MIT EECS 6.815/6.865: Assignment 9:

Halide on your own!

Due Wednesday November 27 at 9pm

1 Summary

- Familiarize yourself with reductions in Halide and implement a simple function
- Implement a general convolution Gaussian Filter and scheduling its computation
- Implement a multi-stage pipeline featuring the Unsharp Mask
- Scheduling the computation pipeline for Unsharp Mask
- 6.865: Writing an auto-tuner to find a good schedule

2 Reductions

We will start off this pset by learning about Halide reductions which will really help us implement more complex pipelines. Read `tutorial07` to learn about reductions.

With these features learned, we are finally ready to do non-trivial tasks in Halide. We will re-implement the Gaussian blur filter, but this time we will compute it for arbitrary sized convolutions. We will explore various scheduling strategies as we did in the previous assignment, but this time you will finally see why Halide is useful when optimizing image processing algorithms.

But just a bit more reading first; read `tutorial08` and `tutorial09` to see how reductions can be used to perform convolutions.

As you implement and debug your Halide code, you might find it useful to run your code on the smaller image `grey-small.png`. Make sure you use the larger `grey.png` image when answering questions about runtimes of your implementations.

1.a Run `tutorial09`. What is the best tile width on your machine? (Answer in the submission form).

1.b Implement a 1D-horizontal Gaussian blur `Gaussian_horizontal` in `a9.cpp`. You can find a (slow, c++) reference implementation named `Gaussian_horizontal_cpp` in `reference_implementations.h`. (ignore the boundary artifacts). **Note:** that this is NOT the same Gaussian you implemented in the previous pset as you need to compute

the kernel weights for the convolution. I.e., the kernel computation and normalization should be done **in Halide!**

1.c Schedule the blur operation you just implemented. What should be the best scheduling strategy for comuting the kernel? Please explain your reasoning. (Answer in the submission form).

2.a Implement a separable Gaussian blur `Gaussian` in `a9.cpp`. You can find a (slow, c++) reference implementation `Gaussian_cpp` in `reference_implementations.h`. (ignore the boundary artifacts). Notice that this time you have multiple producers in your pipeline: the kernel, the normalized kernel, and the horizontal blur.

2.b The best scheduling strategy for a Gaussian blur varies with the size of the kernel. For the following sub-questions, let's optimize the code for a kernel with `sigma = 3` and `truncate = 3`. You will answer each of the following question in the submission form.

- What are the running time and throughput of the reference implementation on your machine? What are the running time and throughput of your Halide implementation with all functions scheduled to `compute_root`? Please provide your machine specs.
- Tile, parallelize and vectorize your computations. Make sure that the producer is computed at the correct granularity. What was the performance improvement?

Now that you have mastered the essential Halide building blocks, let us code a complete image processing pipeline that involves several computation stages. This should give you the opportunity to leverage Halide's scheduling tools and optimize the computation across different stages. This problem set will be less guided than the previous one.

The graduate version also asks you to implement an auto-tuning algorithm to explore various schedules and find the most efficient one *automatically*.

3 Final Stretch: Unsharp Mask

We provide you with a reference C++ implementation of the Unsharp Mask operation you already implemented in the first problem set with slight modifications. You can find the code in `reference_implementation.h`. The rough outline for the problem is shown below:

- Apply a contrast-adjusted gamma correction to the image following the equation below. Here, the equation assumes your input image range is $[0, 1]$. If your image is in the range $[0, 255]$, you need to either adjust the equation or normalize your input. Please make sure to clamp the output

image to valid pixel values.

$$factor = \frac{1.0 - 0.5}{1.0 - 0.5^{1/\gamma}}$$
$$Out(x, y, c) = factor \times (In(x, y, c)^{1/\gamma} - 1.0) + 1.0$$

- Apply Gaussian Blur to the adjusted image with the specified `sigma` and `truncate`.
- Construct the high-pass image using the input image and the blurred image.
- Recombine the high-pass image (times the sharpening strength) with the gamma adjusted image from step one.

Following this code, implement the same algorithm in Halide in `a9.cpp` (as usual, you can test it using `a9_main.cpp`). Make sure to test the intermediate steps as well!

You can output the intermediate steps of the reference algorithms by uncommenting code in `reference_implementation.cpp`. But make sure that you are not outputting images in the reference function when comes the time to compare performances: writing `.png` files to disk takes a significant chunk of time!

Once you have a working implementation of the filter, run it with a schedule in which all your `Func` are `compute_root` (you can use `apply_compute_root` for this). Then come up with a better schedule for Unsharp Mask. Leverage the Halide primitives as best as you can!

For the scheduling portion of this task, set `sigma = 3` and `truncate = 3`. For time measurement, please also provide your machine's characteristics (number of cores, number of lanes of SIMD units, cache size, etc).

3.a Implement `UnsharpMask` in `a9.cpp`. The final output should match that of the reference code (everywhere but perhaps not at the image boundaries). **Note:** Make sure that your unsharp mask operation consists of **exactly** two Funcs: (1) high-pass image construction and (2) recombination of the high-pass image and the gamma-corrected image. (Obviously the steps before that – gamma correction and gaussian – require additional Funcs as required). We want to see the pipeline organized in this way so that we can construct interesting schedules in the next part.

3.b What are the running time and throughput of the reference implementation on your machine? What are the running time and throughput of your Halide implementation with the default schedule using `apply_compute_root`? Please provide the machine specs. (answer in the submission form)

3.c Find a better schedule than the default one and report your per-

formances (running time and throughput, as well as you machine's characteristics). Answer in the submission form.

4 6.865 only: Auto-tuning

This is an open-ended part. Write C++ code that systematically tries many schedules for your Harris corner detector. Automatically explore different scheduling strategies (e.g. what gets tiled, what gets computed at what granularity) and different numerical values for things such as tile size. `tutorial09` from the previous problem set might give you some idea but it is relatively simplistic. In particular, note that some strategy decisions (e.g. tiling a particular computation) can generate new possible choices (tile size) that might not be relevant for other strategies. Please try at least 4 different scheduling strategies, *e.g.*, inline, compute_at, root, tiling, vectorize/paralllelize at different stages of the pipeline, with various parameters.

Have a look at the function `autoschedule_unsharp_mask` in `a9.cpp` for a simple example of how to explore different schedules. Note that the `UnsharpMask` function you implemented above takes a `schedule_index` and `schedule_parameters` as optional arguments. You can use this to dynamically select the schedule in `UnsharpMask` and pass an array of integer parameters needed by that schedule, if any.

4 In one paragraph, describe your overall approach in the submission form. Describe your best schedule (including parameters) together with its performance. Turn in your code.

5 Extra Credits (10% max)

Implement the following image processing pipelines in Halide.

- Bilinear/Bicubic/Lanczos Resampling (5%)
- Harris Corner Detector (10%)
- Bilateral Filter (10%)
- Full HDR pipeline (computation of weights, merging of images and tone-mapping) (10%)
- GPU implementation of Unsharp Mask [note this requires Halide compiled with GPU support] (10%)
- Demosaicing (10%)
- Demosaicing + denoising (with a small bilateral filter) (10%)
- General convolution implementation with a schedule that depends on the kernel size (10%)

- Computation of final panorama with blending (e.g. images and homographies are computed in C++ and treated as inputs to halide code) (10%)

6 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?