DTU GPU-Lab PyOpenCL Workshop (A. Klöckner)

Advanced Lab

This problem set is intended to train you in applying the concepts taught in the workshop. While Problem 1 is designed to be a straightforward application of what you've learned, Problem 2 is somewhat more open-ended.

Problem 1: Code Generation

Use the Mako templating language language[1] to write a vector addition code generator.

You'll write a versions of this generator for each subproblem. Be sure to keep all of them in one file so that you can easily benchmark them against each other.

a) Start by writing a simple (non-generating) code that adds two single-precision PyOpenCL arrays together. Invent a way to obtain test data. Use one work item per vector entry.

b) Use Mako to substitute different types into your code. (*Hint:* You can find the type of the PyOpenCL or numpy array '`ary`' by looking at '`ary.dtype.char`'. You can convert these type characters to C types through a Python dictionary.)

c) Now compute multiple vector entries per work item. This will lead to a for loop within each work item. Unroll this for loop to a user-given number of iterations using code generation. Make sure your code is still correct even if the vector length is not divisible by the unroll length. You might want to generate a separate, conditional-free fast-path code section.

d) Further modify your code generator so it does the unrolled reads before the writes, storing the intermediate results in temporary variables.

e) Benchmark all versions. Try various unroll lengths for the last two versions. Which one goes fastest?

f) (Advanced:) Can you generalize your code generator to arbitrary binary expressions? to any number of arguments and expressions?

Hint: It's a good idea to always check your answers against numpy.

Problem 2: Monte-Carlo

Consider a Monte Carlo simulation, where for each sample the following steps are taken:

1. Generate a vector x of random numbers on the CPU.
2. Transfer x to the GPU.
3. Compute $y = Ax + b$, for some matrix A and vector b.
4. Transfer y back to the CPU.

When finished, plot a histogram of distribution of 2-norms $\|y\|$.

A first, simple version of this code may be downloaded from `http://tiker.net/tmp/dtu-lab2-prob2.py`.

[1] `http://www.makotemplates.org/`

The objective is to make this code go fast. Here are some things to try, in order of increasing difficulty:

1. Insert (event-based) fine-grained timing code.
2. Overlap Host↔GPU transfers with computation.
    - Turn off profiling, use page-locked memory for actual overlap
      (At least on Nvidia)
3. Compute 2-norms on the GPU.
4. Generate random numbers on GPU.
5. Compute Ax for multiple x alongside each other

    Perhaps load (parts of) x into local memory.

After you've completed your first look at the code, try and answer these questions:

- Where is the most time being spent?
- Is the matrix-vector code compute- or memory-bound?
- Which code change will give the greatest performance win?