Heterogeneous Computing using openCL lecture 2

F21DP Distributed and Parallel Technology

Sven-Bodo Scholz
Example: Squares

result[i] = data[i] * data[i]
Selecting a Platform

- Each OpenCL implementation (i.e. an OpenCL library from AMD, NVIDIA, etc.) defines *platforms* which enable the host system to interact with OpenCL-capable devices
  - Currently each vendor supplies only a single platform per implementation
Selecting Devices

• Once a platform is selected, we can then query for the devices that it knows how to interact with

```c
clGetDeviceIDs(platform, device_type, num_entries, devices, num_devices)
```

• We can specify which types of devices we are interested in (e.g. all devices, CPUs only, GPUs only)

• This call is performed twice as with `clGetPlatformIDs`
  – The first call is to determine the number of devices, the second retrieves the device objects
Contexts

• Is an abstract execution environment

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

void *user_data,
cl_int *errcode_ret)
```
Command Queues

- Command queues associate a context with a device

```c
cl_command_queue clCreateCommandQueue (cl_context context,
                                     cl_device_id device,
                                     cl_command_queue_properties properties,
                                     cl_int *errcode_ret)
```
Lucky You !😊

/****************************************************************************
* initGPU : sets up the openCL environment for using a GPU.
* Note that the system may have more than one GPU in which case
* the one that has been pre-configured will be chosen.
* If anything goes wrong in the course, error messages will be
* printed to stderr and the last error encountered will be returned.
* 
****************************************************************************/
extern cl_int initGPU ();

Choses **platform** and **device** and creates
**a context** and a **command queue** for you 😊
Memory Objects

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

Uninitialized OpenCL memory objects—the original
data will be transferred later to/from these objects

Original input/output data
(not OpenCL memory objects)
Transferring Data

The images are written to a device. The images are redundant here to show that they are both part of the context (on the host) and physically on the device.

```c
cl_int clEnqueueWriteBuffer(cl_command_queue command_queue,  
cl_mem buffer,  
cl_bool blocking_write,  
size_t offset,  
size_t cb,  
const void *ptr,  
cl_uint num_events_in_wait_list,  
const cl_event *event_wait_list,  
cl_event *event)
```
Programs

\[ \text{cl\_program clCreateProgramWithSource (cl\_context context, cl\_uint count, const char **strings, const size_t *lengths, cl\_int *errcode\_ret)} \]
Compiling Programs

- This function compiles and links an executable from the program object for each device in the context
  - If `device_list` is supplied, then only those devices are targeted
- Optional preprocessor, optimization, and other options can be supplied by the `options` argument
Kernels

\[
\text{cl\_kernel} \quad \text{clCreateKernel (cl\_program program, const char \*kernel\_name, cl\_int \*errcode\_ret)}
\]

Conte xt
Runtime Compilation

- There is a high overhead for compiling programs and creating kernels
  - Each operation only has to be performed once (at the beginning of the program)
    - The kernel objects can be reused any number of times by setting different arguments
Kernel Arguments

Data (e.g. images) are set as kernel arguments

```
cl_int clSetKernelArg(cl_kernel kernel, 
                      cl_uint arg_index, 
                      size_t arg_size, 
                      const void *arg_value)
```
/*******************************************************************************
* setupKernel : this routine prepares a kernel for execution. It takes the
* following arguments:
* - the kernel source as a string
* - the name of the kernel function as string
* - the number of arguments (must match those specified in the
  kernel source!)
* - followed by the actual arguments. Each argument to the kernel
  results in two or three arguments to this function, depending
  on whether these are pointers to float-arrays or integer values:
*
* legal argument sets are:
* FloatArr::clarg_type, num_elems::int, pointer::float *, and
* IntConst::clarg_type, number::int
*
typedef enum {
  FloatArr,
  IntConst
} clarg_type;
Lucky you II cont.

* If anything goes wrong in the course, error messages will be printed to stderr. The pointer to the fully prepared kernel will be returned.

* Note that this function actually performs quite a few openCL tasks. It compiles the source, it allocates memory on the device and it copies over all float arrays. If a more sophisticated behaviour is needed you may have to fall back to using openCL directly.

****************************************************************************/

extern cl_kernel setupKernel( const char *kernel_source, char *kernel_name, int num_args, ...);

count = 1024;
kernel = setupKernel( KernelSource, "square", 3, FloatArr, count, data, FloatArr, count, results, IntConst, count);
const char *KernelSource = "\n" "__kernel void square(\n" "__global float* input,\n" "__global float* output,\n" const unsigned int count)\n{"\n" int i = get_global_id(0);\n" output[i] = input[i] * input[i];\n"}"

data = (float *) malloc (count * sizeof (float));
results = (float *) malloc (count * sizeof (float));

for (int i = 0; i < count; i++)
  data[i] = rand() / (float) RAND_MAX;

kernel = setupKernel( KernelSource, "square", 3, FloatArr, count, data,
  FloatArr, count, results,
  IntConst, count);
Executing the Kernel

An index space of threads is created (dimensions match the data)

```c
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)
```
Executing the Kernel

- A thread structure defined by the index-space that is created
  - Each thread executes the same kernel on different data

SIMT = Single Instruction Multiple Threads
Copying Data Back

```c
cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
cl_mem buffer,
cl_bool blocking_read,
size_t offset,
size_t cb,
void *ptr,
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)
```
Lucky you III 😊

/*******************************************************************************
 * runKernel :
 * this routine executes the kernel given as first argument.
 * The thread-space is defined through the next two arguments:
 * <dim> identifies the dimensionality of the thread-space and
 * <globals> is a vector of length <dim> that gives the upper
 * bounds for all axes. The argument <local> specifies the size
 * of the individual warps which need to have the same dimensionality
 * as the overall range.
 * If anything goes wrong in the course, error messages will be
 * printed to stderr and the last error encountered will be returned.
 *
 * Note that this function not only executes the kernel with the given
 * range and warp-size, it also copies back *all* arguments from the
 * kernel after the kernel's completion. If a more sophisticated
 * behaviour is needed you may have to fall back to using openCL directly.
 *
*******************************************************************************/

extern cl_int runKernel( cl_kernel kernel, int dim, size_t *global, size_t *local);

size_t global[1] = {1024};
size_t local[1] = {32};
runKernel( kernel, 1, global, local);
Finally: Release the Resources

extern cl_int freeDevice();
OpenCL Timing

- OpenCL provides “events” which can be used for timing kernels
  - Events will be discussed in detail in Lecture 11
- We pass an event to the OpenCL enqueue kernel function to capture timestamps
- Code snippet provided can be used to time a kernel
  - Add profiling enable flag to create command queue
  - By taking differences of the start and end timestamps we discount overheads like time spent in the command queue

```c
cl_event event_timer;
clEnqueueNDRangeKernel(
    myqueue, myKernel,
    2, 0, globalws, localws,
    0, NULL, &event_timer);

unsigned long starttime, endtime;

clGetEventProfilingInfo( event_time,
    CL_PROFILING_COMMAND_START,
    sizeof(cl_ulong), &starttime, NULL);

clGetEventProfilingInfo( event_time,
    CL_PROFILING_COMMAND_END,
    sizeof(cl_ulong), &endtime, NULL);

unsigned long elapsed =
    (unsigned long)(endtime - starttime);
```