Introduction of General Purpose of Graphics Processor Unit (GPGPU)

MTAT.08.020 Paralleelarvutused / Parallel Computing
Instructor: Prof. Eero Vainikko
Presentor: Mohammad Anagreh
9- NOV -2020
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 calculation that were formerly the domain of high-power CPUs, such as physics calculations, encryption/decryption, scientific computations, crypto currencies such as Bitcoin and 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.
- Ability of GPU for performing the calculation is more than CPU.
- A CPU has a higher **clock speed**, meaning it can perform an individual calculation faster than a GPU. But in case, there are a lot of calculation at the same time GPU is better[1].
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 consist of two parts, Host and Kernel codes.
  - Serial portion of the code run on the CPU (Host Code).
  - Parallel portion of the code run on the GPU (Kernel code).
Nvidia vs. Intel

The link below is for showing the specification of different kind 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].
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 new 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
- Same code can be run on different devices
- It has some restrictions in OpenCL as:
  - No recursion, variadics, or function pointer.
  - Cannot dynamically allocate memory from device
OpenCL working Group

Some companies in OpenCL working groups as follows:
Terminology CUDA vs. OpenCL

**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
To use the GPU/CPU through OpenCL, one needs:
- **A context** (linked to a device)
- **A Program** (Program to host Kernels, that will compile code for different devices)
- **A Kernel** (method running in the device)
- **A Command Queue** (to operate a device, either with FIFO or Events)
- **A Buffer** (Allocation in the Global Mem. of a Device -- Linked to a context + device)
- **Write to Buffer** (passing arguments into the Global Memory of the Device)
- **Execute Kernel** (enqueueing the parallel execution in the GPU/CPU)
- **Read from buffer** (Get the result from the device back into the HOST program)
The host program of OpenCL (1)

- The host program controls the execution of kernels on the compute 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 is as follows:
  
  - Get information about the platform and the devices available on the computer.
    
    ```
    cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
    ```
  
  - Select devices to use in execution.
    
    ```
    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);
```
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);
```
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 );
```
Compile the Kernel (online execution) or load the precompiled binary OpenCL program (offline execution).

ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

Create kernel object

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);
  ```
Read memory objects. In this case we read the list C from the compute 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
The source code for this example can be downloaded from here:
THANK YOU FOR ATTENTION
References