command_queue queue, cl_device_id device_id, unsigned long size, unsigned long data_size, void *task);

/*
*	Frees the cltask structure, releasing all resources used by it on the host and ond the device.
*	@param task_pool Pointer to the cltask structure to be released.
*/
void cltask_free(struct cltask *task_pool);

/*
*	Returns the pointer to the buffer containing the scheduler on the device memory.
*	@param task_pool pointer to the cltask structure that contains the buffer.
*	@return Returns a pointer to the cl_mem containing the scheduler on the device memory.
*/
cl_mem *cltask_get_pool(struct cltask *task_pool);


#endif
```

File cltask.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

#include "common.h"
#include "cltask.h"


struct cltask *cltask_init(cl_context context, cl_command_queue queue, cl_device_id
device_id, unsigned long size, unsigned long data_size, void *task)
{
        size_t global_size = 1;
        size_t local_size = 1;
        struct cltask *task_pool;
        FILE *fp;
        size_t source_size;
        char *source_str;

        cl_mem task_buffer;
        cl_int error;
        task_pool = (struct cltask *) malloc(sizeof(struct cltask));
        if (task_pool == NULL)
                return NULL;
        task_pool->size = size;
        task_pool->data_size = data_size;
        task_pool->context = context;
        task_pool->queue = queue;
        task_pool->buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, data_size*(size
+sizeof(struct cltask_gpu_task))+sizeof(struct cltask_gpu_pool), NULL, &error);
        assert(error == CL_SUCCESS);
        fp = fopen("cltask.cl", "r");
        assert(fp != NULL);
```

```
	source_str = (char *) malloc(100000);
	assert(source_str != NULL);
	source_size = fread(source_str, 1, 100000, fp);
	fclose(fp);
	task_pool->program = clCreateProgramWithSource(context, 1, (const char
**)&source_str, (const size_t *) &source_size, &error);
	assert(error == CL_SUCCESS);
	free(source_str);
	error = clBuildProgram(task_pool->program, 1, &device_id, NULL, NULL, NULL);
	if (error != CL_SUCCESS)
	{
		size_t length;
		char buffer[5000];
		error = clGetProgramBuildInfo(task_pool->program, device_id,
CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &length);
		assert(error == CL_SUCCESS);
		printf("Error in compilation cltask.cl\n");
		for (int i=0; i<length; i++)
			putchar(buffer[i]);
		exit(0);
	}
	task_pool->init = clCreateKernel(task_pool->program, "cltask_gpu_init", &error);
	assert(error == CL_SUCCESS);
	task_buffer = clCreateBuffer(task_pool->context, CL_MEM_USE_HOST_PTR,
task_pool->data_size, task, &error);
	assert(error == CL_SUCCESS);
	error = clSetKernelArg(task_pool->init, 0, sizeof(cl_mem), &task_pool->buffer);
	assert(error == CL_SUCCESS);
	error = clSetKernelArg(task_pool->init, 1, sizeof(cl_mem), &task_buffer);
	assert(error == CL_SUCCESS);
	error = clEnqueueNDRangeKernel(task_pool->queue, task_pool->init, 1, NULL,
&global_size, &local_size, 0, NULL, NULL);
	assert (error == CL_SUCCESS);
	error = clFlush(task_pool->queue);
	assert(error == CL_SUCCESS);
	error = clFinish(task_pool->queue);
	assert(error == CL_SUCCESS);
	error = clReleaseMemObject(task_buffer);
	assert(error == CL_SUCCESS);
	return task_pool;
```

```
}

void cltask_free(struct cltask *task_pool)
{
        cl_int error;
        error = clReleaseKernel(task_pool->init);
        assert(error == CL_SUCCESS);
        error = clReleaseProgram(task_pool->program);
        assert(error == CL_SUCCESS);
        error = clReleaseMemObject(task_pool->buffer);
        assert(error == CL_SUCCESS);
        free(task_pool);
}

cl_mem *cltask_get_pool(struct cltask *task_pool)
{
        return &task_pool->buffer;
}
```

File cltask.cl

```
#ifndef CLTASK_CL
#define CLTASK_CL


#define TASK_PROCESSING -2
#define TASK_EMPTY -1
#define TASK_READY 0

#define DATA_SIZE 12
#define SIZE 128

#define LOCK(X) atomic_xchg(&(X), 0)
#define UNLOCK(X) (X = 1)


struct task
{
	/*
	 *	State = -1 Empty state, the task currently does not hold any data.
	 *	State = 0 Ready state, means that the task is ready to be processed.
	 *	State > 0 Dependant state, the task is waiting for -state tasks to be ready.
	 *	State = -2 Processing, the task is being processed.
	 */
	int state;
	int lock;
	int parent_offset;
	unsigned char data[DATA_SIZE];
};

struct pool
{
	int rand_x;
	int rand_y;
	int rand_z;
	int task_number;
	struct task tasks[SIZE];
};
```

```
/*
*	Inits the scheduler data and inserts the first task.
*	@param task_pool The struct to be initialized.
*	@param first_task Pointer to the data of the first task that will be inserted in the
scheduler.
*/
__kernel void cltask_gpu_init(global struct pool *task_pool, global unsigned char *first_task);

/*
*	Inserts a new task in the task pool.
*	@param task_pool The struct that the task will be added.
*	@param task Pointer to the data of the task that will be inserted in the scheduler.
*	@param parent_offset The offset of the parent which is dependant of this newly
inserted task.
*	@return Returns 1 if the insertion was successfull, 0 otherwise.
*/
int cltask_new_task(global struct pool *task_pool, void *task, int parent_offset);

/*
*	Removes a task from the task pool.
*	@param task_pool The struct from which the task will be removed.
*	@param task_offset The offset of the task in the task pool.
*/
void cltask_finish_task(global struct pool *task_pool, int task_offset);

/*
*	Retrieves a task from the task pool to be processed.
*	@param task_pool The scheduler struct.
*	@param task_ret A pointer to a pointer in which will hold the task data.
*	@param offset_ret A pointer to a int which will hold the offset of the task retrieved.
*
*/
int cltask_retrieve_task(global struct pool *task_pool, global void **task_ret, int *offset_ret);

/*
*	Makes a task given by its offset dependant of a number of tasks.
*	@param task_pool The task pool which contain the task.
*	@param offset The offset that identifies the task that will be dependant.
*	@param dependants The number of dependant tasks spawned by the task identified
by offset.
```

```
*/
void cltask_set_dependant(global struct pool *task_pool, int offset, int dependants);

/*
*	A pseudo randomic number generator
*	@param task_pool The task pool struct that contains the seeds for the random number.
*	@return A pseudo randomic integer
*/
int rand(global struct pool *task_pool);

/*
*	Seeds the pseudo randomic generator.
*	@param task_pool The struct which contains the seed.
*	@param seed The seed.
*/
void srand(global struct pool *task_pool, int seed);


__kernel void cltask_gpu_init(global struct pool *task_pool, global unsigned char *first_task)
{
	for (int i=1; i<SIZE; i++)
	{
		task_pool->tasks[i].lock = 1;
		task_pool->tasks[i].state = TASK_EMPTY;
	}
	for (int i=0; i<DATA_SIZE; i++)
		task_pool->tasks[0].data[i] = first_task[i];
	task_pool->tasks[0].lock = 1;
	task_pool->tasks[0].state = TASK_READY;
	task_pool->tasks[0].parent_offset = -1;
	task_pool->task_number=1;
	task_pool->rand_x = 123456789;
	task_pool->rand_y = 678912345;
	task_pool->rand_z = 987651234;
	srand(task_pool, (int) first_task);
}


int cltask_new_task(global struct pool *task_pool, void *task, int parent_offset)
{
```

```
	int offset;
	task_pool->task_number++;
	while (task_pool->task_number < SIZE)
	{
		offset=rand(task_pool)%SIZE;
		if (task_pool->tasks[offset].state==TASK_EMPTY && LOCK(task_pool->tasks[offset].lock)==1)
		{
			for (int i=0; i<DATA_SIZE; i++)
			{
				task_pool->tasks[offset].data[i] = ((unsigned char *) task)[i];
			}
			task_pool->tasks[offset].state = TASK_READY;
			task_pool->tasks[offset].parent_offset = parent_offset;
			UNLOCK(task_pool->tasks[offset].lock);
			return 1;
		}
	}
	return 0;
}

void cltask_finish_task(global struct pool *task_pool, int task_offset)
{
	task_pool->task_number--;
	LOCK(task_pool->tasks[task_offset].lock);
	task_pool->tasks[task_offset].state = TASK_EMPTY;
	if (task_offset != 0)
		task_pool->tasks[task_pool->tasks[task_offset].parent_offset].state--;
	UNLOCK(task_pool->tasks[task_offset].lock);
}

int cltask_retrieve_task(global struct pool *task_pool, global void **task_ret, int *offset_ret)
{
	int offset;
	for (offset = rand(task_pool)%SIZE; task_pool->tasks[0].state != TASK_EMPTY; offset = rand(task_pool)%SIZE)
	{
		if (task_pool->tasks[offset].state==TASK_READY && LOCK(task_pool->tasks[offset].lock)==1)
		{
```

```
                        *offset_ret = offset;
                        task_pool->tasks[offset].state = TASK_PROCESSING;
                        *task_ret = (global void *)task_pool->tasks[offset].data;
                        return 1;
                }
        }
        return 0;
}

void cltask_set_dependant(global struct pool *task_pool, int offset, int dependants)
{
        task_pool->tasks[offset].state = dependants;
        UNLOCK(task_pool->tasks[offset].lock);
        return;
}

void srand(global struct pool *task_pool, int seed)
{
        task_pool->rand_x ^= seed;
        task_pool->rand_y ^= seed<<3;
        task_pool->rand_z ^= seed<<7;
}

int rand(global struct pool *task_pool)
{
        int t;
        task_pool->rand_x ^= task_pool->rand_x<<16;
        task_pool->rand_x ^= task_pool->rand_x>>5;
        task_pool->rand_x ^= task_pool->rand_x<<1;
        t = task_pool->rand_x;
        task_pool->rand_x = task_pool->rand_y;
        task_pool->rand_y = task_pool->rand_z;
        task_pool->rand_z = t^task_pool->rand_x^task_pool->rand_y;
        return task_pool->rand_z;
}


#endif
```

File main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <assert.h>

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

#include "common.h"
#include "cltask.h"

#define SIZE 2950

int main(void)
{
        int task[] = {0, SIZE-1, 0};
        int values[SIZE];
        cl_context context;
        cl_device_id device_id;
        cl_platform_id platform_id;
        cl_command_queue queue;
        cl_program program;
        cl_kernel run;
        cl_mem array;
        cl_int error;
        cl_uint ret_num_devices, ret_num_platforms;
        size_t global = 32;
        size_t local = 4;
        struct cltask *task_pool;
        FILE *fp;
        size_t source_size;
        char *source_str;
        srand(time(NULL));
        error = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
        assert(error == CL_SUCCESS);
```

```
	error = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
&ret_num_devices);
	assert(error == CL_SUCCESS);
	context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &error);
	assert(error == CL_SUCCESS);
	queue = clCreateCommandQueue(context, device_id, 0, &error);
	assert(error == CL_SUCCESS);
	task_pool = cltask_init(context, queue, device_id, 256, 12, &task);
	assert(task_pool != NULL);
	for (int i=0; i<SIZE; i++)
	{
		values[i] = rand();
	}
	array = clCreateBuffer(context, CL_MEM_USE_HOST_PTR, SIZE*sizeof(int), values,
&error);
	assert(error==CL_SUCCESS);
	fp = fopen("main.cl", "r");
	assert(fp != NULL);
	source_str = (char *) malloc(100000);
	source_size = fread(source_str, 1, 100000, fp);
	fclose(fp);
	program = clCreateProgramWithSource(context, 1, (const char **) &source_str, (const
size_t *) &source_size, &error);
	assert(error == CL_SUCCESS);
	free(source_str);
	error = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
	if (error != CL_SUCCESS)
	{
		size_t length;
		char buffer[5000];
		clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
sizeof(buffer), buffer, &length);
		printf("Error in compilation main.cl\n");
		printf("%s\n", buffer);
	}
	run = clCreateKernel(program, "run", &error);
	assert(error == CL_SUCCESS);
	error = clSetKernelArg(run, 0, sizeof(cl_mem), (cltask_get_pool(task_pool)));
	assert(error == CL_SUCCESS);
	error = clSetKernelArg(run, 1, sizeof(cl_mem), &array);
```

```
        assert(error == CL_SUCCESS);
        error = clEnqueueNDRangeKernel(queue, run, 1, NULL, &global, &local, 0, NULL,
NULL);
        assert (error == CL_SUCCESS);
        error = clEnqueueReadBuffer(queue, array, 0, 0, SIZE*sizeof(int), values, 0, NULL,
NULL);
        assert(error == CL_SUCCESS);
        error = clFlush(queue);
        assert(error == CL_SUCCESS);
        error = clFinish(queue);
        assert(error == CL_SUCCESS);
        cltask_free(task_pool);
        error = clReleaseMemObject(array);
        assert(error == CL_SUCCESS);
        error = clReleaseProgram(program);
        assert(error == CL_SUCCESS);
        error = clReleaseKernel(run);
        assert(error == CL_SUCCESS);
        error = clReleaseCommandQueue(queue);
        assert(error == CL_SUCCESS);
        error = clReleaseContext(context);
        return SUCCESS;
}
```

File main.cl

```
#include "cltask.cl"
```

```
#define SWAP(X, Y) \
{                  \
        int tmp = (X); \
        X = (Y);       \
        Y = tmp;       \
}



__kernel void run(global struct pool *pool, global int *array)
{
        global int *task;
        int current_offset, array_offset, left, right, type;
        // Retrieves a new task from the task pool
        while (cltask_retrieve_task(pool, &task, &current_offset) != 0)
        {
                left = task[0];
                right = task[1];
                type = task[2];
                if (type == 0)
                {
                        //If the sub-array is large enough, subdivide it.
                        if (right-left > 300)
                        {
                                int new_task[3];
                                int pivot, i, j;
                                pivot = array[left];
                                i = left; j = right+1;
                                //Partition the array
                                while(1)
                                {
                                        do ++i; while( array[i] <= pivot && i <= right );
                                        do --j; while( array[j] > pivot );
                                        if( i >= j ) break;
                                        SWAP(array[i], array[j]);
                                }
                                SWAP(array[left], array[j]);
```

```
                        cltask_set_dependant(pool, current_offset, 2);
                        new_task[0] = left;
                        new_task[1] = j-1;
                        new_task[2] = 0;
                        //Insert first partition in the task pool
                        cltask_new_task(pool, new_task, current_offset);
                        new_task[0] = j+1;
                        new_task[1] = right;
                        new_task[2] = 0;
                        //Insert second partition in the task pool.
                        cltask_new_task(pool, new_task, current_offset);
                        task[2] = 1;
                }
                else
                //If it is a small partition, order it using insertion sort
                {
                        for (int i=1; i<right-left+1; i++)
                        {
                                int j = i;
                                while (j > 0 && array[left+j]<array[left+j-1])
                                {
                                        SWAP(array[left+j-1], array[left+j]);
                                        j--;
                                }
                        }
                }
        }
        if (type == 1)
        {
                cltask_finish_task(pool, current_offset);
        }
    }
}
```