Lab1 Assembly Lab

Overview of this lab

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

- A laptop with native Linux OS/WSL-based Win10 (refer to WSL setup instructions)/Linux VPS access
- Basic Linux operation (refer to Linux cheetsheet)
- Basic C programming language
- Basic MIPS assembly

After this lab, you will understand following:

- Conception of image processing, particularly smooth filter
- Conception of cross-compiling and emulating, especially SPIM, a MIPS emulator
- Conception of several timing models, particularly simple 5-stage pipelining CPU model
- 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 cheetsheet http://cheatsheetworld.com/programming/unix-linux-cheat-sheet/
- WSL setup instructions https://docs.microsoft.com/en-us/windows/wsl/install-win10
- Image Convolution & Gauss Smooth Filter https://en.wikipedia.org/wiki/Kernel_(image_processing) https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm
- 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

Task Requirement

Given a original image `gauss.jpg`, try a gaussian-like smoothing to generate a blurred image `blurred-gauss.jpg`. Implement a **C version** and a **MIPS assembly version** to achieve this. Test the correctness and compare the performance of both your programs, under different timing models.

Provided infrastructure

Inside this repository, we provide following files:

```
|-lab1-handout.pdf --- This document

|-spim-timingmodel --- SPIM emulator

    |-CPU --- Critical spim code

    |-spim --- folder for spim.c & spim ELF

|-run --- Our workspace folder

|-gen_pixel.py --- Convert an image into pixel file

    |-parse_pixel.py --- Convert a pixelfile 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
```

Task Instructions

Figure 1 introduces the work flow in lab1. [Blank]

Figure 2 shows our three timing models. [Blank]

0. Set up whole environment. `git clone https://github.com/tsinghua-ideal/yao-arch/`

   `cd lab1`

   `export LAB1=$PWD`

   `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 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$LAB1/spim-timingmodel/spim # build original SPIM`

   [note] This step may not work first time. Be aware of any output (check if error occurs), which is usually because of lack of necessary dependencies. Contact me if you don't know how to install those dependencies.

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

Now you have setup lab1 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 statistics of that code segment.

1. Generate pixel file

   Just type:

   `python gen_pixel.py gauss.jpg gauss.pixel`

   [note] If you encounter error with lib missing, you can either install it or just use pixel file `gauss.pixel` provided in sample/ folder.

   [note] The `pixelfile` format:

   N M # for first line, height and width of image

   pi ... # for next N lines, M numbers each line, representing grayscale of pixel: 0-255

2. Write gauss-like filter

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

   1 2 1

   2 4 2

   1 2 1

   Apply convolution (element-wise multiplication) of K and each pixel to a new P', normalize the result to 1/16 and make sure the value of P falls in [0, 255].

   $P' = max(min(\frac{1}{16}(P * K)(1,1), 255), 0), \forall P \in img$

   Assume your `lab1-c.c`, `lab1-mips.S` and their output `gauss-c.pixel`, `gauss-mips.pixel` respectively. [hint] You may compile it use your native gcc compiler. For checking its correctness, `python parse_pixel.py gauss-c.pixel`

   `./compile.sh lab1-c`

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

   For simplicity, TA provides a simple MIPS and C starter file. Fill in all TODOs.

   [Question] My C program works fine on x86. Why I can't emulate it under MIPS?

- SPIM emulator provides its own interface for input/output. We should use the IO syscall that SPIM defines. For C program I have written a simple header 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`

- There is some inconsistency between MIPS cross compiler and SPIM emulator. Correct your code by:
    - Use global variable for large memory allocation. Set initial value for every global variable.
    - In MIPS file, make sure all global data segment have `.data` header.
    - Contact TA if you find more errors.

3. Emulate and optimize your program

   `./spim -lab1-rel gauss.pixel gauss-c.pixel file lab1-c.S` Try to eliminate unnecessary cycles. You may notice large overhead is from memory access. Try to avoid it by reusing as many registers as possible for kernel access.

   For common code optimization:

- Use shifting instruction (`sra`, `sll`) instead of actual multiply/division.
- Use one instruction with assertion instead of multiple instructions: `movz`, etc.
- Use inline function to alleviate ABI overhead.
- One of a common loop optimization technique is called loop blocking. For more information, refer to the wiki at the top.

  Typical unoptimized program results:

  C version

  statistics of instructions

  branch inst. #3130569 estimated cycle 8918748

  memory inst. #30298157 estimated cycle 3029815700

  register inst. #76035488 estimated cycle 380177440

  total #109464214 estimated cycle 3418911888

  MIPS version branch inst. #2815957 estimated cycle 7974914

  memory inst. #3277189 estimated cycle 327718900

  register inst. #16564604 estimated cycle 82823020

  total #22657750 estimated cycle 418516834

4. Finish writing your design document and following documents

- Design document

No specific format required, but you should demonstrate how your program works, in a clear way

- Record performance of your algorithm

Timing Module	Instruction Count	Cycle Count
Multicycle non-pipelining core		
Multicycle pipelining core, without bypass		
Multicycle pipelining core, with bypass		

- Answer following question
  - Why is your MIPS program better/worse than your C version?
  - Describe the code optimizations you have used to improve the performance of your implementation. Explain why it helps increase the IPC.
  - What are the limitations still in your submitted optimized code? Any ideas on how to overcome them?

Grading policy

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

Correctness

TA emulates your program to generate a output pixel file. Then TA compares it with a pre-processed golden pixel file. More matches mean higher correctness score. Note that we will use different input image pairs for grading!

Performance

Output of SPIM total cycle.

Overall Grade = f(Correctness, -Performance), f is monotonically increasing.