Intro to OpenCL

BITES AR-GE

Contents

OpenCL

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms. OpenCL is a framework for parallel programming and includes a language, API, libraries and a runtime system to support software development.

Specification

Installation

OpenCL comes as a runtime environment and has to be installed on your target machine, no matter if you are using Windows or Linux. For Mac OS X, OpenCL is already part of the system, so there is nothing to install there.

[Intel, Drivers and Runtimes]

[AMD, Tools & SDKs]

[NVIDIA, GPU Computing SDK for]

Basic Concepts

Host:

In developing an OpenCL project, the first step is to code the host application. This runs on a user's computer (the host) and dispatches kernels to connected devices. The host application can be coded in C or C++, and every host application requires five data structures: cl_device_id, cl_kernel, cl_program, cl_command_queue, and cl_context.

Basic Concepts

The host creates a command queue for each device and enqueues commands. One type of command tells a device to execute a kernel.

Host selects which devices should be placed in a context.

Host can dispatch the same kernel to multiple devices through their command queues.

The host identifies the number of work items that should be generated to execute the kernel.

Basic Concepts

Device:

A device receives kernels from the host. In code, a device is represented by a cl_device_id.

Kernel:

A function that is executed on the device. A host application distributes kernels to devices. In code, a kernel is represented by a cl_kernel.

Basic Concepts

Program:

The host selects kernels from a program. In code, a program is represented by a cl_program.

Command queue:

An object that holds commands that will be executed on a specific device. The command-queue is created on a specific device in a context. In code, a command queue is represented by a cl_command_queue.

Basic Concepts

Context:

The environment within which the kernels execute and the domain in which synchronization and memory management is defined. In code, a context is represented by a cl_context.

Buffer Object:

A memory object that stores a linear collection of bytes. Buffer objects are accessible using a pointer in a kernel executing on a device. Buffer objects can be manipulated by the host using OpenCL API calls.

Basic Concepts

Work Item:

The smallest execution entity.

Each work-item has two IDs: a global ID and a local ID. The global ID identifies the work-item among all other work-items executing the kernel. The local ID identifies the work-item among other work-items in the work-group. Work-items in different work-groups may have the same local ID, but they'll never have the same global ID.

Basic Concepts

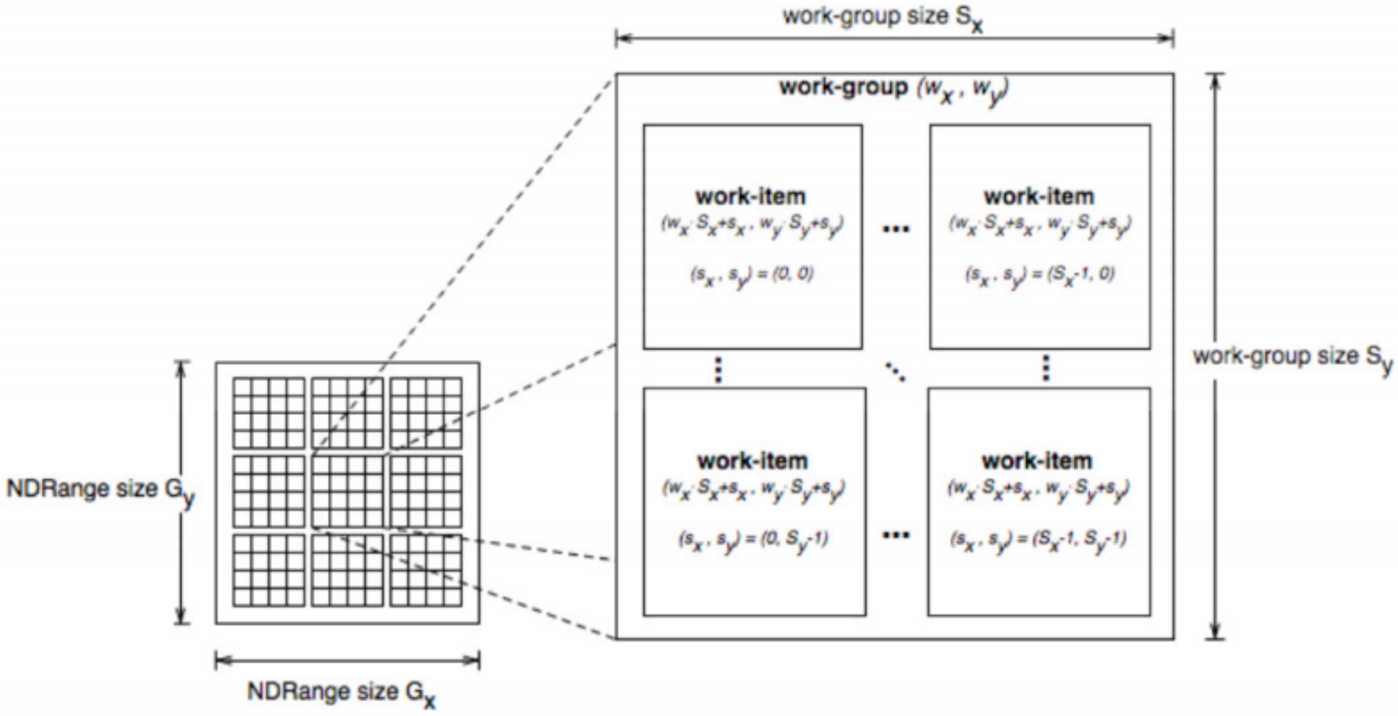
Work Group:

Work-groups exist to allow communication and cooperation between work-items. As work-items, work-groups also have an unique ID that can be referred from the kernel.

ND-Range:

Host program launches kernel in index space called NDRange. The N-Dimensional Range is a multitude of kernel instances arranged into 1, 2 or 3 dimensions.

Basic Concepts



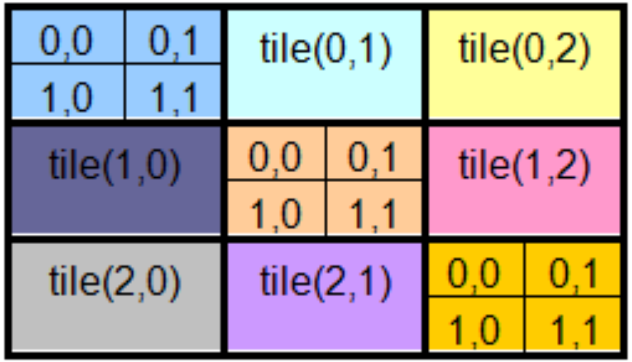
Execution Model

The OpenCL execution model is based on the parallel execution of a computational *kernel* over a 1D, 2D, or 3D grid, or *NDRange* ("N-Dimensional Range"). A single kernel instance, or *work-item*, operates at each point in a local grid while a *work-group* operates in a global grid.

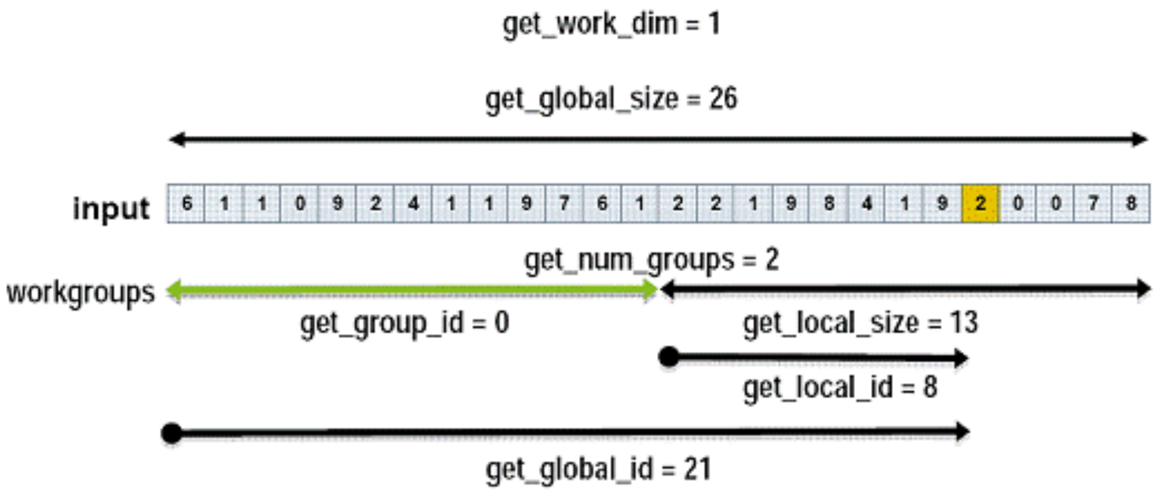
get_global_id(index)

get_local_id(index)

get_work_dim()



Execution Model



Execution Model

Aside from terminology, an essential aspect of the OpenCL execution model is the definition of a unique global and set of local IDs for each work-item in the multidimensional space defined via the NDRanges. Through these unique identifiers, the OpenCL execution model allows the developer to **exactly identify where each parallel instance of a kernel resides in the index space** so it can perform the necessary computations required to correctly implement an application.

OpenCL Memory

Global Memory:
Stores data for the entire device. This memory is commonly the largest memory region on an OpenCL device, but it's also the slowest for work-items to access.

Constant Memory:
A region of global memory that remains constant during the execution of a kernel.

OpenCL Memory

Local Memory:

Stores data for the work-items in a work-group. Work-items can access local memory much faster (~100x) than they can access global/constant memory. Work items in the same work group can access the same block of local memory.

OpenCL Memory

Private Memory:

Stores data for an individual work-item. Each work-item has exclusive access to its private memory and it can access this memory faster than it can access local memory or global/constant memory. But this address space is much smaller than any other address space, so it's important not to use too much of it.

[An Article on Memory Spaces]

[Memory Model]

[Address Space Qualifiers]

OpenCL Memory

Following is the *try_slow_iii.cl* kernel. (The 'iii' notation indicates that this kernel uses three integer vectors). This kernel is slow because the processor must decrement the b array in place in global memory.

Collapse | Copy Code

```
__kernel void vec_iii_1d(__global int *a, __global int *b,
                         __global int *c)
{
  size_t tid = get_global_id(0);
  c[tid] = a[tid];
  while(b[tid] > 0) {
    b[tid]--;
    c[tid]++;
  }
}
```

In comparison, the *try_fast_iii.cl* kernel uses private memory to speed the increment and deincrement operations. Specifically, the variable tmp is contains the value of the a vector for each value of tid. Similarly, the private variable i is loaded with the value of b.

Collapse | Copy Code

```
__kernel void vec_iii_1d(__global int *a, __global int *b,
                         __global int *c)
{
  size_t tid = get_global_id(0);
  int tmp = a[tid];
  for(int i=b[tid]; i > 0; i--) tmp++;
  c[tid] = tmp;
}
```

Kernel

[The OpenCL C Programming Language]

C code with some restrictions and extensions

<u>File name is in the form:</u>

<*kernel name*>.cl

<u>Code is in the form:</u>

__kernel void <*kernel name*>(<*parameter*>,<*parameter*>,...){

//code

}

Kernel

Example: Addition of 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

Example: Addition of two vectors, on GPU.

```
__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];
}
```

Kernel

```
/* get_global_id(0) returns the ID of the thread in execution.
   As many threads are launched at the same time, executing the same
kernel,
   each one will receive a different ID, and consequently perform a
different computation.*/



/* Now each work-item asks itself: "is my ID inside the vector's range?"
   If the answer is YES, the work-item performs the corresponding
computation*/
```

Kernel

A more complicated kernel example here.

Built-in functions here or here.

Host Application

Prepare and trigger device code execution

– Create and manage device context(s) and associate work queue(s), etc…

– Memory allocations, memory copies, etc

– Kernel launch

Host Application

Declarations:

```
cl_device_id device_id;              // compute device id
cl_context context;                  // compute context
cl_command_queue commands;           // compute command queue
cl_program program;                  // compute program
cl_kernel kernel;                    // compute kernel
size_t global;                       // global domain size for our calculation
size_t local;                        // local domain size for our calculation
cl_mem input;                        // device memory used for the input array
cl_mem output;                       // device memory used for the output array
```

Host Application

Functions:

clGetDeviceIDs

clCreateContext

clCreateCommandQueue

clCreateProgramWithSource

clBuildProgram

clCreateKernel

clSetKernelArg

clCreateBuffer

clEnqueueWriteBuffer

clEnqueueReadBuffer

clGetKernelWorkGroupInfo

clEnqueueNDRangeKernel

clFinish

Host Application

Shutdown and cleanup:

```
clReleaseMemObject(input);
clReleaseMemObject(output);
clReleaseProgram(program);
clReleaseKernel(kernel);
clReleaseCommandQueue(commands);
clReleaseContext(context);
```

Other

- [Another Beautiful Start Point]
- [Image Addressing and Filtering]
- [Memory Objects: Buffer and Image]
- [3D Fluid Simulation Sample]
- [Tim Matteson's Supercomputing 2009 tutorial]
- [Synchronization]
  - [Within a Work-group]
  - [Command Queues]
- [An Example Using Work-groups]

References

- A Gentle Introduction to OpenCL
- Getting started with OpenCL
- CodePlex OpenCL Tutorial
- An example: Vector Adding
- Work Groups and Synchronization
- OpenCL Memory Spaces

Questions?

Hilal Özdemir

BITES
Defence & Aerospace Technologies Inc.
July 2014

Thank you for listening.