高性能计算系统II(B)

基于图形处理器的并行计算及CUDA编程

Ying Liu, Associate Prof., Ph.D

School of Computer and Control, University of Chinese Academy of Sciences

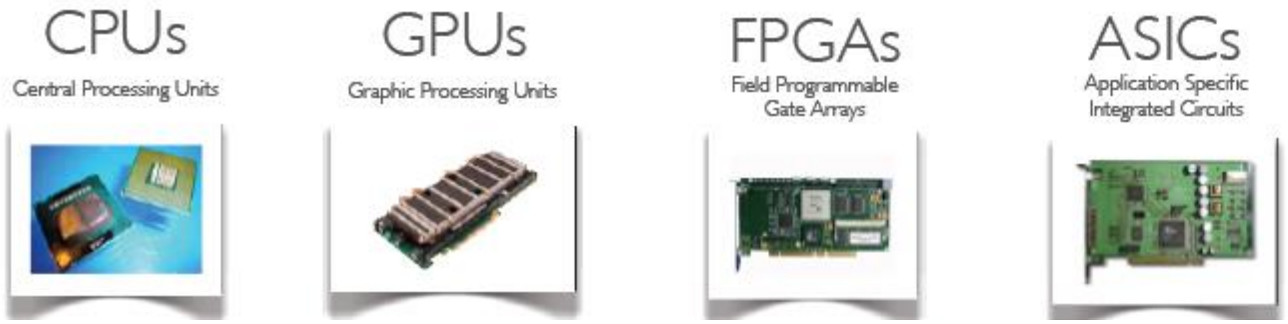
Key Lab of Big Data Mining & Knowledge Management, Chinese Academy of Sciences

Programming the GPU using OpenCL

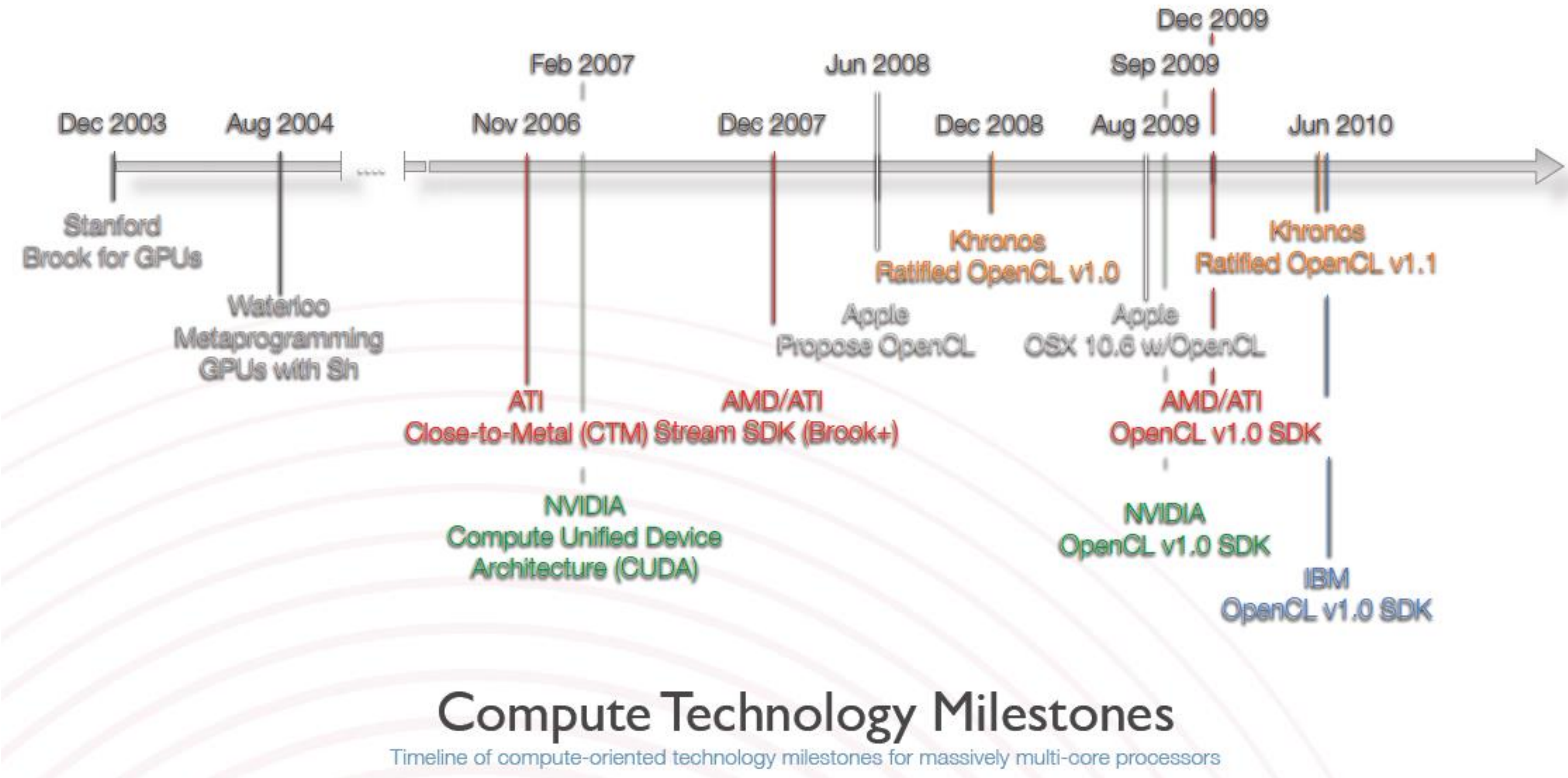
- The 1st open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices
- Support a wide range of applications, from embedded and consumer software to HPC solutions, through a low-level, high-performance, portable abstraction
- Form the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications

OpenCL

- Open standard for parallel programming on heterogeneous systems
  - CPU, GPU, other accelerators



- Easy to use: C code + APIs
- Portable: compiles automatically to the platform

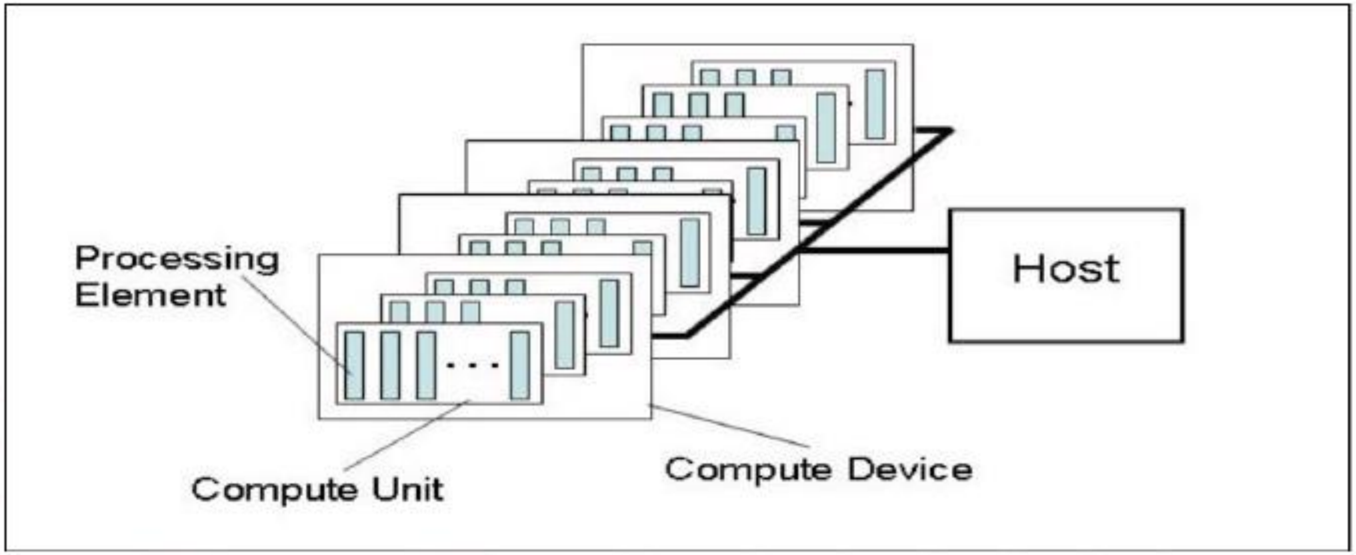


Dec 2009
Feb 2007
Jun 2008
Sep 2009
Dec 2003
Aug 2004
Nov 2006
Dec 2007
Dec 2008
Aug 2009
Jun 2010
Stanford
Brook for GPUs
Waterloo
Metaprogramming
GPUs with Sh
ATI
Close-to-Metal (CTM)
NVIDIA
Compute Unified Device
Architecture (CUDA)
AMD/ATI
Stream SDK (Brook+)
Apple
Propose OpenCL
Khronos
Ratified OpenCL v1.0
Apple
OSX 10.6 w/OpenCL
NVIDIA
OpenCL v1.0 SDK
AMD/ATI
OpenCL v1.0 SDK
Khronos
Ratified OpenCL v1.1
IBM
OpenCL v1.0 SDK
Compute Technology Milestones
Timeline of compute-oriented technology milestones for massively multi-core processors

OpenCL

- OpenCL programs are divided in two parts
  - Host code
  - Device code (kernel code)
    - SIMT: the same code is executed in parallel by a different thread, each thread executes the code with different data
    - Execution model: work-item, work-group, ND-Range

OpenCL Platform Model



- One Host + one or more Compute Devices
  - Each Compute Device is composed of one or more Compute Units

OpenCL Program Structure

- 'host' code:
  - C/C++ code that will run on the CPU - Compiled using standard compilers + OpenCL headers
  - Uses OpenCL APIs to:
    - Move data from system memory to device DRAM
    - Start multiple instances of the kernel to run on the device
    - Each instance acts on a portion of the data
    - Copy back results
- 'kernel' code
  - C code that will run on the device – Compiled using the vendor‘s compiler (e.g. NVIDIA's compiler)
  - Operates on data stored in the device DRAM
  - Writes results in device DRAM

Configuration

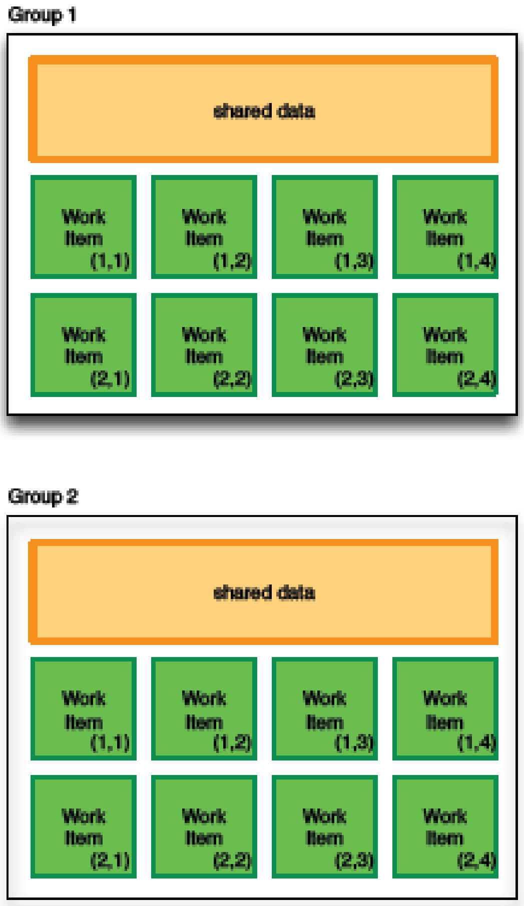
- Work-item
  - Equivalent to CUDA threads, the smallest execution entity
  - Lots of work-items (specified by the programmer) are launched, each one executing the same code
  - Each work-item has an ID
  - Can have local (private) data that belong to that thread only

```
int a[N], b[N], c[N];
int tid;

tid = getThreadID();
c[tid] = a[tid] + b[tid];
```

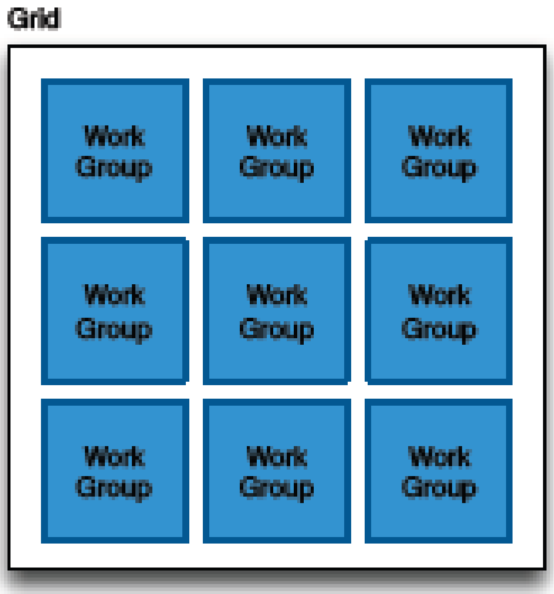
Configuration

- Work-group
  - Groups are collections of Work Items
  - Equivalent to CUDA thread blocks
  - Each work-group has an unique ID
  - Items inside a group are executed in parallel
  - Items inside a group can share local data
  - Items can be organized as 1D, 2D or 3D arrays

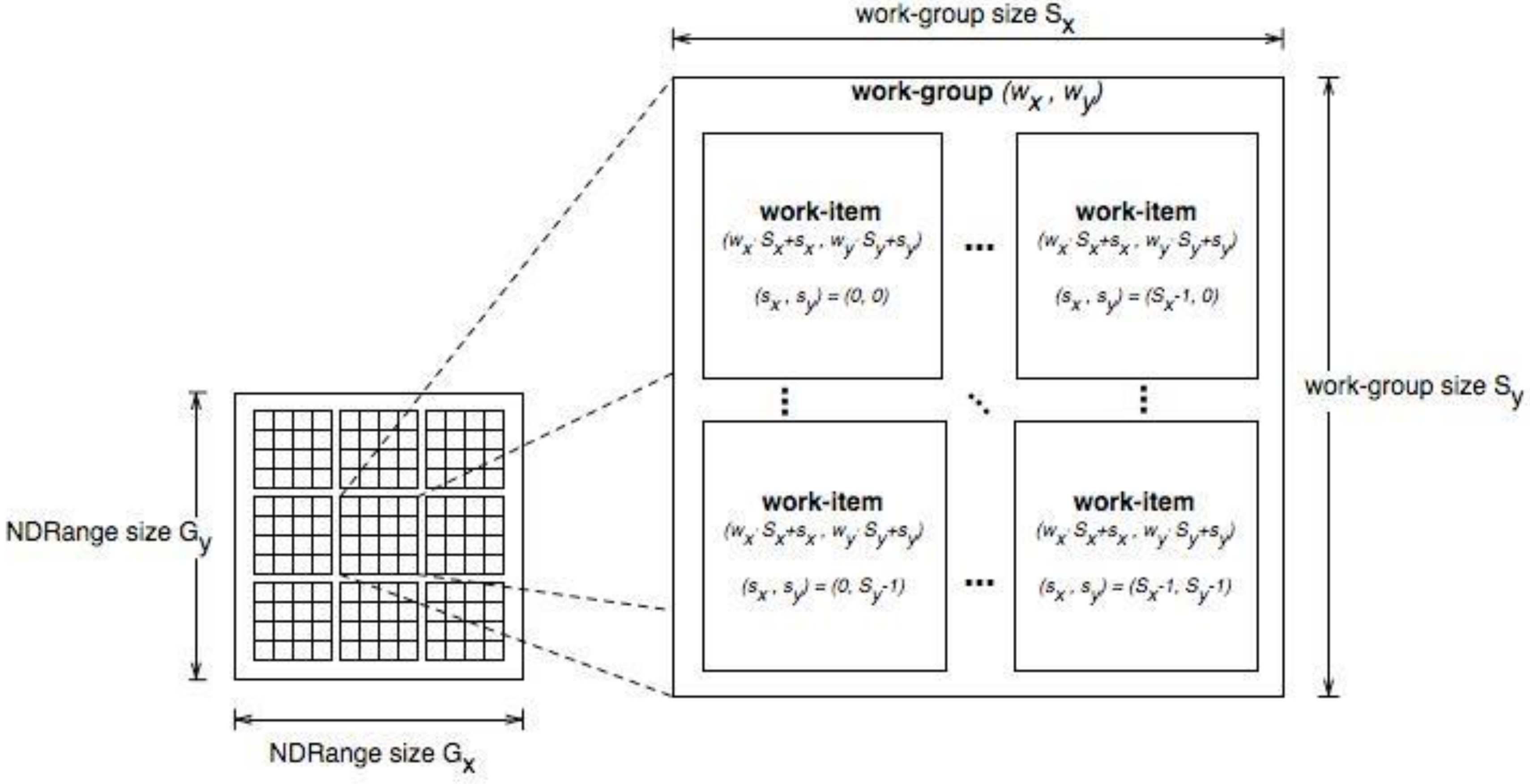


Configuration

- ND-Range
  - Work Groups are organized in a grid (1D, 2D or 3D) as part of the Kernel definition
  - No communications between groups
  - No synchronization between groups

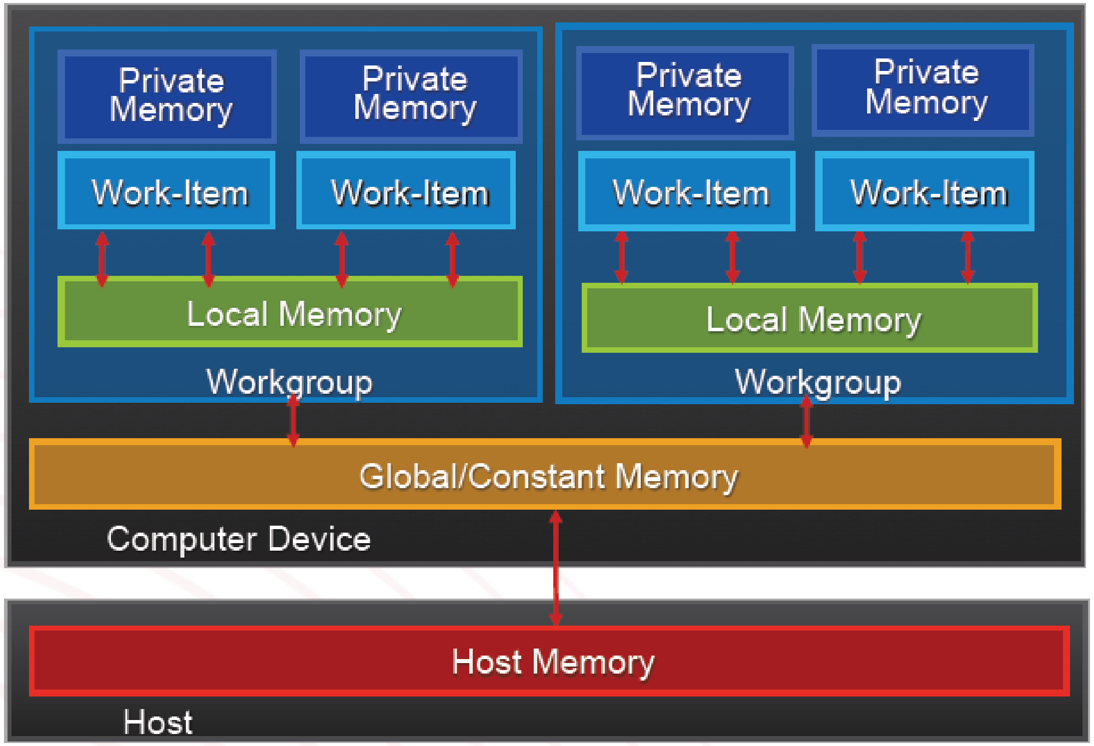


Configuration



OpenCL Memory Hierarchy

- Private Memory
  - Per work-item
- Local Memory
  - Shared within a workgroup
- Global/Constant Memory
  - Visible to all workgroups
- Host Memory
  - On the CPU



✓ *Memory management is explicit*

✓ *Move data from host -> global -> local and back*

Kernel Code

- Adding two vectors on CPU

```
void vector_add_cpu (const float* src_a,
                     const float* src_b,
                     float*  res,
                     const int num)
{
   for (int i = 0; i < num; i++)
      res[i] = src_a[i] + src_b[i];
}
```

Kernel Code

- Adding two vectors

```
__kernel void vector_add_gpu (__global const float* src_a,
                    __global const float* src_b,
                    __global float* res,
                    const int num)
{
    const int idx = get_global_id(0);

    if (idx < num)
        res[idx] = src_a[idx] + src_b[idx];
}
```

Host Code

- Creating the basic OpenCL run-time environment
  - Platform: The host plus a collection of devices managed by the OpenCL framework that allow an application to share resources and execute kernels on devices in the platform

//returns the error code

cl_int oclGetPlatformID (cl_platform_id *platforms)   // Pointer to the platform object

Host Code

- Creating the basic OpenCL run-time environment
  - Device: are represented by cl_device objects

```
//returns the error code
cl_int clGetDeviceIDs  (cl_platform_id platform,
        cl_device_type device_type,   // Bitfield identifying the type for
    the GPU we use CL_DEVICE_TYPE_GPU
        cl_uint num_entries,   // Number of devices, typically 1
        cl_device_id *devices,   // Pointer to the device object
        cl_uint *num_devices)   // Puts here the number of devices
    matching the device_type
```

Host Code

- Creating the basic OpenCL run-time environment
  - Context: defines the entire OpenCL environment, including OpenCL kernels, devices, memory management, command-queues, etc.

// Returns the context

cl_context clCreateContext (const cl_context_properties *properties,
    // Bitwise with the properties (see specification)

            cl_uint num_devices,   // Number of devices

            const cl_device_id *devices,   // Pointer to the devices
    object

            void (*pfn_notify)(const char *errinfo, const void
    *private_info, size_t cb, void *user_data),   // (don't worry about
    this)

            void *user_data,   // (don't worry about this)

            cl_int *errcode_ret)   // error code result

Host Code

- Creating the basic OpenCL run-time environment
  - Command-Queue: an object where OpenCL commands are enqueued to be executed by the devices

```
cl_command_queue clCreateCommandQueue
    (           cl_context context,
                cl_device_id device,
                cl_command_queue_properties properties,   // Bitwise with the properties
                cl_int *errcode_ret)   // error code result
```

Example

```
cl_int error = 0;   // Used to handle error codes
cl_platform_id platform;
cl_context context;
cl_command_queue queue;
cl_device_id device;

// Platform
error = oclGetPlatformID(&platform);
if (error != CL_SUCCESS) {
   cout << "Error getting platform id: " << errorMessage(error) << endl;
   exit(error);
}
// Device
error = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
if (err != CL_SUCCESS) {
   cout << "Error getting device ids: " << errorMessage(error) << endl;
   exit(error);
}
```

Example

```
// Context
context = clCreateContext(0, 1, &device, NULL, NULL, &error);
if (error != CL_SUCCESS) {
   cout << "Error creating context: " << errorMessage(error) << endl;
   exit(error);
}
// Command-queue
queue = clCreateCommandQueue(context, device, 0, &error);
if (error != CL_SUCCESS) {
   cout << "Error creating command queue: " << errorMessage(error) << endl;
   exit(error);
}
```

Example

- Allocating Memory

```
const int size = 1234567
float* src_a_h = new float[size];
float* src_b_h = new float[size];
float* res_h = new float[size];

// Initialize both vectors
for (int i = 0; i < size; i++) {
   src_a_h = src_b_h = (float) i;
}
```

Example

- Allocating Memory

```
// Returns the cl_mem object referencing the memory allocated on
    the device

cl_mem clCreateBuffer (cl_context context,   // The context where
    the memory will be allocated
        cl_mem_flags flags,
        size_t size,   // size in bytes
        void *host_ptr,
        cl_int *errcode_ret)
```

Example

- *flags* is bitwise and the options are:

  CL_MEM_READ_WRITE

  CL_MEM_WRITE_ONLY

  CL_MEM_READ_ONLY

  CL_MEM_USE_HOST_PTR

  CL_MEM_ALLOC_HOST_PTR

  CL_MEM_COPY_HOST_PTR - copies the memory pointed by *host_ptr*

Example

const int mem_size = sizeof(float)*size;

// Allocates a buffer of size mem_size and copies mem_size bytes from src_a_h

cl_mem src_a_d = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, mem_size, src_a_h, &error);

cl_mem src_b_d = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, mem_size, src_b_h, &error);

cl_mem res_d = clCreateBuffer(context, CL_MEM_WRITE_ONLY, mem_size, NULL, &error);

Create a Program

// Returns the OpenCL program

```
cl_program clCreateProgramWithSource (cl_context context,
            cl_uint count, // number of files
            const char **strings, // array of strings, each one is a file
            const size_t *lengths, // array specifying the file lengths
            cl_int *errcode_ret) // error code to be returned
```

Compile a Program

cl_int clBuildProgram (cl_program program,

cl_uint num_devices,

const cl_device_id *device_list,

const char *options, // Compiler options, see the specifications for more details

void (*pfn_notify)(cl_program, void *user_data),

void *user_data)

View a Compile Log

cl_int clGetProgramBuildInfo (cl_program program,

cl_device_id device,

cl_program_build_info param_name, // The parameter we want to know

size_t param_value_size,

void *param_value, // The answer

size_t *param_value_size_ret)

Example

// Creates the program

// Uses NVIDIA helper functions to get the code string and it's size (in bytes)

size_t src_size = 0;

const char* path = shrFindFilePath("vector_add_gpu.cl", NULL);

const char* source = oclLoadProgSource(path, "", &src_size);

cl_program program = clCreateProgramWithSource(context, 1, &source, &src_size, &error);

assert(error == CL_SUCCESS);

Example

```
// Builds the program
error = clBuildProgram(program, 1, &device, NULL, NULL,
NULL);
assert(error == CL_SUCCESS);
// Shows the log
char* build_log;
size_t log_size;
// First call to know the proper size
clGetProgramBuildInfo(program, device,
CL_PROGRAM_BUILD_LOG, 0, NULL, &log_size);
build_log = new char[log_size+1];
```

Example

```
// Second call to get the log
clGetProgramBuildInfo(program, device,
CL_PROGRAM_BUILD_LOG, log_size, build_log, NULL);
build_log[log_size] = '\0';
cout << build_log << endl;
delete[] build_log;


// Extracting the kernel
cl_kernel vector_add_kernel = clCreateKernel(program,
"vector_add_gpu", &error);
assert(error == CL_SUCCESS);
```

Launching the Kernel

cl_int clSetKernelArg (cl_kernel kernel, // Which kernel

cl_uint arg_index, // Which argument

size_t arg_size, // Size of the next argument (not of the value pointed by it!)

const void *arg_value) // Value

Launching the Kernel

cl_int clEnqueueNDRangeKernel

(cl_command_queue command_queue,

cl_kernel kernel,

cl_uint work_dim, // Choose if we are using 1D, 2D or 3D work-items and work-groups

const size_t *global_work_offset,

const size_t *global_work_size, // The total number of work-items (must have work_dim dimensions)

const size_t *local_work_size, // The number of work-items per work-group (must have work_dim dimensions)

cl_uint num_events_in_wait_list,

const cl_event *event_wait_list, cl_event *event)

Example

// Enqueuing parameters

// Note that we inform the size of the cl_mem object, not the size of the memory pointed by it

```
error = clSetKernelArg(vector_add_k, 0, sizeof(cl_mem), &src_a_d);
error |= clSetKernelArg(vector_add_k, 1, sizeof(cl_mem), &src_b_d);
error |= clSetKernelArg(vector_add_k, 2, sizeof(cl_mem), &res_d);
error |= clSetKernelArg(vector_add_k, 3, sizeof(size_t), &size);
assert(error == CL_SUCCESS);
// Launching kernel
const size_t local_ws = 512;     // Number of work-items per work-group
// shrRoundUp returns the smallest multiple of local_ws larger than size
const size_t global_ws = shrRoundUp(local_ws, size);   // Total number of work-items
error = clEnqueueNDRangeKernel(queue, vector_add_k, 1, NULL, &global_ws, &local_ws, 0, NULL, NULL);
assert(error == CL_SUCCESS);
```

Reading Back

```
cl_int  clEnqueueReadBuffer (cl_command_queue command_queue,
                    cl_mem buffer,   // from which buffer
                    cl_bool blocking_read,   // whether is a blocking or non-blocking read
                    size_t offset,   // offset from the beginning
                    size_t cb,   // size to be read (in bytes)
                    void *ptr,   // pointer to the host memory
                    cl_uint num_events_in_wait_list,
                    const cl_event *event_wait_list,
                    cl_event *event)
```

Example

```
// Reading back
float* check = new float[size];
clEnqueueReadBuffer(queue, res_d, CL_TRUE, 0, mem_size, check, 0,
NULL, NULL);
```

Cleaning Up

```
// Cleaning up
delete[] src_a_h;
delete[] src_b_h;
delete[] res_h;
delete[] check;
clReleaseKernel(vector_add_k);
clReleaseCommandQueue(queue);
clReleaseContext(context);
clReleaseMemObject(src_a_d);
clReleaseMemObject(src_b_d);
clReleaseMemObject(res_d);
```