Practical 3 - OpenCL

Roan J. Song[†] and Sean P. Le Roux[‡]
EEE4084F Class of 2017
University of Cape Town
South Africa
[†]SNGROA001 [‡]LRXSEA001

***Abstract*—This report concerns the implementation of OpenCL computations on both CPU and GPU devices. Comparisons between the performance of the two devices in different applications will be made.**

I. Introduction

Open Computing Language (OpenCL) is a computing standard that allows portable implementation of programs across different computing platforms. Different platforms are more suited to different applications, and so this report aims to analyse the difference in performance between an Intel CPU and an NVIDIA GPU when faced with the same tasks.

II. Methodology

A. Device Characteristics

It is important to know the characteristics of the devices being used for computation. The architecture of a device can determine its suitability for sequential work, or the degree of its parallel effectiveness. OpenCL provides built-in methods to view the characteristics of its devices.

```
clGetDeviceInfo(clGetDeviceIDs()[0], cl_device_info.###) // Intel Device
clGetDeviceInfo(clGetDeviceIDs()[1], cl_device_info.###) // NVIDIA Device
// where ### is the device parameter in question
```

B. Kernel Performance on CPU vs GPU

OpenCL kernels perform a function and can be assigned to an OpenCL device for execution. Kernels can be executed sequentially, in parallel on a single device, or on multiple devices. This section focuses on analysing the performance of a sum kernel running on the provided Intel CPU versus the provided NVIDIA GPU. The implementation requires writing a sum kernel, shown below:

```
kernel void sum(global float *a, global float *b, global float *c){
  int gid = get_global_id(0);
  c[gid] = a[gid] + b[gid];}
```

An OpenCL context is created for each device:

```
nvidia_platform = pyopencl.get_platforms()[0]
nvidia_devices = nvidia_platform.get_devices()
nvidia_context = pyopencl.Context(devices=nvidia_devices)
intel_platform = pyopencl.get_platforms()[1]
intel_devices = intel_platform.get_devices()
intel_context = pyopencl.Context(devices=intel_devices)
```

The sum kernel is built for each device:

```
nvidia_program_source = pyopencl.Program(nvidia_context,program_source)
nvidia_program = nvidia_program_source.build()
intel_program_source = pyopencl.Program(intel_context,program_source)
intel_program = intel_program_source.build()
```

CPU-only program:

```
# Code provided by the 2017 EEE4084F OpenCL Workshop
# Memory buffers are created
a_intel_buffer = pyopencl.Buffer(intel_context,
                                 flags=pyopencl.mem_flags.READ_ONLY,
                                 size=a.nbytes)
b_intel_buffer = pyopencl.Buffer(intel_context,
                                 flags=pyopencl.mem_flags.READ_ONLY,
                                 size=b.nbytes)
c_intel_buffer = pyopencl.Buffer(intel_context,
                                 flags=pyopencl.mem_flags.WRITE_ONLY,
                                 size=c.nbytes)
# Command Queue
intel_queue = pyopencl.CommandQueue(intel_context)
def run_cpu_program():
    #copying data onto CPU
    pyopencl.enqueue_copy(intel_queue,
                          src=a,
                          dest=a_intel_buffer)
    pyopencl.enqueue_copy(intel_queue,
                          src=b,
                          dest=b_intel_buffer)

    #running program
    kernel_arguments = (a_intel_buffer,b_intel_buffer,c_intel_buffer)
    intel_program.sum(intel_queue,
                       a.shape, #global size
                       None, #local size
                       *kernel_arguments)

    #copying data off CPU
    copy_off_event = pyopencl.enqueue_copy(intel_queue,
                                           src=c_intel_buffer,
                                           dest=c)
    copy_off_event.wait()
```

GPU-only program:

```
# Code provided by the 2017 EEE4084F OpenCL Workshop
# Memory buffers are created
a_nvidia_buffer = pyopencl.Buffer(nvidia_context,
                                 flags=pyopencl.mem_flags.READ_ONLY,
                                 size=a.nbytes)
b_nvidia_buffer = pyopencl.Buffer(nvidia_context,
                                 flags=pyopencl.mem_flags.READ_ONLY,
                                 size=b.nbytes)
c_nvidia_buffer = pyopencl.Buffer(nvidia_context,
                                 flags=pyopencl.mem_flags.WRITE_ONLY,
                                 size=c.nbytes)
# Command Queue
nvidia_queue = pyopencl.CommandQueue(nvidia_context)
def run_gpu_program():
    #copying data onto CPU
    pyopencl.enqueue_copy(nvidia_queue,
                          src=a,
                          dest=a_nvidia_buffer)
    pyopencl.enqueue_copy(nvidia_queue,
                          src=b,
                          dest=b_nvidia_buffer)

    #running program
    kernel_arguments = (a_nvidia_buffer,b_nvidia_buffer,c_nvidia_buffer)
    nvidia_program.sum(nvidia_queue,
                       a.shape, #global size
                       None, #local size
                       *kernel_arguments)

    #copying data off CPU
    copy_off_event = pyopencl.enqueue_copy(nvidia_queue,
                                           src=c_nvidia_buffer,
                                           dest=c)
    copy_off_event.wait()
```

C. Counting Factors

A more complex task to allocate to the devices is generating a matrix of random integers, and calculating the number of factors each element has. This required the implementation of a new kernel, shown below.

```
kernel void checkfactors(global int *X, global int *c, int row_length){
  int gid = row_length*get_global_id(0) + get_global_id(1);
  int num = 1;
  int max = X[gid];
  if(X[gid] > 100)
    max = 100;
  else
    max = X[gid];
  for(int i = 2; i <= max; i++){
      if(X[gid]%i == 0)
          num++;}
  c[gid] = num;}
```

More than one operation is conducted for every element in the matrix, and this should lead to a different set of results when compared to the sum kernel mentioned previously. For simplicity it was decided that square matrices should be used for testing, but rectangular matrices can be used with only slight alteration to the code. To initialise the program, the size of the matrix is specified, and filled with random integers. An empty matrix is initialised, and this is where the output of the function will be written.

```
N = M = numpy.int32(1e4)
X = numpy.random.randint(100,size=(N,M)).astype(numpy.int32)
c = numpy.empty_like(X)
```

The program is built in much the same way as the sum kernel was, for both the CPU and GPU, although one less memory buffer is required and the parameters taken by the kernel function are slightly different.

III. CPU VS GPU COMPARISON

A. Device Characteristics

The two devices being used are both examples of high-end consumer hardware, suited for hobbyist high performance applications. Table I shows the different characteristics of both devices.

Device Characteristic	Intel Device	NVIDIA Device
Device name:	Intel(R) Xeon(R) CPU E5-2686 v4	Tesla K80
Global memory (MB):	61406	11439
Local memory (KB):	32	48
Max. compute units:	4	13
Max. work groups:	8192	1024
Max. work items:	[8192, 8192, 8192]	[1024, 1024, 64]
Max. memory allocation (bytes):	16097245184	2998894592
Max. clock frequency (MHz):	2300	823

TABLE I: Comparison of device characteristics

B. Sum kernel performance

Table II shows the results of running the sum kernel on both CPU and GPU, on a variety of different vector sizes. Each program was run twice, to eliminate variance from caching. Vectors of up to 10^7 elements were run successfully. Trying with 10^8 elements failed to run, as the memory buffers on either device did not have the requisite cache size for storage. An implementation could be made where elements are swapped in and out of device memory to allow for 10^8 elements to be processed, but that would introduce speed differences due to memory operations and would be outside the scope of this task.

Vector Size	CPU Time [ms]	GPU Time [ms]	CPU/GPU speed-up
1e2 (run 1)	1.81	1.55	1.168
1e2 (run 2)	1.81	1.55	1.168
1e4 (run 1)	1.89	1.59	1.189
1e4 (run 2)	1.89	1.59	1.189
1e6 (run 1)	3.60	3.43	1.050
1e6 (run 2)	3.60	3.42	1.053
1e7 (run 1)	55.2	16.4	3.367
1e7 (run 2)	54.8	16.4	3.341
1e8 (run 1)	476	146	3.260
1e8 (run 2)	475	146	3.253

TABLE II: CPU vs GPU performance on sum kernel

C. Factor count performance

Table III shows the results of running the factor count program on both CPU and GPU, on a variety of different matrix sizes. The golden measure (GM) was performed sequentially using Python code on the system's Intel CPU. This gives a realistic comparison to the parallel devices of the system, rather than comparing it with the performance of an unrelated machine, in a different hardware/software environment. The golden measure took too long to complete on datasets larger than 10,000 elements. The factor count program is a coarse parallel operation, requiring no communication between threads. It has more computation per thread than the sum kernel, which helps to explain the difference in speed-up between the two programs.

Matrix Size	CPU Time [ms]	GPU Time [ms]	CPU/GPU speed-up
1e2 (GM)	17.9	-	-
1e4 (GM)	1800	-	-
1e2 (run 1)	2.08	1.55	1.342
1e2 (run 2)	2.07	1.54	1.344
1e4 (run 1)	2.46	1.57	1.567
1e4 (run 2)	2.47	1.58	1.563
1e6 (run 1)	61.5	4.35	14.14
1e6 (run 2)	61.5	4.35	14.14
1e8 (run 1)	5980	283	21.13
1e8 (run 2)	5980	284	21.06

TABLE III: CPU vs GPU performance counting factors

D. Average Speed-up

Analysis of the results shows that the different tasks were both favoured by the GPU as the number of elements increased. Figure 1 demonstrates that the sum program yielded less speed-up, and had only began to show improvement between 10^6 and 10^7 elements. The average speed-up of the GPU versus the CPU implementation was 2.0038; almost exactly double. The factor count program began to show improvements earlier, and continued to increase until the limits of the devices' memory was reached. Its average speed-up was 9.536 over all cases, but this is somewhat misleading, given the large discrepancy between small and large vector sizes.

IV. CONCLUSIONS

OpenCL is evidently a powerful tool for developing unified multi-platform computing solutions. Given different platforms

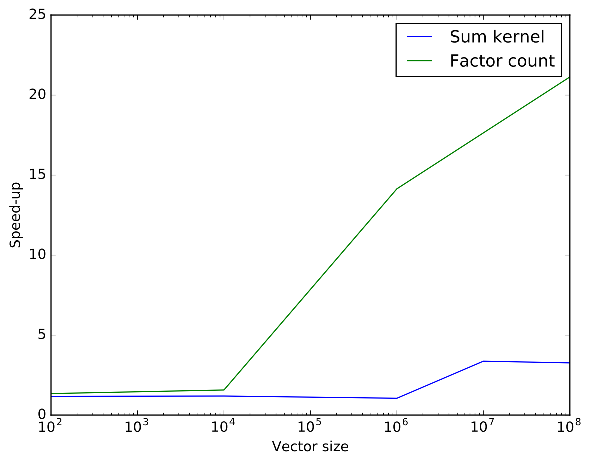

Fig. 1: Average speed-up of GPU/CPU vs vector size

with which to work, it is important to understand the strengths and capabilities of each, in order to allow them to be applied to tasks suited to them. This report has found that the NVIDIA GPU is suited to coarse parallel tasks and offers significant speed-up over the Intel CPU with which it is paired. This speed-up is proportional to the size of the data at hand. If working on datasets of less than 10,000 items, the average speed-up is 1.5, which may not justify an investment into the more expensive high-end GPU.
Consideration needs to be taken towards the predominant type of computations run on the system, as this will change the point at which the speed-up is effected by larger data sets. These trade-offs can be estimated when designing a HPEC system, but it is recommended to sample a greater number of datasets as well as a more diverse range of kernels; accurate depiction of the trade-off points for various hardware types will help to optimize the systems' performance at minimal cost.