Computer Architecture 2020 Lab 1

Assembly Lab

Table of Contents

Update: 20200305

Update: 20200303

Overview

Convolution is an important primitive widely used in many applications including computer vision (CV), natural language processing (NLP) and signal processing. One of the most popular algorithms is convolutional neural networks (CNNs). In this lab, we will implement in the MIPS assembly language a simple convolution that uses a Gaussian-like filter, and optimize its performance using a cycle-accurate CPU emulator.

Before starting this lab, you are supposed to be equipped with the following:

- Access to a Linux server (suggested approach, set at lab0)
- Basic Linux operation (refer to Linux cheat sheet)
- Basic C programming language
- Basic MIPS assembly

After this lab, you will understand the following:

- Conception of image processing, particularly smooth filter
- Conception of emulating, especially SPIM, a MIPS emulator
- Conception of code optimization and simple performance statistical analysis of your own program

During this lab, you are suggested to refer to the following materials:

- Linux cheat sheet
- http://cheatsheetworld.com/programming/unix-linux-cheat-sheet/
- Image Convolution & Gauss Smooth Filter
- https://en.wikipedia.org/wiki/Kernel_(image_processing)
- https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm
- MIPS32 ISA Quick Reference
- https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00565-2B-MIPS32-QRC-01.01.pdf
- Detailed documents: https://www.mips.com/products/architectures/mips32-2/
- Code optimization
- https://en.wikipedia.org/wiki/Loop_nest_optimization
- https://www.agner.org/optimize/optimizing_assembly.pdf
- Syscalls defined by SPIM
- http://students.cs.tamu.edu/tanzir/csce350/reference/syscalls.html

Provided Infrastructure

Inside this repository, we provide the following files:

```
+-- lab1-handout.pdf --- This document
+-- run --- Our workspace folder
|   +-- gen_pixel.py --- Convert an image into pixel file
|   +-- parse_pixel.py --- Convert a pixel file into image
|   +-- mystdio.h --- IO header for our MIPS emulator. Include it when
SPIM IO is in need
|   +-- lab1-mips-startercode.S --- Starter code for MIPS assembly
|   +-- lab1-c-startercode.c --- Starter code for C
|   +-- simple_add.c --- sample program
+-- sample --- Origin image and expected blurred results
+-- submit --- Your final works to submit
```

Tasks

1 Setup Environment

```
$ mkdir ~/yao-archlab-s20 && cd "$_"
```

Upload the lab 1 package to your server:

- For MobaXterm users, use the sidebar button.
- For other users, use `scp` tool on your local host: `scp -P <machineID> lab1.zip <studentID>@<IPaddr>:/home/<studentID>/yao-archlab-s20`

```
$ unzip lab1.zip -d ~/yao-archlab-s20
```

```
$ git clone https://github.com/leepoly/spim-timingmodel
```

```
$ git -C ./spim-timingmodel/ checkout lab1-v1.0
```

```
$ cd ./lab1
```

```
$ export LAB1=$HOME/yao-archlab-s20/lab1
```

Note: Redoing `export` is needed every time you reconnect to the server. Or you can put it in your `.bashrc` file.

```
$ wget http://ftp.loongnix.org/toolchain/gcc/release/mips-loongson-
gcc7.3-2019.06-29-linux-gnu.tar.gz
```

```
$ tar xvf mips-loongson-gcc7.3-2019.06-29-linux-gnu.tar.gz # unzip mips
cross-compiler
```

```
$ ln -sf $LAB1/mips-loongson-gcc7.3-linux-gnu/2019.06-29/bin/mips-
linux-gnu-gcc ./run/ # create a shortcut for compiler
```

```
$ make -C../spim-timingmodel/spim # build original SPIM
```

```
$ make install -C../spim-timingmodel/spim
```

(Updated) Note: Your lab directory structure should be the following:

```
+-- ~/yao-archlab-s20
|   +-- spim-timingmodel
|       +-- spim
|       +-- CPU
|       +-- …
|   +-- lab1
|       +-- run
|       +-- sample
|       +-- submit
|       +-- mips-loongson-gcc7.3-2019.06-29-linux-gnu
```

Note: these steps may not work at the first time, especially on other Linux environments (e.g., your personal laptop). Be aware of the output and check if any error occurs, which is usually because of lack of necessary dependencies. Contact me if you don't know how to handle those dependencies. It is okay if there are only compiling warnings.

```
$ ln -sf $LAB1/../spim-timingmodel/spim/spim ./run/ # create a shortcut for emulator
```

Now you have setup the lab 1 environment successfully.

Now test a simple `add` program:

```
$ cd ./run
```

```
$ ./compile.sh simple_add # compile C file into MIPS assembly code
```

```
$ ./spim file simple_add.S # emulate this assembly
```

type any two integers, split by `<space>`, for example:

```
3 4 <enter>
```

You will see `7` on the screen, along with instruction and cycle statistics of that code segment.

2 Generate Pixel File

To make it simpler to work on the image data, we will extract only the pixels from a `.jpg` image into a pixel file.

```
$ python3 gen_pixel.py gauss.jpg gauss.pixel
```

Note: If you encounter errors with lib missing, you can either install it or just use the pixel file `gauss.pixel` provided in `sample/` folder. For example, when you see the error `ImportError: No module named Image,` do:

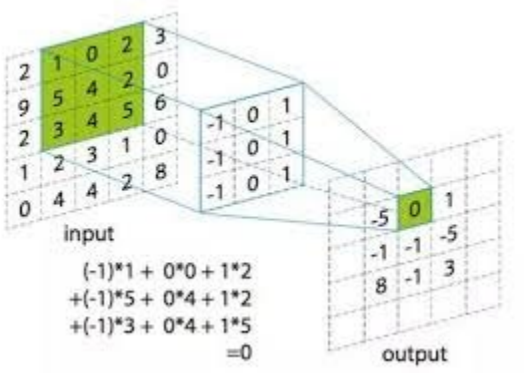
`$ python3 -m pip install pillow numpy --user` first, and then try the above command again.

Note: `pixel file` format description:

```
N M : for first line, height and width of image
p(i,j) ... : for the next N lines, M numbers each line, p(i, j)
representing grayscale of each pixel: 0-255
```

Check if the output file meets this format.

3 Write Gaussian-like Filter in C Programming Language



There are many existing algorithms for image processing. Most of them apply convolution operation to image. We propose a simple gauss-like image filter which has a 3 * 3 kernel matrix K:

```
1 2 1
2 4 2
1 2 1
```

We have provided you a basic framework to start with: `lab1-c-startercode.c`. In your C implementation, you need to implement two critical functions marked as `TODO`s:

- `ImageInput`: We have provided the code to read `n` and `m` from screen, which are the height and width of an image. You should read the entire remaining pixels into `img` from screen. Assume the screen input follows the format of the pixel file defined above.
- `ImageProcess`: For each pixel P in `img`, we define its 8 neighboring pixels, as well as itself, as the 3 * 3 neighboring matrix `N(P)`. Then we do elementwise-multiplication of `N(P)` and the kernel `K`. Finally we accumulate all the values and normalize it by dividing with 16 and round it down, to generate the new pixel `P'` of image. This entire image get blurred because the grayscale of each pixel has "diluted" to its neighboring pixels. **We**

leave the image border (pixels in first and last row/column) to be unchanged. Therefore, the size of the output image should be the same with the input.

The processing formula is like this:

$$O[x,y] = \begin{cases} \lfloor \frac{1}{16} \sum_{i=0}^{2} \sum_{j=0}^{2} I[x+i-1, y+j-1] * K[i,j] \rfloor, \forall x,y > 0 \wedge x < N-1 \wedge y < M-1 \\ I[x,y], \forall x = 0 \vee x = N-1 \\ I[x,y], \forall y = 0 \vee y = M-1 \end{cases}$$

Assume your final C program is named `lab1-c.c`. To compile and run your program:

```
$ ./compile.sh lab1-c
```

```
$ ./spim -lab1-dev file lab1-c.S
```

Input `.pixel` format to screen, just as you did in `simple_add.S`. Your program should return the processed `.pixel` format to screen.

This lab supports two modes of spim-timingmodel (SPIM-T). In lab1-developing mode, execution trace is demonstrated. `stdin/stdout` is used. You can test it with small input cases. In release mode, execution trace is hidden and all IOs are directed from/to specified files: `./spim -lab1-rel inputfile outputfile file lab1-c.S`. Make sure you have a valid input file that exists (for example, `gauss.pixel`)

Hint: This seems cool but what is SPIM-T?

SPIM is a classical emulator, where your MIPS assembly get emulated on a x86 machine, instead of being executed by a real MIPS CPU which is uncommon today. SPIM uses a bunch of `int32` to imitate the functionality of CPU registers. Then SPIM uses `switch...case...` structure to imitate the CPU decoding. Thus SPIM can only emulate the functionality of a program (with no cycle information). In this lab, we use a naive counter on SPIM for timing statistics, thus called SPIM-T. In the next lab, we will learn more about timing emulation within the CPU.

Hint: You can literally view the processed image with your outputfile named `gauss-c.pixel`! Simply type `python3 parse_pixel.py outputfile`.

Important Question: My C program works fine on x86. Why I can't emulate it using SPIM?

- SPIM emulator provides its own interface for input/output. We should use the IO syscall that SPIM defines. For C programs, TA has prepared a simple wrapper called `mystdio.h`. Check the code for its usage!

Modify your code by:

- Adding a header `+ #include "mystdio.h"`

- Removing standard IO header: `- #include <stdio.h>`
- Removing all file-related IO functions: `fopen`, `fclose`, ...
- Directing all read and write to `stdio` and identify their data type: `fscanf`/`scanf` -> `_scanf_num`, `fprintf`/`printf` -> `_printf_char`

Also, there is some inconsistency between MIPS cross compiler and SPIM emulator. Correct your code by:

- Do NOT turn on the `-Ox` optimization switch. Use `-O0` as `compile.sh` does.
- Using global variables for large memory allocation. Set an initial value for every global variable.
- **In MIPS file, making sure all global data segments have `.data` header instead of `.text` header.**
- Contact TA if you find other errors that you do not know how to fix.

4 Write Gaussian-like Filter in MIPS

Sometimes the compiler does the code translation in an ineffective way. That is the time when we write assembly language directly. Let's see which version performs better.

Hint: Again, we provide a basic framework to start with. See `lab1-mips-startercode.S`. Complete all `TODO`s. Finish your implementation and name the file as `lab1-mips.S`.

Hint: you should keep the results consistent between your C and MIPS implementations. Otherwise there must be a bug in either of them. That is the reason why you should start with simpler C programming.

We do not need to compile a `.S` file. Just emulate it:

```
./spim -lab1-dev file lab1-mips.S
```

```
./spim -lab1-rel inputfile outputfile file lab1-mips.S
```

Take a look at the instruction and cycle statistics given by SPIM-T. Is your MIPS implementation better than the C version?

5 Optimize MIPS

You may (or may not) see the MIPS implementation outperforms the C implementation. But in any case, let's not just call it a day here. We can do better with MIPS! Your next

task is to optimize the MIPS implementation. The faster, the better. Name the optimized implementation as `lab1-mips-opt.S`.

Hint: Some tips for code optimization that you may find useful:

- Use shift instructions (e.g., `sra`, `sll`) instead of multiply/division. Notice the kernel elements are all the powers of 2.
- Use more compact instructions (e.g., `movz`, etc.). To see the full list of SPIM supported MIPS instruction, refer to `yytokentype` struct in `$LAB1/spim-timingmodel/spim/parser_yacc.h`, or the decoding part code in `$LAB1/spim-timingmodel/CPU/run.cpp`
- You may notice large overhead is from memory access. Try to avoid it by reusing as many registers as possible for kernel access.
- Another way to optimize memory access is through loop optimization techniques, such as loop blocking. For more information, refer to the wiki page at the top.

Write-up

You should write a design document for lab 1. There is no specific format required, but you should demonstrate how your program works, in a clear way. You should at least include the following parts:

Methodology

How you finished your labs. Show your debugging process if there is anything interesting.

Summarize the optimizations you have used to improve the performance of the MIPS implementation. For each optimization, include a short summary of the code changes, which one(s) in instruction count and/or CPI it reduces and why, how much percentage reduction in the number of cycles it results in.

Results

In your design document, use the table below to record the performance of your programs: 1) the C implementation; 2) the unoptimized MIPS implementation; 3) the optimized, final MIPS implementation.

Instruction Type	Total Instructions	Total Cycles	IPC
branch inst.			

memory inst.			
register inst.			
total			

Question Answering

- Why is your MIPS program better/worse than your C version? Provide 3 reasons at least. Use the differences between the two assembly implementations to explain.
- What are the limitations still in your optimized code? Any ideas on how to overcome them? Describe the ideas and no need to implement.

Submission

Create a folder with the name of `lab1-<student ID>-<first name>-<last name>`, e.g., `lab1-2018011222-xiaoming-wang`. Put the following files in it. Compress the folder as a `.zip` package and upload to **learn.tsinghua.edu.cn** (网络学堂) by **March 27**. You may submit for multiple times, and we will grade based on the latest submission.

```
+-- lab1-designdocument.pdf --- Your lab1 design document
+-- lab1-c.c --- C version of your program
+-- lab1-mips.S --- unoptimized MIPS version of your program
+-- lab1-mips-opt.S --- optimized MIPS version of your program
+-- mystdio.h --- Include this header only if you have modified it
+-- gauss-processed.pixel --- gaussian-processed gauss pixel file
```

Grading Policy

Plagiarism is **strictly** forbidden in this lab. Peer discussion is **not** suggested. Contact TA if you are in trouble.

- 20% - Correct C implementation.
- 20% - Correct unoptimized MIPS implementation.
- 20% - Correct optimized MIPS implementation.
- 20% - Design document: the description of the optimization methods used and why they help improve the performance.
- 10% - Design document: question answering
- 5% - Design document: others.

- 5% - Performance of the optimized MIPS implementation. Score by ranking.

Correctness

We will emulate your program to generate an output pixel file, and then compare it with a pre-processed ground truth. More matches mean higher correctness score. (Update: similar but not identical outputs can still earn some points) Notice that **we will use different input images for grading** (update: the size of input image is no more than (500, 500))! So you should thoroughly test your programs before submitting.

Performance

Measured by the total cycles of your programs from the output of SPIM-T.