CS 243 Homework 6

Winter 2017

Due March 9, 2017 (Thursday), 3pm

Directions:

- This is a programming assignment. Download the starter code from the course website, complete the task (see "Deliverables" in Introduction below), and submit your code using the submission script on the `myth` machines.
- You may work alone or in pairs. If you are working in a pair, be sure to supply your partner's SUID to the submit script so they get credit.
- If you need to use late days, just hand in your code up to two days late using the submission script. The submissions are timestamped so your late days will be used automatically.
- **We will be addressing installation issues about the code and questions about Halide in a hackathon on Friday, March 3, in Gates B01, 2:30pm-3:20pm.** We will also go through some Halide tutorials in the session. Please come to the review-session/hackathon if you need help with Halide, or installing and running Halide.

1 Introduction

In this lab, you will get an introduction to Halide and have the opportunity to explore some of the scheduling mechanisms that the language offers. You will begin by working through a series of Halide introductory tutorials before experimenting with different scheduling policies for an unsharp mask filter. **Deliverables include your scheduled unsharp mask filter as well as a `README` that includes answers to various short-answer questions.**

We recommend using a local Linux or OS X machine for this assignment. The starter code will not run on Windows. You can also use `myth` for this assignment. We have tested the assignment on `myth`.

2 Starter Code and Getting Started

Please download the starter code here. The directory is organized as follows:

- `hw6/Makefile` : Makefile for compiling our code.
- `hw6/submit` : Specify schedules here. Each file specifies unsharp algorithm, and space for you to specify a schedule. You will submit this directory only.

  - `hw6/submit/filter_part_1_naive.cpp`: file for naive schedules
  - `hw6/submit/filter_part_2_single.cpp`: single core schedule goes here
  - `hw6/submit/filter_part_3_multi.cpp`: multi core schedule goes here

- `hw6/images` Test images for running unsharp:
  - `parrot.png` : A 768×1280 image
  - `oak.png` : A 4608×2592 image

- `hw6/build` : Build files will go here. This directory is not part of zip, and will be overwritten.

You will need to install all required dependencies before starting. Please see instructions in section 3 for installing dependencies. The Makefile expects halide files in a `halide` directory in hw6. If you want halide files elsewhere, you can edit the `HALIDEROOT` variable at the top of Makefile. After installing all dependencies, you can build the code as follows, from `hw6` directory:

`make`	Builds files for all parts
`make part_1`	Builds only `part_1`
`make part_2`	Builds only `part_2`
`make part_3`	Builds only `part_3`
`make clean`	Remove all built files

You can run each part as follows from `hw6` directory:

part 1: `./build/filter_part_1_naive` *`input-image [output-prefix]`*
part 2: `./build/filter_part_2_single` *`input-image [output-prefix]`*
part 3: `./build/filter_part_3_multi` *`input-image [output-prefix]`*

Replace *`input-image`* by `images/parrot.png` or `images/oak.png`. The argument *`[output-prefix]`* is optional. If you supply it, the output image will be prefixed with that argument. Example:

- `./build/filter_part_1_naive images/parrot.png`
- `./build/filter_part_1_naive images/parrot.png parrot-output`
- `./build/filter_part_1_naive images/parrot.png output/`, where output is an existing directory.

Note: On Mac, if you receive errors saying "Library not loaded" when executing the filters, try running the commands with `DYLD_LIBRARY_PATH=./halide/bin` prepended (assuming you extracted the halide distribution into your `hw6` distribution.

3 Installing Dependencies

- **Halide:** You will need to download the proper Halide distribution for your machine. Download a distribution of Halide here, and extract the files into `hw6` directory. **You need to follow this step even if you are developing on `myth`.**
- **libpng-dev:** Check if your machine has `libpng-dev`. You can do this using `libpng-config --version`. If it is not found, you will have to install `libpng-dev`. You can do this using your package manager. **You do not need to do this on myth.**

4 Halide Tutorials

The distribution contains several tutorials (with animated versions of these available online here). Read through tutorials 1-8. After working through the tutorials please answer the following questions in your `README`:

1. What is the purpose of Halide's split and tile operation?
2. What is the purpose of Halide's reorder operation?
3. What architecture specifics should be considered when vectorizing an operation?
4. Imagine that you are running a Halide pipeline on an old cell-phone with limited RAM (i.e. storing intermediate results may result in paging to disk). Discuss the performance tradeoffs and considerations (parallelism, locality, etc.) you must consider when scheduling your pipeline for the phone.
5. Now you are running the same Halide pipeline on a new smartphone (i.e. we are no longer memory-limited). How will your analysis of tradeoffs when scheduling your pipeline change?

5 Unsharp Mask

An unsharp mask is a multi-stage image processing pipeline, often used to sharpen images. It computes a blurred (unsharp) image, and subtracts that from the original image to emphasize the high spatial frequency components, and reduce the low frequency components. This sharpened image is then combined with the original image to produce an image with a higher contrast. In this assignment, we will schedule an unsharp mask pipeline. The pipeline consists of 6 stages:

1. `gray`: computes a gray scale image
2. `blur_y`: blurs output of 1 (gray scale image) along `y` using a 1×7 mask
3. `blur_x`: blurs output of 2 along `x` using a 7×1 mask
4. `sharp`: subtracts output of 3 (blurred gray scale) from 1 (gray scale)

5. `ratio`: scales output of 4 (sharp) by that of 1 (gray scale value)

6. `result`: combines output of 5 (ratio) and input image

The input images are colored (RGB) images. In Halide, consecutive elements of `x` are contiguous in memory. All `x, y` values for first color are contiguous, followed by second color and then the third.

6 Scheduling

We will be running the image processing pipeline over CPUs only. Please go through the parts 0 to 3 below, and then answer the questions at the end in the `README` file. You need to rebuild your code everytime you update your schedule.

6.1 Part 0: Machine Specs

Lookup information about your CPU using the command. On Linux, you can do this using `lscpu` and `sudo dmidecode -t processor`. On OS X, you can do this using `sysctl`, as `sysctl -a | grep machdep.cpu`, `sysctl -n hw.ncpu`, `sysctl -n hw.l1dcachesize`, etc Check how much cache is available, how many CPUs you have, and whether your CPU supports vectorization (SSE or AVX).

6.2 Part 1: Naive Scheduling

Halide, by default, inlines all functions. In this part, we will measure the time for a schedule where all functions are inlined, and implement a naive schedule, where we compute and store all intermediate results before proceeding to the next stage. Implement the naive schedule in `filter_part_1_naive.cpp`. You can test your code as described in section 2. This is your baseline, any schedule you write in later parts should do better than the naive schedules!

6.3 Part 2: A Better Schedule for a Single Core

Implement your schedule for a single core, in `filter_part_2_single.cpp`. You can use split, tile, reorder, unroll, vectorize operations, and any other operations discussed in the scheduling tutorials. You can modify how many intermediate results you store, how much of intermediate results you compute, which ones you inline, etc. Look at the operations and dependencies in the pipeline, and give some thought to how you might want to schedule the pipeline for your machine. Do not modify the algorithm, and do not use multiple cores (parallel). Try out many different schedules, and save the best schedule you can find. Test it as described in section 2. Optimize your schedule for the larger image, `oak.png`.

If you do not see any expected improvements from vectorization, LLVM is probably automatically vectorizing code generated by Halide.

6.4 Part 3: Running Even Faster with Multi Core

Now write a schedule for multiple cores, in `filter_part_3_multi.cpp`. You may have to edit your schedule from the previous part slightly, or have to write a completely new schedule, to get a good schedule for multiple cores. As before, do not modify the algorithm. Try out many different schedules, and save the best schedule you can find. Test it as described in section 2. Optimize your schedule for the larger image, `oak.png`.

6.5 Questions

Answer the following questions, in `README`.

6. Where did you run the assignment (`myth` or a local laptop or a local desktop)?
7. Does the machine you ran the assignment on support vectorization?
8. How much data cache is available?
9. How many total cores does the machine have?
10. How much time did the 2 naive schedules take, for `parrot.png` and `oak.png`?
11. Explain your single core schedule, and why you think it is the best for your machine, in 2-4 sentences.
12. How much time did the single core schedule take, for `parrot.png` and `oak.png`?
13. Explain your multi core schedule, and why you think it is the best for your machine, in 2-4 sentences.
14. How much time did the multi core schedule take, for `parrot.png` and `oak.png`?
15. How did you modify your single core schedule for multiple cores?

7 Submission

Please submit your modified scheduling code and `README` with answers to questions from section 4 and section 6.

To submit your assignment, follow the following steps:

1. Copy your `submit` directory to `myth.stanford.edu`, if you are not already working there. Make sure that you have added your `README` to the `submit` directory.
2. SSH into `myth.stanford.edu` or open a terminal in person.
3. `cd` to the directory containing your `README` and modified scheduling code that you want to submit.

4. run `/usr/class/cs243/submit` in this directory. If you're working with a partner, type:

   ```
   /usr/class/cs243/submit partner_SUID
   ```

   (ex: `./submit jrkoenig`) Only one submission is required per pair.

5. If you discover a bug after submitting (and before the due date), simply run the submission script again. We'll take the latest version.