SIEMENS EDA

Edge Detect Lab #1 Profiling

Lab Workbook

Catapult 10.6a

SIEMENS

Support Center: support.sw.siemens.com
Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

Table of Contents

Before you begin Before you Begin

Do the following after logging into the AWS Virtual Machine (VM):

- Open a terminal by right-clicking on the desktop and select “Terminal”
- Enter “ls catapult_edge_detect”
    - If the folder is already there you have already copied the files over and can proceed to the rest of the lab
- Copy the lab files to the user home directory by entering the following at the terminal prompt:
    - cp /project/catapult_edge_detect_106.tar .
    - tar -xvf catapult_edge_detect_106.tar
- Source the setup script in the terminal
    - Type “source /project/setup.sh” and hit return

You are now ready to proceed with the lab exercises.

NOTE: *If you find yourself with a need to retrieve the original data for a lab, you can always download the lab data to another directory within your student directory*

Objectives

- Profile the Edge Detect algorithm running on a host processor and identify the best candidate function for hardware acceleration
- Run the Edge Detect algorithm in a RISC-V RTL simulation and compare the results against the profile results from the host

Host Based Estimate

Go to the directory for lab 1:

```
% cd to ~/catapult_edge_detect/system_designs/labs/lab1
```

In this directory there are three sub-directories: host_profile, embedded_profile and hardware_sources.

Change directories into the “host profile” directory. Here we will get an estimate of the performance of different parts of the edge detect algorithm by running it on a general-purpose computer.

```
% cd host_profile
```

Here you will find main_profile.cpp and Edge_Detect_Algorithm.h. These implement an algorithmic version of edge detect. No notion of hardware or architecture has been put into this implementation.

main_profile.cpp simply loads an image and calls the “run” method from the EdgeDetect_Algorithm class.

The edge detect algorithm has 3 main components: a vertical differential computation, a horizontal differential computation, and a magnitude angle computation. There are a number of ways to profile a program running on Linux. In this case we will make calls to the function times(). This is called before and after the section of code that we want to profile. This allows us to see the user and system CPU time associated with the code. Importantly, times does not record any time that the process was suspected by the operating system.

The Makefile will build and run the edge detection algorithm with profiling code embedded. Execute the make file:

```
% make
```

You should see output that looks like this:

```
% make
g++ -O0 -Wno-write-strings -o profile_main profile_main...
./profile_main ../../../Edge_Detect_Workshop/image/peop...
```

```
Loading Input File
Running
Run          User time: 16.99 System time:  0.04 Total time:
17.03
Finished
```

Your actual numbers will be different, but this shows the amount of CPU time used to compute the edges for 200 images. Multiple images are used, as the granularity of the times() call is too small for a single, or even a few, images.

This shows the time for the complete computation. We want to know how much time is taken up by each of the constituent functions. To do this we need to add timer calls around each step of the function.

Edit the file EdgeDetect_Algorithm.h. The "run" function is defined around line 50. Before the calls to vericalDerivative, horizontalDerivative, and magnitudeAngle, you will see a call to start_timer(). After the calls there is a call to end_timer().

```
44  //----------------------------------------------------------------------------
45  // Function: run
46  //   Top interface for data inout of class. Combines vertical and
47  //   horizontal derivative and magnitude/angle computation.
48  void run(unsigned char *dat_in,  // image data (streamed in by pixel)
49           double        *magn,    // magnitude output
50           double        *angle,   // angle output
51           unsigned int  imageWidth,
52           unsigned int  imageHeight)
53  {
54    // allocate buffers for image data
55    double *dy = (double *)malloc(imageHeight*imageWidth*sizeof(double));
56    double *dx = (double *)malloc(imageHeight*imageWidth*sizeof(double));
57
58    start_timer();
59    for (int i=0; i<200; i++) verticalDerivative(dat_in, dy, imageWidth, imageHeight);
60    for (int i=0; i<200; i++) horizontalDerivative(dat_in, dx, imageWidth, imageHeight);
61    for (int i=0; i<200; i++) magnitudeAngle(dx, dy, magn, angle, imageWidth, imageHeight);
62    end_timer("Run");
63
64    free(dy);
65    free(dx);
66  }
```

Modify the code to bracket each of the function calls with a start_timer call and an end_timer call. Put an appropriate descriptor as a string argument to the end_timer call. This will be printed along with the times.

```
//--------------------------------------------------------------------------
// Function: run
//   Top interface for data inout of class. Combines vertical and
//   horizontal derivative and magnitude/angle computation.
void run(unsigned char *dat_in,  // image data (streamed in by pixel)
         double        *magn,    // magnitude output
         double        *angle,   // angle output
         unsigned int  imageWidth,
         unsigned int  imageHeight)
{
  // allocate buffers for image data
  double *dy = (double *)malloc(imageHeight*imageWidth*sizeof(double));
  double *dx = (double *)malloc(imageHeight*imageWidth*sizeof(double));

  start_timer();
  for (int i=0; i<200; i++) verticalDerivative(dat_in, dy, imageWidth, imageHeight);
  end_timer("Vertical");

  start_timer();
  for (int i=0; i<200; i++) horizontalDerivative(dat_in, dx, imageWidth, imageHeight);
  end_timer("Horizontal");

  start_timer();
  for (int i=0; i<200; i++) magnitudeAngle(dx, dy, magn, angle, imageWidth, imageHeight);
  end_timer("Mag/Angle");

  free(dy);
  free(dx);
}
```

Rebuild and run the new program:

```
% make
```

The output should look something like this:

```
Make
g++ -O0 -Wno-write-strings -o profile_main profile_main.cpp...
./profile_main ../../../Edge_Detect_Workshop/image/people_g...
Loading Input File
Running
Vertical      User time:  2.38 System time:  0.01 Total time:  2.39
Horizontal    User time:  2.33 System time:  0.02 Total time:  2.35
Mag/Angle     User time: 12.27 System time:  0.03 Total time: 12.30
Finished
```

The horizontal differential calculation and the vertical differential take about the same amount of time. But the magnitude angle takes significantly more. In this case (your numbers will be slightly different) the magnitude angle computation is taking 72% of the compute time.

However, we are not fully optimizing the code, and this may have some impact on the distribution of the load.

Modify the Makefile to change the optimization level from 0 to 3. The optimization level is on line 5 of the makefile, the setting of the variable “CXX_FLAGS”. Change the “-O0” to “-O3”.

```
1
2  CATAPULT_HOME  ?= /wv/hlsb/CATAPULT/10.6a/PRODUCTION/aol/Mgc_home
3  EDGE_DETECT     = ../../../Edge_Detect_Workshop
4  CXX             = g++
5  CXX_FLAGS       = -O0 -Wno-write-strings
6  IMAGE           = $(EDGE_DETECT)/image/people_gray.bmp
7
8  all: profile
9
10 profile: profile_main $(IMAGE)
11         ./profile_main $(IMAGE)
12
```

Rebuild and rerun the program:

```
% make
```

The output should look something like this:

```
Make
g++ -O3 -Wno-write-strings -o profile_main profile_main.cpp...
./profile_main ../../../Edge_Detect_Workshop/image/people_g...
Loading Input File
Running
Vertical     User time:  0.24 System time:  0.01 Total time:  0.25
Horizontal   User time:  0.48 System time:  0.02 Total time:  0.50
Mag/Angle    User time:  8.68 System time:  0.02 Total time:  8.70
Finished
```

Some observations: The vertical differential calculation speed up by a factor of 10. The Horizontal differential speed up by a factor or 5. And the magnitude angle speed up by only about 40%. Simple algorithms tend to gain the most from compiler optimizations. The optimizer can speed up code by more than an order of magnitude. So always fully optimize and algorithm before considering moving it to hardware.

The magnitude angle computation is taking 92% of the compute time for the function. Making it an ideal candidate for acceleration.

Recall this is a measurement based on processing on an Intel core, with massive amounts of memory bandwidth and cache. And it gives us an estimate of what the load will look like on an embedded processor. In most cases the performance will be in the same order of magnitude, but an estimate from a different type of CPU should be considered a bit suspect. Much closer estimates can be obtained by running the profile on the same type of CPU and packaged in a similar configuration as the target system. This can be done using a development board, which are widely available. The ideal case would be to run on a simulation (or emulation, or FPGA prototype) of the target system. This would give us an exact measurement, not an estimate, of the processing time for any software.

Embedded Measurement

Change directories into the embedded_profile directory:

```
% cd ../embedded_profile
```

Here we have an RTL level implementation of the RISC-V processor/memory subsystem configured as the target system. Here we can make a measurement of the computational load and compare it with the estimates from the host run.

Build the software:

```
% cd sw.edge
% make
% cd ..
```

Build and execute the design:

```
% make
```

This will take a few minutes.

You should see the following output:

```
VSIM -work ./work -voptargs=+acc -L rocket_lib -do run...
testbench_opt
loading data...
Running...
sw execution time: 53650 clocks
Finished
```

Here the time to process part of one image is measured. A free running timer is in the design, and it can be read by software. It is 64 bits wide and counts the number of clocks since reset and will rollover on overflow.

To time any software execution, bracket it with reads from the timer and take the difference to determine the number of clocks elapsed.

The code of main.c in sw.edge is shown below measuring the time for the full algorithm:

```c
135 int main(int argument_count, char *argument_list[])
136 {
137   unsigned long         start;
138   unsigned long         end;
139   static unsigned char  data_in[IMAGE_SIZE];
140   static float          sw_data_out[2 * IMAGE_SIZE];
141
142   printf("loading data... \n");
143
144   load_data_array(data_in);
145
146   printf("Running... \n");
147
148   start = TIMER;
149   edge_detect_sw(data_in, sw_data_out);
150   end = TIMER;
151
152   printf("sw execution time: %d clocks \n", end-start);
153
154   printf("Finished \n");
155 }
156
```

At line 148 the timer in read, and at line 150 the timer is read again. Line 152 computes the difference.

Here is the function edge_detect_sw():

```c
void edge_detect_sw(unsigned char *data_in,     // image data (streamed in by pixel)
                    float         *data_out)    // magnitude and angle output
{
  // buffers for image data
  static float        dx[IMAGE_SIZE];
  static float        dy[IMAGE_SIZE];

  const int kernel[] = KERNEL;
  unsigned long start, end;

  verticalDerivativeSw(data_in, dy, kernel);
  horizontalDerivativeSw(data_in, dx, kernel);
  magnitudeAngleSw(dx, dy, data_out);
}
```

Modify the function to time each of the sub-functions:

```
void edge_detect_sw(unsigned char *data_in,     // image data (streamed in by pixel)
                    float         *data_out)    // magnitude and angle output
{
  // buffers for image data
  static float         dx[IMAGE_SIZE];
  static float         dy[IMAGE_SIZE];

  const int kernel[] = KERNEL;
  unsigned long start, end;

  start = TIMER;
  verticalDerivativeSw(data_in, dy, kernel);
  end = TIMER;
  printf("Vertical derivative clocks: %d \n", end-start);

  start = TIMER;
  horizontalDerivativeSw(data_in, dx, kernel);
  end = TIMER;
  printf("Horizontal derivative clocks: %d \n", end-start);

  start = TIMER;
  magnitudeAngleSw(dx, dy, data_out);
  end = TIMER;
  printf("Magnitude/angle clocks: %d \n", end-start);

}

```

Rebuild the software image:

```
% make
```

Then run the program on the simulated design:

```
% cd ..
% make
```

You should see an output like:

```
VSIM -work ./work -voptargs=+acc -L rocket_lib -do run...
testbench_opt
loading data...
Running...
Vertical derivative clocks: 3341
Horizontal derivative clocks: 2499
Magnitude/angle clocks: 47812
sw execution time: 54720 clocks
Finished
```

The Magnitude/angle computation is taking 89% of the total time in the edge detect algorithm. This is close to the estimate from the host runs in the first part of the lab, at 92%. Note that on an Intel processor the time for the horizontal derivative takes twice as long as the vertical derivative, but on the RISC-V core the horizontal derivative is 33% faster than the vertical derivative.

Profiling on a general-purpose computer will usually allow you to find the bottleneck but looking into the details of the smaller consumers may lead to some incorrect conclusions. It is better to generate the profile on the same type of CPU and systems where the code will ultimately run. This is best done on development boards that are widely available. Even better is to profile on a clock cycle accurate model (RTL) in simulation, emulation, or an FPGA prototype.

Bonus Question

What impact does the optimizer have on the Rocket Core profile? Perform a “make clean” followed by a “make DEBUG=1” command in the sw.edge directory. This will rebuild the software with -O0. Return to the embedded_profile directory and run the profile with the “make” command.