MIT EECS 6.815/6.865: Assignment 8:

Introduction to Halide and paper review

Due Wednesday November 20 at 9pm

1 Summary

- Setting up Halide
- Tutorials on the main concepts of the language
- Application Exercises
- **6.865:** paper review

In this pset, you will get acquainted with Halide. We will first go through environment setup. Then you will follow a tutorial that introduces the basic elements of the language and apply them on simple exercises.

Make sure to look at the lecture slides for background as this assignment takes a large side-step from the material we've covered in the first half of the semester.

The description of the tutorials exercises can be found in the `tutorial*.cpp` files. Go through the tutorials one by one, each tutorial is supposed to explain basic concepts and test them on slightly different use cases.

The grad version has a paper review in addition of the Halide tutorials and exercises.

2 Installing Halide

The next two problem sets will use Halide. If you have trouble setting it up, please come to office hours or ask on Piazza.

2.1 Obtaining Halide

Precompiled binaries. We recommend that you use precompiled binaries of Halide from https://github.com/halide/Halide/releases. Use the trunk version for your favourite operating system. Linux will be easiest, followed closely by Mac. Windows should be more troublesome, but not impossible. For Linux, make sure to get the binaries corresponding to your version of GCC and 32/64 bits depending on your OS.

We recommend Windows users use Athena for the Halide problem sets. Alternatively, you can try the Windows Subsystem for Linux.

Compiled from source. If the precompiled binaries do not work right away, you can compile Halide from source, which is what most of us who actually use Halide have done. You will find instructions at https://github.com/halide/Halide (see "Building Halide"). But try the precompiled binaries first as those will save you some time.

External libraries (Mac users). You need to install libpng through Macports https://www.macports.org/ or Homebrew https://brew.sh/. Consult the sites for install instructions and specifics on installing packages.

Assuming that the installation succeeded, you should be able to run:

```
libpng-config --I_opts
libpng-config --L_opts
```

Please add the output of the above commands to the variables `PNG_INC` and `PNG_LIB` in the Makefile.

2.2 Working on Athena

Here is a step-by-step on working on Athena for anyone who wishes to.

- log in to Athena, e.g.: `ssh username@athena.dialup.mit.edu`.
- check the version of gcc installed: `gcc -v`, should return something like: `gcc version 4.8.4`.
- choose a directory to install Halide. e.g. `cd $HOME/Documents`
- download the appropriate version of Halide from https://github.com/halide/Halide/releases, e.g. for Linux 64 with gcc5.3, it's `wget https://github.com/halide/Halide/releases/download/release_2018_02_15/halide-linux-64-gcc53-trunk-46d8e9e0cdae456489f1eddfd6d829956fc3c843.tgz`.
- extract the archive `tar -zxf halide.tar.gz` and note the root path of the installation.
- To transfer files from your local machine to Athena you can use `rsync -rv localpath userame@athena.dialup.mit.edu:remotepath`.
- To transfer files from Athena to your local machine you can use `rsync -rv userame@athena.dialup.mit.edu:remotepath localpath`. This can come in handy to pull the `Output` folder after running your code for example.

2.3 Documentation

You can find Halide documentation and examples at https://github.com/halide/Halide/wiki and http://halide-lang.org/tutorials/tutorial_introduction.html. Let us know if things don't work!

3 Running the code

At this point you should have Halide installed on your machine.

Make sure to modify the `HALIDE_DIR` and `HALIDE_LIB` variables in the Makefile to point to the right path and library file. `HALIDE_DIR` should point to the root of the Halide path. While `HALIDE_LIB` points to the library in `HALIDE_DIR/lib/bin/libHalide.<ext>` where `<ext>` is dependent on your operating system.

1.a Change the environment variable (in the Makefile) `HALIDE_DIR` to point to the root of your Halide installation.

1.b OSX users: install `libpng` via Homebrew of Macports and update the environment variables `PNG_INC` and `PNG_LIB` if needed.

1.c Test your installation by running `make`. The code should compile and you will be able to run the tutorials.

In this problem set we will not use any of the code you wrote for the previous assignments. In particular we will *not* be using the standard `Image` class. Instead we will be using Halide's Image buffer.

All deliverables have to be implemented in a `a8.cpp` in their respective functions. Each function has a description of what-to-do in comments above. Please read the comments as they are your main guide on this problem set. Performance statistics have to be reported on the submission system.

As usual, use `a8_main.cpp` to test and debug your functions.

3.1 Helpful Functions

- We have provided a function `profile()` to measure Halide function performance.
- We have provided a function `apply_compute_root` that sets all Halide functions to `compute_root`. When applicable, your schedules should be faster that this baseline.
- Use `compile_to_lowered_stmt()` to output loop nests for debugging schedules, some tutorials show how it is meant to be used.

4 Func, Var, Expr and input/output buffers

Read on through `tutorial01` to `tutorial04` in the corresponding `.cpp` files. These will show you the basic Halide syntax, using variables (`Var`), expressions (`Expr`) and functions (`Func`).

2.a In `a8.cpp`, implement the function `SmoothGradNormalized`. This function is very similar to the example in `tutorial01`, but the output is normalized by 1024 inside the Halide pipeline. This will teach you how to generate a grayscale image.

2.b Implement `WavyRGB` in `a8.cpp`. This will teach you how to generate an RGB image.

2.c Implement `Luminance` in `a8.cpp`. You will learn to read an input RGB image and produce an output grayscale image.

2.d Implement `Sobel` in `a8.cpp`. You will learn to write a multi-stage pipeline, that uses neighboring pixels.

5 Scheduling a computation

The previous section should have walked you through Halide's basic syntax. You are now able to specify your algorithm in Halide using Func, Var, Expr and Image buffers.

In this section we will learn how to change the *schedule* of a computation. That is, determine which pixel gets computed in what order. Halide's scheduling syntax allows us to explore various schedules: tiling, parallelizing, vectorizing operations, fusing stages together, etc. This allows us to quickly optimize our algorithm and is really where the language shines.

Read `tutorial05` to get acquainted with the essential scheduling primitives of Halide on a single stage pipeline: `reorder, fuse, split, tile, vectorize, parallelize, unroll`.

For multiple stage pipelines (for example, extract the x and y gradients from an image, get their magnitude and blur the result), Halide's default schedule is to *inline* all the operations. This means that no intermediate buffer will be allocated, and for every pixel in the output, the values in the intermediate stages will be recomputed. The extreme opposite to inlining is when every stage in the pipeline is scheduled as `compute_root`: in this case every stage writes all of the required values to an intermediate buffer. While scheduling Halide algorithms, we try to find the optimal mid-point between these options, balancing locality (i.e. minimizing large memory transfers) with redundancy (i.e. minimizing the extra-work).

To make things more concrete, read `tutorial06` which walks through several schedule options for a fixed-sized box blur. The algorithm has two stages, first blur along x then along y. You will learn about `compute_at, compute_root,`

`compute_inline`. You will also learn about tiling as a trade-off between memory locality and recomputation.

3.a Report the runtime, throughput of the four schedules in `tutorial06` (in the submission form). Which is best? Can you say why it was the best? (**Note:** the written portions of this assignment is weighted more heavily than in previous assignments). (Answer in the submission form).

3.b Write equivalent C++ code for schedule 5 of `tutorial06` in `a8.cpp`. (See example for schedule 1 in `a8.cpp`).

3.c Write equivalent C++ code for schedule 6 of `tutorial06` in `a8.cpp`. (See example for schedule 1 in `a8.cpp`).

3.d Write equivalent C++ code for schedule 7 of `tutorial06` in `a8.cpp`. (See example for schedule 1 in `a8.cpp`).

6 Putting it together

Now we are ready to do some real image processing. We will implement a Gaussian blur filter in Halide. We will explore various scheduling strategies in order to produce efficient code. This will leverage the skills you learned in the previous tutorials.

4.a Implement a separable Gaussian blur `Gaussian` with binomial weights `1/16 (1, 4, 6, 4, 1)`. (i.e. the filter size should be 5) in `a8.cpp`. You can find a (slow, C++) reference implementation `Gaussian_cpp` in `reference_implementations.h`. (ignore the boundary artifacts). **Note:**

- This first applies a blur on the horizontal dimension and then a blur on the vertical direction.
- **Important:** In this problem, we do not want a general convolution as doing so requires use of more advanced Halide features we have not covered yet. Instead, compute the weights of the kernel offline by hand and hard-code it into your implementation.

4.b We will be looking for the best scheduling strategy for the Gaussian blur we've implemented above. You will answer each of the following question in the submission form.

- The default behavior of Halide is to inline all Halide functions. Let's focus on the two functions that perform the horizontal and vertical blur. Compare the performance between the default

schedule (i.e. `compute_inline`) and a schedule that applies `compute_root` on both of them. Which one is faster? Why?

- Next, instead of `compute_root` or `compute_inline`, implement a schedule by tiling the image and apply separable Gaussian filter on each tile and only store one tile per time. Specifically, use `tile` and `compute_at` to schedule the program.

  Is it faster than the previous best strategy (`compute_inline` or `compute_root`)? Why? Which tile size is the best (on your machine)?

- Next, we will try to parallelize and vectorize the computation using `parallel` and `vectorize`. Let's go through this step-by-step.

  - Vectorize the inner-most-loops for both blur functions (along `x` and `y` dimensions). The width of your vectorization should be the same as your CPU's vector unit width.
  - Parallelize your pipeline so that the outer-most tile iterations are multi-threaded.
  - Answer the following questions by running the code with different combinations of the above steps.

  How much speedup do you expect? Why? How much speedup do you actually get? If there are discrepancies between expected and actual speedup, can you say why?

  Note: In order to check your CPU capabilities:

  - Linux: Run `lscpu`
  - Mac: Go to "About This Mac" under the Apple menu.
  - Windows: Run `dxdiag` from the Run application

  to check the number of cores and the model of your core. You can then check online what the vector ALU unit width is for your CPU core model on the Intel website. If you're using a laptop younger than 4-5 years old, you can safely assume that the vector unit is at least 8×32 (meaning you can run 8 32-bit or 4 64-bit floating numbers concurrently).

- Finally, we want to compare the performance between parallelization and vectorization with `compute_at` or without `compute_at` (i.e. using `compute_root` or `compute_inline` whichever is faster from the first question). Which one is faster? Why?

  Note: Note that if you naively remove the `compute_at` schedule, the compiler will complain that vectorizing an inline function is not allowed. So make sure to switch the schedule from `compute_at` to `computer_root`.

7 6.865: Paper Review

This part is for 6.865 only.

Unlike the previous psets, it's going to be all reading and writing. Your job is to review a recent publication in computational photography. This will hopefully give you some more insight into the review process that these papers go through.

You should select one paper from the provided `papers.html` file (if the link is dead, search for the paper on Google), or a computational photography paper from the SIGGRAPH lists here: http://kesen.realtimerendering.com/. If there's another relevant paper that you're particularly interested in reading, ask us about it and we'll probably be okay with it (unless it's something that you're already supposed to read for one of the problem sets). If you're not sure if a particular SIGGRAPH paper falls under computational photography, feel free to ask us on piazza.

You should use the SIGGRAPH review form, available at the following URL: https://sa2018.siggraph.org/en/submitters/technical-papers/review-form. For "Explanation of Rating", try to come up with at least one outstanding question that you think the authors didn't address in the paper. You can skip the "Private Comments". Try to be thorough! Probably a paragraph or two for each question, and more for "Explanation of Rating".

One final note: don't be compelled to like a paper because it was already published. Every paper has some shortcomings. Maybe the method is very elegant and general but doesn't end up producing very impressive results. Maybe the results are spectacular, but the method itself has some mathematical holes. Maybe everything seems great, but there just aren't enough details to reproduce the results. It's up to you as a reviewer to weigh the strengths and weaknesses of each paper

5 Include your paper review as an ascii text file called `paper_review.txt` in the `asst` directory of your submission.

See the text for details on what should be included. In addition please add paper title, authors, venue, year and valid link.

8 Submission

Turn in your files to the online submission system and make sure all your files are in the `asst` directory under the root of the zip file. If your code compiles on the submission system, it is organized correctly. The submission system will run code in your main function, but we will not use this code for grading. The submission system should also show you the image your code writes to the `./Output` directory

In the submission system, there will be a form in which you should answer the following questions:

- How long did the assignment take? (in minutes)
- Potential issues with your solution and explanation of partial completion (for partial credit)
- Any extra credit you may have implemented and their function signatures if applicable
- Collaboration acknowledgment (you must write your own code)
- What was most unclear/difficult?
- What was most exciting?