Scan Amino Acid sequence

Contents

Goals of this tutorial

1. Test a single OpenCL program using pyopencl throw CPU and GPU
2. Implement a amino acid comparator in every work-item
3. Test usage of local memory in kernels

Run a openCL Kernel over CPU and GPU

soroban server have a OpenCL implementation on CPU with AMD OpenCL 1.1 SDK and OpenCL 1.2 over GPU using NVIDIA CUDA Driver This is the occasion to you to take in hand portability of OpenCL.

- Run this script

```
# code inspired from example provided by Roger Pau Monn'e

import pyopencl as cl
import numpy
import numpy.linalg as la
import datetime
from time import time


# number of data
num_data=4*25000*128
fragment_size= 200
sequence = numpy.random.randint(0,20,num_data).astype(numpy.uint8)
fragment = numpy.random.randint(0,20,fragment_size).astype(numpy.uint8)

# just for test, inject fragment somewhere in sequence
sequence[100:100+fragment_size]=fragment
score = numpy.empty(num_data).astype(numpy.uint16)


for platform in cl.get_platforms():
    for device in platform.get_devices():
                print("===============================================================")
                print("Platform name:", platform.name)
                print("Platform profile:", platform.profile)
                print("Platform vendor:", platform.vendor)
                print("Platform version:", platform.version)
                print("---------------------------------------------------------------")
                print("Device name:", device.name)
                print("Device type:", cl.device_type.to_string(device.type))
                print("Device memory: ", device.global_mem_size//1024//1024, 'MB')
                print("Device max clock speed:", device.max_clock_frequency, 'MHz')
                print("Device compute units:", device.max_compute_units)

                ctx = cl.Context([device])
                queue = cl.CommandQueue(ctx,
                                properties=cl.command_queue_properties.PROFILING_ENABLE)
```

```
                mf = cl.mem_flags

                sequence_buf = cl.Buffer(ctx, mf.READ_ONLY, sequence.nbytes)
                fragment_buf = cl.Buffer(ctx, mf.READ_ONLY, fragment.nbytes)
                score_buf = cl.Buffer(ctx, mf.WRITE_ONLY, score.nbytes)
                time1 = time()
                cl.enqueue_write_buffer(queue, sequence_buf, sequence)
                cl.enqueue_write_buffer(queue, fragment_buf, fragment)
                prg = cl.Program(ctx, """
                        __kernel void scanSequence(__global const uchar *sequence,
                        __global const uchar *fragment, __global ushort *score, const uint size_data, c
                        {
                                int gid = get_global_id(0);
                                ushort s=0;
                                // TODO: implement a scan of fragment in current sequence element
                                score[gid]=s;
                                }
                                """).build()

                exec_evt = prg.scanSequence(queue, (num_data,), None,
                                                                        sequence_buf,
                                                                        fragment_buf,
                                                                        score_buf,
                                                                        numpy.uint32(num_data),
                                                                        numpy.uint32(fragment_size))
                exec_evt.wait()
                elapsed = 1e-9*(exec_evt.profile.end - exec_evt.profile.start)
                cl.enqueue_read_buffer(queue, score_buf, score).wait()
                time2 = time()
                execTime2 = time2 - time1
                print("Execution time of test: %g s" % elapsed)
                print("Execution time of test + memory transfert time: %g s" % execTime2)

                print (score[90:110])
```

This script try to use all plateforms and device OpenCL installed.

Implement a naive Amino Acid comparator

Proteins are composed of sequences of about 20 amino acids. This is a very naive scan of amino acid fragments in some sequences.

- Modify kernel to compare a amino acid fragment to a part of a amino acid sequence. use *s* has intermediate score accumulator. Every *work-item* process a different par of sequence. If to acids match score is incremented by one.
- To test your code, fragment are inserted in sequence in position 100. If your code work well score must be equal to fragment size a position 100

See more about amino acid here (http://en.wikipedia.org/wiki/Amino_acid) Quick Reference Card of OpenCL 1.2 (http://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf)

Test with local memory

This problem is memory dominant problem. So you can test to preload data in local memory.

- Add a local memory data in kernel with:

```
__local uchar* local_data
```

- Add in kernel call a parameter for local memory reservation like this:

```
                exec_evt = prg.scanSequence(queue, (num_data,), None,
                                                                    sequence_buf,
                                                                    fragment_buf,
                                                                    score_buf,
                                                                    numpy.uint32(num_data),
                                                                    numpy.uint32(fragment_size),
                                                                    cl.LocalMemory(shared_mem_size))
```

- You can define **shared_mem_size** like this:

```
shared_mem_size = number_elements*numpy.uint8(1).nbytes
```

- Define manually the size of each work-group in kernel call :

```
                exec_evt = prg.scanSequence(queue, (num_data,), (size_group,),
                                                                    sequence_buf,
                                                                    fragment_buf,
                                                                    score_buf,
                                                                    numpy.uint32(num_data),
                                                                    numpy.uint32(fragment_size),
                                                                    cl.LocalMemory(shared_mem_size))
```

Back to Parallel Computing tutorials