Assignment 0

Hardware Ray Tracing

Due: 11:59:59 PM, January 15, 2014

Getting started

This simple assignment is to set up and introduce you to the tools used in this class. It will guide you through downloading, installing, and running the TRaX simulator and compiler. **The instructions listed here are for the CADE linux machines or Mac OSX Mavericks. Windows is supported, but may require additional steps to configure. For Windows instructions, see:**
https://code.google.com/p/simtrax/wiki/Building

1. **Prerequisites**

   Before you start, you will need 2 third-party tools: ImageMagick and Clang. Most systems will already have Clang, and possibly even ImageMagick. To test if you have them, run "`which convert`" for ImageMagick, and "`which clang`" for Clang. If they both list a path to a binary, then you have them.

   ImageMagick: http://www.imagemagick.org/script/index.php

   Clang: http://llvm.org/releases/download.html

   For CADE machines: The default path on the CADE machines will try to use the wrong version of python, which is required to install TRaX. Your path must have `/usr/bin` listed BEFORE `/usr/local/bin`. To fix this for one session run the following command:

   ```
   set path = ( /usr/bin $path )
   ```

   Or to fix it permanently, add the above line to your `.tcshrc` file in your home directory. These instructions assume you are using tcsh. If you use a different shell environment, changing your path may require a different command.

2. **Download the TRaX framework**

   The TRaX simulator, compiler, and examples are stored in a google code repository. The project page is located at `https://code.google.com/p/simtrax/` The source is controlled using `svn`, and can be downloaded with the following Linux/Mac command:

   ```
   svn checkout http://simtrax.googlecode.com/svn/trunk/ simtrax
   ```

   This will download the project to a new folder called `simtrax`. Change to that directory, and you will see three folders:

   - `sim`: The TRaX cycle accurate simulator
   - `llvm_trax`: The TRaX compiler for writing our own C++ programs to execute on TRaX
   - `samples`: Various samples including example programs and ray tracing scenes

3. **Install the TRaX compiler**

   Since TRaX has a custom instruction set, we need a custom compiler if we want to write our own programs for it. The TRaX compiler supports C/C++ source code, and uses Clang and LLVM to generate a TRaX assembly file.

   Change directory to `simtrax/llvm_trax`, then run `./setup.sh`

   This command will unpack and install llvm-3.1, including our own added TRaX target. Be patient, as this will take 10-15 minutes.

4. **Install the simulator**

   Change to `simtrax/sim` and run `make`

   This will build the cycle accurate simulator and create an executable called `simtrax`.

5. **Compile and run an example**

   Change to `simtrax/samples/src/helloworld` and open `helloworld.cc`. This is the most basic example of a C/C++ program, with two key differences:

   - All TRaX programs must include `trax.hpp`. This includes the TRaX programming API.
   - Instead of `int main()`, TRaX programs must define `void trax_main()`. This ensures that various setup and cleanup routines needed by the simulator are invoked before and after your program executes.

   To compile this example, simply run `make`. This will generate several files, but the two we care about are:

   - `run_rt`: A Linux/OSX executable version of your program (functional simulator)
   - rt-llvm.s: The TRaX assembly file for running in the cycle-accurate simulator

   To run the CPU (functional simulator) version of your program, use the command:

   ```
   ./run_rt --no-scene --num-cores 1
   ```

   `--no-scene` indicates that this is not a ray tracer, and does not need to load a scene including a 3D model, camera information, etc...

   `num-cores 1` indicates we only need one core to run the program. The default is 4 cores, and we would see our "Hello world!" printed 4 times.

   This functional simulator is only useful for verifying that our programs produce the correct output, but does not simulate the TRaX architecture.

   To run your program on the real TRaX architecture, change directory to `simtrax/sim/` and run the following command:

   ```
   ./simtrax --no-scene --load-assembly ../samples/src/helloworld/rt-llvm.s
   ```

   This will print, along with "Hello world!" details about the execution of your program on the default single-core TRaX processor. We will study these details thoroughly later, and experiment with more interesting architectural features and programs.

6. **Graphical examples**

   Also included in `simtrax/samples/src/` are `gradient/` and `mandelbrot/`, more interesting examples which generate an image output (although they still aren't ray tracers). After running these programs, both the functional and cycle-accurate simulators will produce `out.png` containing an image. `gradient` simply assigns each pixel a color based on its x-y coordinates, and `mandelbrot` draws a Mandelbrot fractal. Compile and run both of these examples both in the functional and cycle-accurate simulators, just as with the `helloworld` example above. Be patient, the Mandelbrot program requires significantly more computation than the others, and will take a few minutes to run in the cycle-accurate simulator.

What to turn in:

This assignment is designed to make sure everyone can install and run the simulator, and serve as a basic tutorial on understanding TRaX output. By midnight on the due date, you should send e-mail to teach-cs6958@list.eng.utah.edu with the following information:

1. **URL:** A pointer to a web page containing the following information:
   (a) **Images:** .png images from the `gradient` and `mandelbrot` samples.
   (b) **Output:** The output of the cycle-accurate simulator (`simtrax`) for all 3 samples `helloworld`, `gradient`, and `mandelbrot`. The simplest way to get this output is to redirect the simulator's output to a file, for example:

   ```
   ./simtrax --no-scene --load-assembly ../samples/src/helloworld/rt-llvm.s > hello_output.txt
   ```

   (c) **Report:** Write a simple report analyzing the performance of the `gradient` and `mandelbrot` samples, addressing the following separately for both samples:
      i. How many cycles did each program take to complete?
      ii. How many instructions did each program execute?
      iii. If not equal, theorize one reason why the number of cycles would be higher than the number of instructions.
      iv. Assuming the algorithms are perfectly optimized, (the number of instructions required can't be reduced), without adding more cores, what aspect of the processor might you be able to change to increase the performance (reduce total cycles)?

   There is no need to analyze the performance of `helloworld`, since there is essentially no computation involved.