Digital Systems

EEE4084F

Practical 3 – OpenCL using Amazon Web Services

[30 Marks]

Introduction

The focus of this practical is developing OpenCL code and doing performance testing. Prac3 is done in collaboration from Dr Gordon Inggs from the Amazon Engineering Team in Cape Town. If you have not already got a team partner please find a classmate to team up with. On the Vula site see the wiki for the assigned lab teams, please change the wiki (e.g. swapping names around) to form a group for a particular slot number. Each prac pair gets an Amazon AWS private virtual machine in the Amazon cloud. Please ask Gordon Inngs to provide a machine IP and login if your team doesn't have one yet; there is one login per team (you need to share the login).

What needs to be submitted

For this practical you need to prepare a Prac3.pdf report that responds to the tasks for which you are requested to provide a response and documentation in your report. See headings indicating "To submit". Submit the report using Vula assignment Prac 3.

The Programming Model

OpenCL uses a programming and memory model in which the CPU must set up and compile a kernel for an OpenCL device context (e.g. the GPU), then set up and copy data to the context and then run the kernel (an OpenCL worker) on the data put into the context. When the kernel is finished the CPU needs to read back the results from the device context.

Task #1: Complete Lab 1 (not for marks)

Complete the first practical, this will be available in a Jupiter Notebook, once you are given an IP address go to the site IP/opencl_workshop2017 in your browser and you should see a list of JupiterNotebooks, double click on the first one to get in to it.

Read through the steps and execute the code in the sequence shown, this is done by pressing the →| step button on the control panel for the window.

From this you should get an understanding of what the program is doing.

Task #2: Report device characteristics

Add code (e.g. to the block in Jupiter that gets the devices) to print out characteristics of each device using (e.g.) the clGetDeviceInfo().e.g.

d = clGetDeviceIDs()[0] *(or skipping this and using the already initialized nvidia_devices should be fine)*

clGetDeviceInfo(d, cl_device_info.CL_DEVICE_NAME)

See https://media.readthedocs.org/pdf/pycl/latest/pycl.pdf for more detail

Want to know what the configuration is for:

- CL_DEVICE_NAME
- CL_DEVICE_GLOBAL_MEM_SIZE
- CL_DEVICE_MAX_COMPUTE_UNITS,
- CL_DEVICE_MAX_WORK_ITEM_SIZES,
- CL_DEVICE_MAX_MEM_ALLOC_SIZE, and
- CL_DEVICE_MAX_CLOCK_FREQUENCY

This should provide information printed out as per the example below:

Name: Intel(R) OpenCL

CUDA Global memory size : 1073741824

Global memory cache size: 65536

Local memory size : 49152

Maximum compute units : 4

…

To submit in report:

Indicate the code that you used to obtain the device information and provide at least the following details about each device: global memory and local memory size, the number of compute units and the maximum work group size.

[10 marks]

Task #3: Performance Analysis of CPU vs GPU 1

The objective for this task is to compere the performance of an Intel CPU OpenCL kernel to that of an nVidia GPU kernel. The platform you have been given access to should have two OpenCL devices, one named something like “Intel(R) OpenCL” and another like “NVIDIA CUDA” – the former is provided for running on the Intel multicore CPU that the platform provides and the latter is for the nVidia GPU.

Create a context for each of the available devices, build the sum kernel for each device. Implement a run_cpu_program and a run_gpu_program functions which load and run kernels; the functions should each send the same a and b vectors to both contexts, then run in the kernels and read the c vector result back to the cpu memory.

Time how long the run_cpu_program and run_gpu_program takes (note that these times need to include the time for moving memory). You can use the commands as follows in Python to time the kernel execution:

```
%timeit run_cpu_program()
%timeit run_gpu_program()
```

To submit in report:

Please submit your code for `run_cpu_program` and for `run_gpu_program`.

Provide run statistics. For vector sizes of 1e2, 1e4, 1e8, 1e9 do the following: perform two runs (run 1 and run 2) and provide a table comparing the run_cpu_program execution time to the run_gpu_program execution time. Prove speedup statistics for each row. Provide a graph showing the speedup of the gpu vs. the cpu versus the vector size. Discuss the results, does there appear to be a trend? Indicate the average speedup for all the runs.

[10 marks]

Task #4: Implement factor count program compare CPU vs GPU speed

The objective for this task is to implement a C kernel for OpenCL that computes the number of factors for a particular element in a matrix. Given an NxM matrix **X** you need to calculate the number of factors for each element X[i,j] of the matrix. To simplify the assignment you can assume that the numbers will not have any factor greater than 100. You can use the simpleminded approach below provided as a m file that runs in OCTAVE. Convert this into an OpenCL kernel and integrate it with a Python or C / C++ CPU part to load and run the kernel. Also provide a golden measure. You can also provide the run time of the OCTAVE version (which will be a shockingly slow operation for big arrays).

```
# Program to return the number of factors
# for each element of the input matrix

function [factors] = checkfactors (X)
  [N,M] = size(X);
  factors=zeros(N,M);
  for i=1:N
     for j=1:M
        nf=0;
        for k=2:100
           if (mod(X(i,j),k)==0)
              nf=nf+1;
           end
        end # k
        factors(i,j)=nf;
     end # j
   end  # i
endfunction
```

M file: checkfactors.m

```
>> # generate 5x10 matrix of random ints from 0 to 100
>> X = randi(100,5,10)
X =
   14   78   48   87   11   96    8   21    6   15
    4   50   15   53   88   22   91   26    5   30
   36   29   22   10   16   82   24   22   82   47
   61   84   63   87   33    8   22   64    3   99
   55    9    7   48   31   47   18   59    9   28

>> Y=checkfactors(X)
Y =
  3  7  9  3  1 11  3  3  3  3
  2  5  3  1  7  3  3  3  1  7
  8  1  3  3  4  3  7  3  3  1
  1 11  5  3  3  3  3  6  1  5
  3  2  1  9  1  1  5  1  2  5
>>
```

Example run of OCTAVE code to test the checkfactors routine with sample output

To submit in report:

Please submit your C / Python code (or Jupiter notebook) for your OpenCL application. Provide both the Python code and the C kernel code. You can indicate your kernel code in the report. In your report show the usual thing of golden measure time, OpenCL application time and Speedup. Try for a range of matrix sizes, e.g. 10x10, 100x100 and 1000x1000 (if not more sizes to get an interesting looking speed comparison graph).

Note that you can choose to use C++ instead of Python for your CPU side cide, see http://www.rrsg.ee.uct.ac.za/courses/EEE4084F/Resources/Practicals/Prac3B/Source.zip for a starting point (this can be run either in the BlueLab or using the Amazon machine assigned to you, although you probably need to create a modified makefile in that case as Code::Blocks probably isn't available on the machine).

[10 marks]

REPORT REQUIREMENTS

Compile your experiments and findings into an IEEE-style conference paper. You can use Word, OpenOffice or Latex (I recommend TexStudio) – templates for Word and Latex is on the EEE4084F website (http://www.rrsg.ee.uct.ac.za/courses/EEE4084F/Practicals.html).

There is no page limit (although more than 5 pages is excessive and marking is not guaranteed to be done after the 5th page), but try to keep it below about 3 pages. Submit your report to the Vula Assignment for this practical.

END OF ASSIGNMENT