General Purpose GPU Programming

PARALLEL COMPUTING/PARALLEELARVUTUSED
(MTAT.08.020)

Lecturer
Mohammad Anagreh
A general-purpose GPU (GPGPU) is a graphics processing unit (GPU) that performs non-specialized calculations that would typically be conducted by the CPU [1].

We use the GPU to perform some formerly conducted on high-power CPUs, such as
- Physics calculations,
- Encryption/decryption,
- Scientific computations,
- Cryptocurrencies such as Bitcoin,
- Image processing.
GPU vs. CPU

- A graphics processing unit is able to render images more quickly than a central processing unit because of its parallel processing architecture, which allows it to perform multiple calculations at the same time.

- We exploit this feature of GPU to perform different computations not only on the classical calculation (image processing).

- The ability of the GPU for performing the calculation is more (faster) than the CPU.

- A CPU has a higher clock speed, meaning it can perform an individual calculation faster than a GPU. But GPU is more efficient if there is a lot of computation at the same time [1].
CPU is more efficient in small-scale iterative algorithms.

GPU performance is better than CPU in large-scale parallel computing iterations.

A GPU and CPU combination can deliver much better performance.
CPU vs. GPU

- CPUs consist of a few cores optimized for serial processing or a limited parallel.
- GPUs consist of a thousand smaller, more efficient cores.
- We use one code consisting of two parts, Host and Kernel codes.
  - A serial portion of the code runs on the CPU (Host Code).
  - A parallel portion of the code runs on the GPU (Kernel code).
Nvidia vs. Intel

The link below is for showing the specification of different kinds of GPU hardware:

CUDA® is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs).

With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs [3].

CUDA is able to run on Windows, Linux, and MacOS, but only using NVIDIA hardware.

OpenCL is available to run on almost any operating system and most hardware varieties.

CUDA is able to run on the leading operating systems while OpenCL runs on almost all.
void function(...) {
    Allocate memory on the GPU
    Transfer input data to the GPU
    Launch kernel on the GPU
    Transfer output data to CPU
}

__global__ void kernel(...) {
    Code executed on
    the GPU goes here...
}
OpenCL (Open Computing Language) is a framework for writing programs that execute in parallel on different compute devices (such as CPUs and GPUs) from different vendors (AMD, Intel, ATI, Nvidia etc.).

OpenCL code has two codes part. Host code and Kernel code.

Host code has the general structure of the OpenCL code and calling of the kernel code.

The framework defines a language to write “kernels” in. These kernels are the functions which are to run on the different compute devices.

First of all you need to download the newest drivers to your graphics card. This is important because OpenCL will not work if you don’t have drivers that support OpenCL.

To download and install the OpenCL, from following link:
OpenCL Features

- Standards maintained by the Khronos group, Currently 1.0, 1.1, and 1.2.
- Major backers of OpenCL: Apple, AMD, Intel.
- Alternative to CUDA
- Designed for “heterogenous computing”
- Executable on many devices, including CPUs, GPUs, DSPs, and FPGAs
- Similar structure of host programs and kernels
- It has some restrictions in OpenCL as:
  - No recursion, variadics, or function pointer.
  - Cannot dynamically allocate memory from device
Some companies in OpenCL working groups are as follows:
### CUDA vs. OpenCL

<table>
<thead>
<tr>
<th>Comparison</th>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>No specific, dependent code quality, hardware type and other variables</td>
<td>No specific, dependent code quality, hardware type and other variables</td>
</tr>
<tr>
<td>Vendor Implementation</td>
<td>Implemented by only NVIDIA</td>
<td>Implemented by TONS of vendors including AMD, NVIDIA, Intel, Apple, Radeon etc.</td>
</tr>
<tr>
<td>Portability</td>
<td>Only works using NVIDIA hardware</td>
<td>Can be ported to various other hardware as long as vendor-specific extensions are avoided</td>
</tr>
<tr>
<td>Open Source vs Commercial</td>
<td>Proprietary framework of NVIDIA</td>
<td>Open-Source standard</td>
</tr>
<tr>
<td>OS Support</td>
<td>Supported on the leading Operating systems with the only distinction of NVIDIA hardware must be used</td>
<td>Supported on various Operating Systems</td>
</tr>
<tr>
<td>Libraries</td>
<td>Has extensive high-performance libraries</td>
<td>Has a good number of libraries which can be used on all OpenCL compliant hardware but not as extensive as CUDA</td>
</tr>
<tr>
<td>Community</td>
<td>Has a larger community</td>
<td>Has a growing community not as large as CUDA</td>
</tr>
</tbody>
</table>
CUDA vs. OpenCL (Terminology)

CUDA:
- Host Memory
- Global/Device Memory
- Local Memory
- Constant Memory
- Shared Memory
- Registers
- Grid
- Block
- Thread
- Thread ID
- Block Index
- Thread Index

OpenCL:
- Host Memory
- Global Memory
- Global Memory
- Constant Memory
- Local Memory
- Private Memory
- NDRange
- Work group
- Work item
- Global ID
- Block ID
- Local ID
NDRange and work items

- NDRange is to specify the number of work items on a compute device.
- Work items are divided into work groups.
- Work group divided into wavefronts.
- Wavefront is a group of work items.
## Thread structure

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint get_work_dim()</code></td>
<td>The number of dimensions</td>
</tr>
<tr>
<td><code>get_global_id(uint dim)</code></td>
<td>The ID of the current work-item [0, WI) in dimension dim</td>
</tr>
<tr>
<td><code>get_global_size(uint dim)</code></td>
<td>The total number of work-items (WI) in dimension dimi</td>
</tr>
<tr>
<td><code>get_global_offset(uint dim)</code></td>
<td>The offset as specified in the <code>enqueueNDRangeKernel</code> API in dimension dimi</td>
</tr>
<tr>
<td><code>get_group_id(uint dim)</code></td>
<td>The ID of the current work-group [0, WG) in dimension dim</td>
</tr>
<tr>
<td><code>get_local_id(uint dim)</code></td>
<td>The ID of the work-item within the work-group [0, WI/WG) in dimension dim</td>
</tr>
<tr>
<td><code>get_local_size (uint dim)</code></td>
<td>The number of work-items per work-group = WI/WG in dimension dim</td>
</tr>
<tr>
<td><code>get_num_groups(uint dim)</code></td>
<td>The total number of work-groups (WG) in dimension dim</td>
</tr>
</tbody>
</table>
Memory model(1)

- **Buffer object** is a block of memory allocated by an OpenCL program. Usually it stores the input/output data that will be processed/generated by work items.

- **Constant memory** is part of the compute device memory and is read-only.

- **Global memory** is part of the compute device memory. Any work item can read/write any location within the global memory.
  - To find out the amount of global memory, call `clGetDeviceInfo()`
Memory model (2)

- **Host memory** is the memory attached to the host device. Usually, it is the computer’s main memory.
- **Local memory** accessible to all work-items (threads) within a workgroup.
- **Private memory** can only be accessed by one work item.
  - A kernel program can allocate a variable private memory by using the qualifier `__private`. 
One of the most difficult tasks of parallel programming is how to manage the sequence of multiple threads or processes, some of them, all, which one is before others.

- In other words, synchronization often means serialization, a thread waiting for another thread to finish or a thread waiting for a procedure.

Two types of synchronization:

- Synchronization among work items within a work group.
- Synchronization among command queues within a context.
Barrier

- Barrier operation lets work item waits until all work items have reached the same barrier, and then all work items continue to run.

- `work_group_barrier();` ➔ Work-items in a work-group must execute this before any thread can continue.

- `barrier(CLK_LOCAL_MEM_FENCE);`

- It has two flags
  - `CLK_GLOBAL_MEM_FENCE` (only for global memory)
  - `CLK_LOCAL_MEM_FENCE` (only for local memory)
Installing and setting up OpenCL

- First of all, you need to download the newest drivers to your graphics card.
- OpenCL will not work if you don't have drivers that support OpenCL.
- The major graphic vendors NVIDIA and AMD have both released implementations of OpenCL for their GPUs.
- Intel, also have their own OpenCL implementations.
- Apple Mac OS X, you need to use Apple’s OpenCL implementation, which should already be installed on your system.

- For AMD GPUs and CPUs download the [AMD APP SDK](#)
- For NVIDIA GPUs download the [CUDA Toolkit](#)
- For Intel CPUs/GPUs download the [Intel OpenCL SDK](#)
To use the GPU/CPU through OpenCL, one needs:

1. **A context** (linked to a device)
2. **A Program** (Program to host Kernels that will compile code for different devices)
3. **A Kernel** (method running in the device)
4. **A Command Queue** (to operate a device, either with FIFO or Events)
5. **A Buffer** (Allocation in the Global Mem. of a Device -- Linked to a context + device)
6. **Write to Buffer** (passing arguments into the Global Memory of the Device)
7. **Execute Kernel** (enqueueing the parallel execution in the GPU/CPU)
8. **Read from a buffer** (Get the result from the device back into the HOST program)
The host program controls the execution of kernels on the computing devices. The host program is written in C, but bindings for other languages like C++ and Python exist.

The main steps of a host program are as follows:

- Get information about the platform and the devices available on the computer.

  ```c
  cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
  ```

- Select devices to use in execution.

  ```c
  ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, &ret_num_devices);
  ```
Create an OpenCL context

```c
cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
```

Create a command queue

```c
cl_command_queue command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
```

Create memory buffer objects

```c
cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY, LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY, LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY, LIST_SIZE * sizeof(int), NULL, &ret);
```
The host program of OpenCL (3)

- **Transfer data (list A and B) to memory buffers on the device**

  ```c
  ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int), A, 0, NULL, NULL);
  ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int), B, 0, NULL, NULL);
  ```

- **Create program object**

  ```c
  cl_program program = clCreateProgramWithSource(context, 1, (const char **) &source_str, (const size_t *)source_size, &ret);
  ```
The host program of OpenCL (4)

- Load the kernel source code

```c
FILE *fp;
char *source_str;
size_t source_size;

fp = fopen("vector_add_kernel.cl", "r");
if (!fp)
{
    fprintf(stderr, "Failed to load kernel.\n");
    exit(1);
}

source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);
```
The host program of OpenCL (5)

- Compile the Kernel (online execution) or load the precompiled binary OpenCL program (offline execution).
  
  ```c
  ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
  ```

- Create kernel object

  ```c
  cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);
  ```
The host program of OpenCL (6)

- **Set kernel arguments**

  ```c
  ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
  ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
  ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);
  ```

- **Execute the kernel**

  ```c
  ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
                             &global_item_size, &local_item_size, 0, NULL, NULL);
  ```
The host program of OpenCL (7)

- Read memory objects. In this case, we read the list C from the computing device

```c
int *C = (int*)malloc(sizeof(int)*LIST_SIZE);
ret = clEnqueueReadBuffer(command_queue, c_mem_obj, CL_TRUE, 0, LIST_SIZE * sizeof(int), C, 0, NULL, NULL);
```
Kernel program is the program that will be processed in the GPU platform

```c
__kernel void vector_add(__global const int *A, __global const int *B, __global int *C)
{
    // Get the index of the current element to be processed
    int i = get_global_id(0);
    // Do the operation
    C[i] = A[i] + B[i];
}
```
Void trad_mul(int n,
    const float *a,
    const float *b,
    float *c)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] * b[i];
}

kernel void dp_mul(global const float *a
    global const float *b,
    global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over “n” work-items
Multiple kernels

- The GPU’s main function, kernel, is invoked from the CPU host code.

- CPU host code can process multiple kernels codes.

- Each Kernel code has same or different functionality.
We need to release memory from the objects, kernels, and program once we finish the processing.

- To release a memory object, call the function
  - `clReleaseMemObject(cl_mem bufC)`
- To release a kernel, call the function
  - `clReleaseKernel(cl_kernel kernel_1)`
- To release a program. Call the function
  - `clReleaseProgram(cl_program program)`
- To release the command queue, call the function
  - `clReleaseCommandQueue(cl_command_queue command_queue)`
- To release the context, call the function
  - `clReleaseContext(cl_context context)`
The Source Code

The source code (Hello World) for this example can be downloaded from here

THANK YOU FOR ATTENTION
References