2.2. Setup and Walk-through

Contents

2.2.1. Profiling Tutorial

Profiling is the first step in making your programs faster. In this tutorial, we will learn how to measure latency and find the bottleneck in your program.

2.2.2. Obtaining and Running the Code

Login to Biglab and clone the `ese532_code` repository using the following command:

```
git clone https://github.com/icgrp/ese532_code.git
```

If you already have it cloned, pull in the latest changes using:

```
cd ese532_code/
git pull origin master
```

You will find the hw2 code in the `hw2` directory. The code for this tutorial is under `hw2/tutorial`. The code you will use for homework submission is under `hw2/assignment`.

Build and run the tutorial using:

```
cd ese532_code/hw2/tutorial
make
./rendering
```

2.2.3. Measuring Latency

You can measure latency in many different ways—instrumenting the code vs sampling-based profiling, using system timer vs using hardware timer etc. (review these slides and learn about clock sources here). However, the end goal is the same; which is to answer where is the bottleneck?

In this tutorial, we will show you how you can use the system timer to measure latency in seconds for parts of your program. We'll then demonstrate how you can use linux `gprof` to automatically instrument your code and get profiling information. Lastly, we'll show how you can use linux `perf` tool to get ***performance counter*** statistics of your program.

2.2.3.1. Instrumentation-based Profiling with Timers

In C++, you can use `std::chrono::high_resolution_clock::now()` from `<chrono>`. For example:

```
std::chrono::time_point<std::chrono::high_resolution_clock> start_time, end_time;
start_time = std::chrono::high_resolution_clock::now();
// code to measure
end_time = std::chrono::high_resolution_clock::now();
auto elapsed = std::chrono::duration_cast<std::chrono::nanoseconds>(end_time-start_time).count();
std::cout << "elapsed time: " << elapsed << " ns." << std::endl;
```

Note that we need nanoseconds resolution. Combining with a little bit of C++ syntax, we can create a class called `stopwatch` as follows:

```
#include <cstdint>
#include <chrono>

class stopwatch
{
  public:
    double total_time, calls;
    std::chrono::time_point<std::chrono::high_resolution_clock> start_time, end_time;
    stopwatch() : total_time(0), calls(0) {};

    inline void reset() {
      total_time = 0;
      calls = 0;
    }

    inline void start() {
      start_time = std::chrono::high_resolution_clock::now();
      calls++;
    };

    inline void stop() {
      end_time = std::chrono::high_resolution_clock::now();
      auto elapsed = std::chrono::duration_cast<std::chrono::nanoseconds>(end_time-
start_time).count();
      total_time += static_cast<double>(elapsed);
    };

    // return latency in ns
    inline double latency() {
      return total_time;
    };

    // return latency in ns
    inline double avg_latency() {
      return (total_time / calls);
    };
};
```

You can then use this class in `src/sw/rendering_sw.cpp` as follows:

```
#include <iostream>

// processing NUM_3D_TRI 3D triangles
stopwatch time_projection;
stopwatch time_rasterization1;
stopwatch time_rasterization2;
stopwatch time_zculling;
stopwatch time_coloringFB;
stopwatch total_time;

// processing NUM_3D_TRI 3D triangles
TRIANGLES: for (int i = 0; i < NUM_3D_TRI; i ++ )
{
  total_time.start();

  // five stages for processing each 3D triangle
  time_projection.start();
  projection( triangle_3ds[i], &triangle_2ds, angle );
  time_projection.stop();

  time_rasterization1.start();
  bool flag = rasterization1(triangle_2ds, max_min, max_index);
  time_rasterization1.stop();

  time_rasterization2.start();
  int size_fragment = rasterization2( flag, max_min, max_index, triangle_2ds, fragment );
  time_rasterization2.stop();

  time_zculling.start();
  int size_pixels = zculling( i, fragment, size_fragment, pixels);
  time_zculling.stop();

  time_coloringFB.start();
  coloringFB ( i, size_pixels, pixels, output);
  time_coloringFB.stop();

  total_time.stop();
}
std::cout << "Total latency of projection is: " << time_projection.latency() << " ns." << std::endl;
std::cout << "Total latency of rasterization1 is: " << time_rasterization1.latency() << " ns." <<
std::endl;
std::cout << "Total latency of rasterization2 is: " << time_rasterization2.latency() << " ns." <<
std::endl;
std::cout << "Total latency of zculling is: " << time_zculling.latency() << " ns." << std::endl;
std::cout << "Total latency of coloringFB is: " << time_coloringFB.latency() << " ns." << std::endl;
std::cout << "Total time taken by the loop is: " << total_time.latency() << " ns." << std::endl;
std::cout << "------------------------------------------------------------" << std::endl;
std::cout << "Average latency of projection per loop iteration is: " << time_projection.avg_latency()
<< " ns." << std::endl;
std::cout << "Average latency of rasterization1 per loop iteration is: " <<
time_rasterization1.avg_latency() << " ns." << std::endl;
std::cout << "Average latency of rasterization2 per loop iteration is: " <<
time_rasterization2.avg_latency() << " ns." << std::endl;
std::cout << "Average latency of zculling per loop iteration is: " << time_zculling.avg_latency() <<
" ns." << std::endl;
std::cout << "Average latency of coloringFB per loop iteration is: " << time_coloringFB.avg_latency()
<< " ns." << std::endl;
std::cout << "Average latency of each loop iteration is: " << total_time.avg_latency() << " ns." <<
std::endl;
```

Recompile the program using `make` and run `./rendering`. You should see results similar to the following:

```
[ec2-user@ip-172-31-40-51 hw2_profiling_tutorial]$ ./rendering
3D Rendering Application
Total latency of projection is: 154140 ns.
Total latency of rasterization1 is: 191497 ns.
Total latency of rasterization2 is: 1.5305e+06 ns.
Total latency of zculling is: 364093 ns.
Total latency of coloringFB is: 263400 ns.
Total time taken by the loop is: 3.36578e+06 ns.
------------------------------------------------------------
Average latency of projection per loop iteration is: 48.2895 ns.
Average latency of rasterization1 per loop iteration is: 59.9928 ns.
Average latency of rasterization2 per loop iteration is: 479.48 ns.
Average latency of zculling per loop iteration is: 114.064 ns.
Average latency of coloringFB per loop iteration is: 82.5188 ns.
Average latency of each loop iteration is: 1054.44 ns.
Writing output...
Check output.txt for a bunny!
```

> **Tip**
>
> Note that the print statements are carefully placed outside of the timing region. Print statements are generally very slow, and we do not want their time to contaminate our measurement.

From the results, we can see `rasterization2` has the highest latency and is a good candidate for optimization.

For our assignments, the `stopwatch` class we built here should suffice. If you would like to try out something fancier, check out https://github.com/google/benchmark!

2.2.3.2. Profiling using GNU gprof

`gprof` is a tool in the GNU toolchain that can automatically instrument every function in your code and give you the timing breakdown of your program.

To enable it, you'll have to compile your program first with the `-pg` flag. This instruments the functions in your code. Next, you'll run the program, which will output a file called `gmon.out`, which `gprof` uses to prepare the statistics. Finally, you would use `gprof -p executable-name` to see the statistics (`-p` option is for limiting the verbose output of `gprof`).

Run it on our program as follows:

```
make gprof
```

You'll see the following output:

```
g++ -pg -Wall -g -O2 -Wno-unused-label -I/src/sw/ -I/src/host/ -o rendering_instrumented
./src/host/3d_rendering_host.cpp ./src/host/utils.cpp ./src/host/check_result.cpp
./src/sw/rendering_sw.cpp
Executable rendering_instrumented compiled!
Running ./rendering_instrumented to get gmon.out for gprof...
3D Rendering Application
Writing output...
Check output.txt for a bunny!
Running gprof ./rendering_instrumented...
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 43.93      0.25     0.25 80438400     0.00     0.00  pixel_in_triangle(unsigned char, unsigned char,
Triangle_2D)
 26.36      0.40     0.15   319200     0.47     1.26  rasterization2(bool, unsigned char*, int*,
Triangle_2D, CandidatePixel*)
 22.84      0.53     0.13   319200     0.41     0.41  coloringFB(int, int, Pixel*, unsigned char (*)
[256])
  7.03      0.57     0.04   319200     0.13     0.13  zculling(int, CandidatePixel*, int, Pixel*)
  0.00      0.57     0.00   319200     0.00     0.00  projection(Triangle_3D, Triangle_2D*, int)
  0.00      0.57     0.00   319200     0.00     0.00  rasterization1(Triangle_2D, unsigned char*,
int*)
  0.00      0.57     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z13check_resultsPA256_h
  0.00      0.57     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z15check_clockwise11Triangle_2D
...
```

You can see that `gprof`'s result agrees with our results from the manual instrumentation with timers (note that `pixel_in_triangle` is used by `rasterization2`). You will notice that the code that `gprof` profiled looks like as follows:

```
TRIANGLES: for (int i = 0; i < NUM_3D_TRI; i ++ )
  {
    // five stages for processing each 3D triangle
    for(int i = 0; i < 100; i++) {
      projection( triangle_3ds[i], &triangle_2ds, angle );
    }

    for(int i = 0; i < 100; i++) {
      flag = rasterization1(triangle_2ds, max_min, max_index);
    }

    for(int i = 0; i < 100; i++) {
      size_fragment = rasterization2( flag, max_min, max_index, triangle_2ds, fragment );
    }

    for(int i = 0; i < 100; i++) {
      size_pixels = zculling( i, fragment, size_fragment, pixels);
    }

    for(int i = 0; i < 100; i++) {
      coloringFB ( i, size_pixels, pixels, output);
    }
  }
```

We ran each function 100 times because `gprof` samples a timing value every 0.01 seconds. Hence, if your function runs faster than this sampling period, `gprof`'s results will be inaccurate. You can find out more about this from here. Also, refer to the manual to find out about more command line options.

2.2.3.3. Performance Counter Statistics using Perf

> **Note**
>
> Biglab doesn't have perf. The following is for your reference and you can try it in your own machine, or when we provide you with an Ultra96 board.

ARM has a dedicated Performance Monitor Unit (PMU) that can give you the number of cycles your program takes to run (read more about PMU here). We can use `perf` to get the performance counter statistics of your program (read these slides to learn more about perf).

Run perf as follows (`make perf` in the supplied `Makefile`):

```
sudo perf stat ./rendering
```

You should see the following output:

```
[stahmed@macarena hw2_profiling_tutorial]$ make perf
g++ -DWITH_TIMER -Wall -g -O2 -Wno-unused-label -I/src/sw/ -I/src/host/ -o rendering
./src/host/3d_rendering_host.cpp ./src/host/utils.cpp ./src/host/check_result.cpp
./src/sw/rendering_sw.cpp
Executable rendering compiled!
Running perf stat...
3D Rendering Application
Total latency of projection is: 125316 ns.
Total latency of rasterization1 is: 136611 ns.
Total latency of rasterization2 is: 2.29206e+06 ns.
Total latency of zculling is: 244155 ns.
Total latency of coloringFB is: 146167 ns.
Total time taken by the loop is: 3.48606e+06 ns.
------------------------------------------------------------
Average latency of projection per loop iteration is: 39.2594 ns.
Average latency of rasterization1 per loop iteration is: 42.7979 ns.
Average latency of rasterization2 per loop iteration is: 718.065 ns.
Average latency of zculling per loop iteration is: 76.4897 ns.
Average latency of coloringFB per loop iteration is: 45.7917 ns.
Average latency of each loop iteration is: 1092.12 ns.
Writing output...
Check output.txt for a bunny!

 Performance counter stats for './rendering':

              6.36 msec task-clock                #    0.953 CPUs utilized
                 0      context-switches          #    0.000 K/sec
                 0      cpu-migrations            #    0.000 K/sec
               163      page-faults               #    0.026 M/sec
        17,322,039      cycles                    #    2.722 GHz
        39,090,159      instructions              #    2.26  insn per cycle
         5,358,366      branches                  #  842.109 M/sec
            56,019      branch-misses             #    1.05% of all branches

       0.006679970 seconds time elapsed

       0.006685000 seconds user
       0.000000000 seconds sys
```

From the above output, we can see that our program took $17,322,039$ cycles at 2.722 GHz. We can use these numbers to find the run time of our program, which is $17322039/2.722 \approx 6.36$ milli seconds which agrees with the 6.36 msec reported by perf too. Note that perf used the "task-clock" (system timer) to report the latency in seconds, and used the PMU counter to report the latency in cycles. The PMU counter runs at the same frequency as the cpu, which is 2.722 GHz, whereas the system timer runs at a much lower frequency (in the MHz range).

Now that we have shown you three approaches for measuring latency, a natural question is when do you use either of these methods?

- Use Instrumentation-based Profiling with Timers or Profiling using GNU gprof when you want to find individual latencies of your functions.
- Use Performance Counter Statistics using Perf when you just want to know the total latency (either in seconds or cycles) of your program. When you don't have `perf` available in your system, you can also run `time executable-name` to get the total time your program takes to run.

However, the above answer is too simple. The application we showed you is slow enough for `std::chrono` to measure accurately. When the resolution of your system timer is not fine-grained enough, or your function is too fast, you should measure the function for a longer period of time (see the spin loop section from here). Alternatively, that's where the PMU offers more accuracy. Since the PMU runs at the same frequency as the CPU, it can measure any function. However, you will have to isolate your functions and create separate programs to use the PMU through `perf`. There is no stopwatch-like user API for the PMU counter.

For our application above, we saw that the total runtime reported by task-clock and PMU counter doesn't differ. Hence, it doesn't matter which approach you use in this case. If you want to get the latencies of individual function in ***cycles*** instead, you can just use your measured time with the clock frequency to figure out the cycles. Alternatively you could get the fraction of time spent by your function and use the total number of cycles from `perf stat`.

By Department of Electrical and Systems Engineering