Ryan A. Margraf

WES 237B

Assignment 3

8/21/2022

Lab: FIR Filters

How To Run

1. Open a terminal
2. Navigate to the “Assignment_3/lab3_pynq” or “Assignment_3/lab3_jetson” folder depending on which platform you are using.
3. Type “make” and press enter
4. The program can be run by using the command

```
./lab3_fir
```

Performance Comparison

Performance By Optimization Level on Pynq

	Pynq - O0	Pynq - O1	Pynq - O2	Pynq - O3	Pynq - Ofast
fir() percent	34.2%	29.4%	40.2%	40.2%	36.4%
fir_opt() percent	29.3%	29.4%	33.5%	33.5%	36.4%
fir_neon() percent	29.3%	23.5%	26.8%	26.8%	27.3%
fir() time	0.28	0.05	0.06	0.06	0.04
fir_opt() time	0.24	0.05	0.05	0.05	0.04
fir_neon() time	0.24	0.04	0.04	0.04	0.03
overall run time (gprof)	0.82	0.17	0.15	0.15	0.11
real time	1.665	1.021	1.008	1.007	0.973
user time	1.654	1.011	0.987	0.996	0.952

It appears that most of the time improvements comes from going from -O0 to -O1, at least for this specific application.

Performance Comparison of profiling (-pg) turned on and off

	Pynq -O1 no -pg	Pynq -O1 with -pg
real time	1.011	1.021
user time	0.991	1.011

Turning off profiling resulted in a small, but measurable decrease in the real and user time obtained with the "time" command. This makes sense since the compiler adds in extra code that is responsible for measuring the performance.

Obviously, since the profiling was turned off, we cannot directly compare the time values of individual functions here.

Platform Comparison: Jetson vs Pynq

	Jetson - O1	Pynq - O1
fir() percent	50%	29.4%
fir_opt() percent	50%	29.4%
fir_neon() percent	0	23.5%
fir() time	0.02	0.05
fir_opt() time	0.02	0.05
fir_neon() time	0	0.04
overall run time (gprof)	0.04	0.17
real time	0.185	1.021
user time	0.176	1.011

Platform clearly matters, as these results show. The Jetson showed a large performance improvement over the Pynq in all the measurements compared. I was originally going to use the Jetson to compare the optimization levels, but it turned out to be too fast for gprof to measure, as can be seen by the 0 sec. times here.

Assignment: Sobel Filters

How To Run

1. Open a terminal
2. Navigate to the “Assignment_3/sobel” folder
3. Type “make” and press enter
4. The program can be run using a command of the format

./hw3 <mode> (<width>)(<height>)

Mode 1 uses the naïve algorithm, mode 2 uses an unrolled loop, and mode 3 uses NEON. The width and height parameters are optional and specify the size of the image to use. If only width is specified, the height will be made to be the same. For example, to use NEON with an image size of 768x768, type

./hw3 3 768

Optimizations

My optimizations for Neon were based on the optimizations I made for the unrolled loop, so I’ll start with those.

First, I noted that checking the bounds of the kernel at every iteration required a lot of operations, so I made it instead start 1 pixel into the image and end 1 pixel early. This results in a 1-pixel wide border around the image where the results will not be correct, but since this is an edge detection algorithm, that is unlikely to be useful anyway.

As for unrolling the loop itself, I simply wrote out each line in the multiply-accumulate so that 2 loops for the kernel are no longer required. Furthermore, since some of the elements in the Sobel kernel are 0, I removed those lines. This decreased the number of multiply-accumulate operations required for each pixel from 18 to 12.

Noting that the x and y Sobel kernels are just transposes of each other, I also just flipped the indexing when doing the y direction. Since it is only accessing one array, it saves memory and cache space.

Some vectors needed to be created. I created Sobel vectors for the x and y dimensions, and a data vector with the data required to calculate 1 pixel. Since the central element in the Sobel kernel is 0 in both the x and y dimensions, it can be eliminated from their respective vectors, and the corresponding element can be eliminated from the data vector. This allowed me to use the int16x8_t type, so all of the data for 1 pixel can be worked on at once.

With this, the number of operations to calculate the Sobel value for a pixel is reduced to 4 (2 in each dimension: A multiply for the data and Sobel kernel, followed by a sum across the vector).

Performance Comparison

Sobel Algorithm Comparison on Jetson -O1

Image Size	1536	3072
Naïve	0.14	0.56
Unrolled	0.05	0.17
Neon	0.06	0.23
OpenCV*	0	0

The above table shows the performance comparison with 2 different input sizes. The unrolled algorithm significantly improves over the Naïve implementation. The Neon implementation, however, is actually slower than the unrolled. I suspect this is because of the required memory copying to a new array before loading it to the vector, since it is not contiguous. I thought working on several pixels in parallel instead, but the types for the operations I needed always seemed to be incompatible, and still couldn't find a good way to manage the memory.

OpenCV is likely using the GPU on the Jetson, which makes it too fast to be captured by the 0.01 time resolution of gprof, even with large input sizes.

Mini-Assignment: Introduction to CUDA

The global flag specifies a function that is called from the host, but runs on the device.

```
foo<<4,32>>(out, in1, in2)
```

The above indicates a function that uses a grid of 4 blocks of 32 threads.