EE180 Lab #2

Coding for Efficiency

Due: Tuesday, Feb 1st, 2022

Introduction

This lab will illustrate how programming, compiler optimizations, data-level parallelism and multi-threading affect performance and energy efficiency on modern computer systems. Specifically, you will use your classroom knowledge to optimize an image-processing workload on a dual-core ARM processor, commonly used in mobile platforms. By the end of this assignment, you will know how your understanding of computer architecture can help you write highly efficient code.

Lab description

You will be optimizing an implementation of Sobel filter in C++ for an 800MHz ARM Cortex-A9 processor[1] on the Zedboard development board[2]. The Zedboard is powered by a dual-core ARM A9 processor, and contains a Xilinx FPGA on die. They are running an ARM-compatible variant of Ubuntu called Linaro. The Cortex-A9 processor is an extremely popular dual-core design for cellphones and portable computers.

The target system is an ARM A9 processor. While we work to set up the Zedboard on which you will compile and measure the performance of your code, you can test the functionality of your program on an x86 platform using a provided header that effectively converts arm-based vector instructions into their x86 equivalence. This is only for development/testing purposes; the final delivery will be entirely on ARM.

The Sobel filter computes an approximation of the gradient of the image density function and is commonly used to perform edge detection on images, the first step towards image recognition and other image processing tasks. Given a grayscale image, the filter performs a 2-D convolution, transforming each pixel into a weighted average of the 3x3 grid of pixels around it. For further background on Sobel filter, refer to this article[3].

This lab contains two parts. In the first part, you will improve the single-threaded performance of the Sobel filter. In the second part, you will build upon your single-threaded optimizations to use thread-level parallelism.

Starter files

The starter code can be downloaded from Canvas:

This directory contains the following files:

- Makefile
- main.cpp: implementation starting point that contains the main() function.
- pc.h, pc.cpp: functions to initialize and use performance counters. You do not need to edit these files. However, it is important that you read and understand how the code works.

1 http://www.arm.com/products/processors/cortex-a/cortex-a9.php

2 http://www.zedboard.org/

3 http://en.wikipedia.org/wiki/Sobel_operator

4 https://github.com/intel/ARM_NEON_2_x86_SSE

- sobel_st.cpp: single-threaded implementation that grabs input from the camera or file and processes the frames one after another in a loop.
- sobel_mt.cpp: skeleton code that implements same functionality as above, but with multiple threads.
- sobel_calc.cpp: the core kernels that perform the 2D convolution.
- sobel_alg.h: Header file containing function declarations.
- baxter.avi: The video you will apply the Sobel filter to.
- *NEON_2_SSE.h[4]: This is a header that converts Neon vector instructions into x86 vector instructions. You can include this file inside your .cpp file and write your program with neon vector functions and still be able to run on an x86 platform. This is only for development purposes.*

To complete this lab, you will end up editing at least the Makefile, sobel_calc.cpp and sobel_mt.cpp. However, you are completely free to edit any of the starter files if you deem it necessary to implement your optimizations. Nevertheless, you cannot skip functionality or change the definition of the problem.

Baseline implementation

The base code operates in a tight loop that captures a color image from the video camera (cvQueryFrame), turns it into a grayscale image (grayScale), applies the Sobel filter (sobelCalc), and displays its output (imshow). The code also collects basic statistics about performance and energy efficiency, calculates a breakdown of execution time across the functions discussed above, and tracks some lower level statistics such as instruction counts and cache misses. These measurements will help guide your optimizations.

Once you compile and run the initial code (see instructions below), you can run cat on the <st or mt>_perf.csv file to see the following report:

Percent of time per function

Capture, 3.03575%

Grayscale, 25.653%

Sobel, 69.8904%

Display, 1.42079%

Summary

Frames per second, 5.67269

Cycles per frame, 1.52732e+08

Energy per frames (mJ), 246.796

Total frames, 70

Hardware Stats (Cap + Gray + Sobel + Display)

Instructions per cycle, 0.723725

L1 misses per frame, 205815

L1 misses per instruction, 0.00186216

Instruction count per frame, 1.10525e+08

As you can tell, the number of frames per second (FPS) is quite low, well below the 30 FPS that we typically expect. You will optimize the code in two steps – first on a single core, and then on multi-cores.

Note: The performance report is only available when run on ARM core on Zedboard. It is not available if you run it on rice.

Background on Performance Counters

By default, the pc.h and pc.cpp functions initialize and collect performance counter statistics on the number of cycles, L1 cache misses, and instruction count taken by each block of code. These counters all operate using Linux's built-in perf interface and libpfm[4] to lookup the Linux encodings for each of these counters.

The case-insensitive "name" field for each of these counters can be used to lookup the counter, as seen in pc.cpp. Note that not all of these counters work with the perf interface for our Linux image on the Zedboard.

Here are some general guidelines on using the performance counter data:

1. The "instructions per cycle (IPC)" statistics refer to 1/CPI. The higher the IPC, the better.

2. The "instruction count per frame" should be as low as possible.

3. The two cache measurements tell you about data locality (and lack thereof), and points to memory system performance. You will learn more about caches later in the class. In general, as you optimize the program these two numbers will go down.

Part 1: Single-threaded Sobel filter

In this part, you will optimize single-threaded performance. This involves changing the C code and compilation flags in the Makefile. You will also exploit data-parallelism through vector instructions to improve efficiency. You are allowed any optimization you want within function runSobelST(), sobelCalc() and/or grayScale() in order to optimize the execution of grayscale conversion and the Sobel filter itself. You are also allowed to edit the compilation flags in the Makefile. Consider how the structure of the C code affects the number of instructions, the cycles per instruction, and the number of cache misses in the execution of the program. Restructure and refactor

the C code as needed to optimize the execution time and energy efficiency of your code. Do not change the code for capturing input images or for displaying the image output.

The Neon vector instructions for the ARM instruction set allow you to operate on a short vector of 4 numbers with a single instruction, achieving up to a 4x improvement for the portions of the code you vectorize. However, you need to change compilation options and potentially your C code to maximize the effectiveness of vectorization on grayscale conversion and the Sobel filter. Make any such optimizations needed.

Background on Vectorization with Neon

The Cortex-A9 core is each equipped with Neon[5] vectorization units. These vector units are capable of operating on registers that are 128-bits wide and can perform 16 8-bit operations, or 8 16-bit operations, or 4 32-bit operations at a time. Vector units, also known as SIMD (single instruction, multiple data) operate by running the same instruction (add, subtract, etc.) on multiple different data elements, and, therefore, are particularly useful for computing multiple pixels in parallel.

Note: If you use the header file and test it on an x86 machine. Some of the data type declarations may not be available with the x86 compiler. You may have to do tricks to initialize variables in ways different from the arm version to achieve the same thing.

---

4 http://perfmon2.sourceforge.net/

5 http://arm.com/products/processors/technologies/neon.php

In general there are three techniques that can be used to take advantage of the vector units.

1. The first technique involves restructuring the code and changing the compiler flags to have the compiler automatically vectorize the code. Given the right flags, the compiler can take nearby, non-dependent operations and infer that they can be safely parallelized. The compiler will automatically pack these operations into vectors and compute them with the Neon vector engine.

2. The second method involves something called "intrinsics". Neon intrinsics[6] are code constructs designed to express the vector/parallel relationship among different operands. These constructs give the compiler a large amount of guidance about what can be safely parallelized, and therefore produce more efficient results than code that does not use intrinsics.

3. To achieve best results, the programmer can manually program the SIMD unit using assembly language. Since the programmer has high-level knowledge about what the program's using, he or she knows exactly what can and cannot be vectorized. Therefore, the programmer can write assembly code to maximize utilization of the SIMD unit. Assembly code can be written alongside C code using the compiler's "inline assembly" support.

NOTE: Students only need to make sure that the key loops are vectorized. Any of the techniques outlined above can be used to achieve this result.

Part 2: Multi-threaded Sobel filter

The Zedboards used in this lab contain two ARM Cortex-A9 cores. In this part, you will be editing sobel_mt.cpp to use the two cores. If you use the two cores successfully, you will get up to a 2x performance improvement on top of your part 1 performance.

You will rewrite the runSobelMT() method in sobel_mt.cpp. You will build on the code from part 1 in sobel_calc.cpp and rewriting the runSobelMT() method to support multiple threads. To run the given code in a multi-threaded setting, you need to pass a "-m" option to the executable during runtime. Failing to provide the '-m' option will result in running the single-threaded version. The "main()" function we have provided you launches two threads, where each thread begins execution of the runSobelMT() function. The main thread then calls "join" to wait for both the launched threads to terminate.

Do not launch more than two threads for this part. One of the perils of multi-threaded programming is intermittent errors, infamously known as "Heisenbugs[7]". Does your bug "go away" if you add a single printf() somewhere? You most likely have a race condition or a synchronization issue in your multi-threaded code. Your submitted code should run deterministically, even when run multiple times. Locks and barriers have to be used to synchronize the threads correctly. Please refer to the section titled "Background on Threading" for a deeper overview on these primitives. Once you get it right, you will end up with the highest performing code at the end of this part.

---

6 https://gcc.gnu.org/onlinedocs/gcc-4.9.1/gcc/ARM-NEON-Intrinsics.html

7 http://en.wikipedia.org/wiki/Heisenbug

Background on Threading

The processor on the Zed board contains two Cortex-A9 cores that access the same, shared memory. In order to take advantage of both cores in a single program, the program must be written to use independent threads of control that can be executed in parallel. Informally, running two threads is like running two programs with separate program counters (PC) and registers but with shared memory. The two threads can communicate and synchronize through shared memory, often using atomic instructions. Nevertheless, you will program at a higher level and use library abstractions for synchronization.

In Linux, multithreaded programming for C and C++ programs are often done using either the pthread or boost libraries. For our program, we're relying on the pthreads library[8]. Your program is already using one thread when it starts running. Two additional threads are launched using the command:

```
pthread_t sobel1, sobel2;

pthread_create( &sobel1, NULL, runSobelMT, NULL);

pthread_create( &sobel2, NULL, runSobelMT, NULL);
```

Once launched, they will run independently until finished. The main thread will also continue to run independently until it reaches the "join" command, at which time it stalls until the launched threads complete and join the main program.

The two launched threads will run the runSobelMT() function. For the most part, this function implements the math for Sobel filter as we know it. However, it is often necessary to for data and control flow information to pass between threads and for threads to synchronize with each other. Data communication between threads can be easily done through main memory (read and write variables and data structures). However, you need to be careful about synchronization. For example, if we try to communicate data before it is ready, a thread may read the wrong values or some writes may be lost. There are two ways to synchronize threads:

• Barriers (pthread_barrier_wait): When code in a thread reaches a call to the barrier function, it stalls until the other thread also reaches this barrier, synchronizing the threads. In other words, we know that the two threads are at the same point of execution.

• Mutexes (pthread_mutex_lock, pthread_mutex_unlock): Mutexes can be used to implement mutual exclusion. For instance, you can use a mutex to lock a variable so only a single thread can access it (or update it) at the time. Mutexes are particularly useful when multiple threads content for a shared variable such as a shared counter or for a shared data-structure such as a work queue. They also allow for synchronization, without forcing the two threads to be in the same point of execution.

You have to device a synchronization scheme between the two threads. This scheme will be quite simple though.

---

8 http://pages.cs.wisc.edu/~travitch/pthreads_primer.html

In general, threads are best if they can be launched once and re-used, since there is overhead associated with parceling out different threads to different cores. Therefore, for this project we restrict you to using the provided "main()" function in sobel.cpp that calls two instances of runSobelMT(), where those threads are joined only when the user has indicated that they wish to quit the application.

Using the ZedBoard

This project will be completed on a Xilinx Zedboard provided for you in the class lab. You will be logging into the board using ssh. This is the only way in which you will be connecting to the Zedboard. In order to successfully run your program and view the output image, you must enable X-forwarding in your ssh session using the '-Y' option (DO NOT use '-X'). Each group will be assigned a particular board. We will make an announcement when the boards are ready and you will be receiving your log-in credentials on gradescope if you have previously submitted to the group list assignment. The boards can be logged into remotely. You need to be within the Stanford network in order the access the Zedboards.

For students on campus, you can log into your assigned board as follows:

ssh –YC <provided_login>@<boardname>.stanford.edu

For students off-campus, you must log into one of the myth (myth.stanford.edu) or rice (rice.stanford.edu) machines first. For example:

ssh –YC SUNET_ID@myth.stanford.edu

ssh –YC <provided_login>@<boardname>.stanford.edu

Alternatively, you can VPN into the Stanford network by following the instructions at https://uit.stanford.edu/service/vpn for your particular system.

PC users, NOTE: For Windows, you need an ssh client like putty to connect to the Zedboard and enable X-forwarding. Alternatively, you can use a Linux virtual machine[9] and follow the steps above. Another option is to use an Xserver or logging into myth.stanford.edu or rice.stanford.edu through VNC and then using the ssh method to connect to your zed board from within the VNC window. Full information and instructions about the Xserver[10] and VNC11 options are available from the included links.

The starter code can be downloaded from Canvas.

Once the boards are ready, you will need to copy the code over to the Zedboard in order to compile and run it. We are not providing cross compilation support currently, so you have to build your code on the Zedboard only when you run your program on the Zedbaord. To build and run the starter code:

1. Download the starter code and baxter.avi on your assigned Zedboard using scp or wget.

2. Log into the board using ssh -Y, your login ID and board ID.

3. Compile the starter code by typing 'make'. This produces the binary 'sobel'.

4. Run the binary:

Single threaded: ./sobel –n <#frames>

Multi-threaded: ./sobel –m –n <#frames>

The program will automatically stop after processing the specified number of frames from the input. To exit out of the program before the specified frame count is processed, select the Sobel image window so that it is in focus, and press and hold "q" until the window disappears.

TIP: A complete list of supported command-line options can be seen by executing the sobel executable without any options.

TIP: With the exception of reading the performance counters, all of the code provided for this assignment should be x86 compatible. If you'd like to develop, test, and debug the code on your own Linux machine, simply install OpenCV on your machine and use the header provided for neon vector instructions. The grading, however, will be done using Zedboards.

Recommended work-flow:

- Start early
- Do NOT develop your code solely on the Zedboard without a backup
- Read the previous line again – backup your code regularly. Use the Zedboard to only build and test your code.
- Develop your assignments using version control. Use a private bitbucket[12] or github[13] repository to store your work in the cloud.
- Test thoroughly

IMPORTANT: Did we mention that you must backup your code? Under certain circumstances, we might have to reset the Zedboards. When this happens, we are forced to reformat the boards, and all data stored on the board is lost. **No extensions will be given for code eaten by the Zedboard.**

Submission Instructions:

You must create your submission tarball by typing make submit.

You have to submit all the tarballs related to your lab electronically. Please name your tarball as follows:

lab2.<group_no>.tar.gz (Example: If you are group 40, name your tarball lab2.40.tar.gz)

The README file should contain your name, your partner's name, your board's host name (e.g. ee180-40z.stanford.edu), and a description of what you did for that part of the assignment. This README should be a short report that explains optimizations you tried, the outcome, and some explanation (why it worked, why it did not). Please report the speedup for the whole program.

Please submit your tarball on GradeScope under the Lab 2 assignment. Be sure to submit as a group (you can add two people to one submission on GradeScope).

BE SURE TO ADD ALL MEMBERS TO ONE SINGLE SUBMISSION (not 2). Members that are not added to submission will not get a score.

---

9 32 bit Ubuntu 14.04 LTS on Virtual Box (https://www.virtualbox.org/) is a good free option

10 https://itservices.stanford.edu/service/sharedcomputing/moreX

11 https://itservices.stanford.edu/service/sharedcomputing/vnc

12 https://bitbucket.org/

13 https://education.github.com/