Sam Weismuller
WES237B
8/19/2022

Assignment 3 Report

Part 1:

1. Execution time of each FIR implementation at various optimization levels:

	O0	O1	O2	O3	Ofast
naïve	0.3 s	0.06 s	0.06 s	0.06 s	0.06 s
opt	0.24 s	0.05 s	0.05 s	0.05 s	0.04 s
neon	0.23 s	0.04 s	0.04 s	0.04 s	0.03 s

The Neon optimization is narrowly better than the unrolled loop. The time benefit of the optimizations decreases as the optimization level increased. However, the relative performance improvement over the -O0 naïve solution is still noticeable when comparing -Ofast to the other optimizations:

	O0	O1	O2	O3	Ofast
naïve	1	5	5	5	5
opt	1.25	6	6	6	7.5
neon	1.304348	7.5	7.5	7.5	10

2. Execution time comparison of -O0 with and without the -pg flag:

	w/o pg	w/ pg
time	1.681 s	1.683 s

The execution time was marginally longer with the -pg flag. However, it is ultimately not a significant difference and could have just been coincidental. When we use the flag, the compiler is inserting instructions to allow us to gather the profiling information that can be parsed by gprof. However, the execution time of the profiling instructions is minimal compared to the execution time of the actual workload. A lot of the work done by the profiler is done per function call, and our program doesn't really contain a lot of subroutines.

3. Comparing the PYNQ and the Jetson board at -O0:

	PYNQ	Jetson
naïve	0.3 s	0.05 s
opt	0.24 s	0.04 s
neon	0.23 s	0.03 s

The Jetson board is clearly faster, though speed improvement of the optimizations is similar, though the performance improvement over the naïve solution on Jetson is technically higher:

	PYNQ	Jetson
naïve	1	1
opt	1.25	1.25
neon	1.304348	1.666667

Sam Weismuller
WES237B
8/19/2022

Part 2:

1. Outside of the loops, the x and y Sobel kernels were packed into 3 int16x8 variables. The first contained rows 1 and 2 of the x kernel, the second contained rows 1 and 2 of the y kernel, and the final one contained the third rows from both kernels. Inside the actual loops, the 3x3 area of the input image that needed to be considered was loaded row by row into 3 uint8x8s. In order to correct the signage of the data and to prevent overflow when multiplying, these were then converted into int16x8s. The resulting int16x8s were then packed in the same way the kernels were, with rows 1 and 2 going into one variable and two copies of row 3 going into the other. The multiplications were then performed and the results of each kernel were accumulated separately (with the third row requiring us to get the necessary low/high portion of the int16x8). The results of the x and y kernels were then combined as normal by obtaining the combined magnitude and clamping to 255.

2. These execution times were obtained on a Jetson board at -O2 for various square image sizes:

	3500	7000	14000
naïve	0.41 s	1.6 s	6.49 s
unroll	0.26 s	0.99 s	3.8 s
neon	0.25 s	0.99 s	3.81 s

And in terms of performance improvement over the naïve solution:

	3500	7000	14000
naïve	1	1	1
unroll	1.576923	1.616162	1.707895
neon	1.64	1.616162	1.703412

The unrolled loop and neon ended up being practically the same performance wise. However, at lower optimization levels, the unrolled loop would be noticeably better than the Neon implementation. This is due to a couple of main limiting factors that made the Sobel filter a bad fit for optimizing with Neon:

1. The input data has poor locality. We have to read from 3 different rows for each target pixel we run the Sobel kernel over. Since the input data is a large matrix in row major order, the distance between these rows can be great. Neon simply doesn't have intrinsics that can load rows at a large, arbitrary distance away from each other, so we must read each row individually.
2. The input data has to be converted before we can operate on it. The input image is in unsigned 8-bit values, but we must perform a multiply with a signed kernel and get a potentially larger than 8-bit result. Converting the input to signed 16-bit values to deal with this issue costs time.
3. The kernel is only 3x3, so we actually aren't doing that many operations in the inner loop. Assuming the compiler isn't combining the multiply and accumulate steps, it still only comes out to 18 multiplies and 18 adds to apply both kernels. That doesn't give us much room to squeeze out performance gains.

The Neon instructions are probably a bit slower at lower optimization levels due to the compiler not optimizing out the overhead of mapping the Neon intrinsics to the C language and thus it is doing unnecessary data copying and variable initializations.

Sam Weismuller
WES237B
8/19/2022

3. Even utilizing a high frame count, I was unable to capture OpenCV's execution time with gprof. This means it was reliably running at less 0.01 seconds. That makes it incredibly faster than our implementations where the fastest speed we would see is 0.25 seconds. This isn't too surprising as edge detection is a key operation in computer vision, so OpenCV would need to be able to perform it as fast as possible. There are also a lot of zero values in our kernel, which means we are doing a lot of unnecessary lookups, multiplies, and adds.

Extra credit:

Comparing the unrolled loop and the neon implementations with gcov, we can get an idea of what the optimizer was doing to the loops:

Unrolled loop:

```
for(int row = 1; row < HEIGHT-1; row++)
        -:   66:    {
391902006:   67:        for(int col = 1; col < WIDTH-1; col++)
        -:   68:        {
        -:   69:            int accum_x = 0;
        -:   70:            int accum_y = 0;
        -:   71:
        -:   72:            // calculate convolution with x kernel
391888008:   73:            accum_x += (sobel_kernel_x[0][0]*
195944004:   74:                        src_ptr[(row+0-1) * WIDTH + (col+0-1)]);
391888008:   75:            accum_x += (sobel_kernel_x[0][1]*
195944004:   76:                        src_ptr[(row+0-1) * WIDTH + (col+1-1)]);
391888008:   77:            accum_x += (sobel_kernel_x[0][2]*
195944004:   78:                        src_ptr[(row+0-1) * WIDTH + (col+2-1)]);
391888008:   79:            accum_x += (sobel_kernel_x[1][0]*
195944004:   80:                        src_ptr[(row+1-1) * WIDTH + (col+0-1)]);
391888008:   81:            accum_x += (sobel_kernel_x[1][1]*
195944004:   82:                        src_ptr[(row+1-1) * WIDTH + (col+1-1)]);
391888008:   83:            accum_x += (sobel_kernel_x[1][2]*
195944004:   84:                        src_ptr[(row+1-1) * WIDTH + (col+2-1)]);
391888008:   85:            accum_x += (sobel_kernel_x[2][0]*
195944004:   86:                        src_ptr[(row+2-1) * WIDTH + (col+0-1)]);
391888008:   87:            accum_x += (sobel_kernel_x[2][1]*
195944004:   88:                        src_ptr[(row+2-1) * WIDTH + (col+1-1)]);
391888008:   89:            accum_x += (sobel_kernel_x[2][2]*
195944004:   90:                        src_ptr[(row+2-1) * WIDTH + (col+2-1)]);
        -:   91:
        -:   92:            // calculate convolution with y kernel
195944004:   93:            accum_y += (sobel_kernel_y[0][0]*
        -:   94:                        src_ptr[(row+0-1) * WIDTH + (col+0-1)]);
195944004:   95:            accum_y += (sobel_kernel_y[0][1]*
        -:   96:                        src_ptr[(row+0-1) * WIDTH + (col+1-1)]);
195944004:   97:            accum_y += (sobel_kernel_y[0][2]*
        -:   98:                        src_ptr[(row+0-1) * WIDTH + (col+2-1)]);
195944004:   99:            accum_y += (sobel_kernel_y[1][0]*
        -:  100:                        src_ptr[(row+1-1) * WIDTH + (col+0-1)]);
195944004:  101:            accum_y += (sobel_kernel_y[1][1]*
        -:  102:                        src_ptr[(row+1-1) * WIDTH + (col+1-1)]);
195944004:  103:            accum_y += (sobel_kernel_y[1][2]*
        -:  104:                        src_ptr[(row+1-1) * WIDTH + (col+2-1)]);
195944004:  105:            accum_y += (sobel_kernel_y[2][0]*
        -:  106:                        src_ptr[(row+2-1) * WIDTH + (col+0-1)]);
195944004:  107:            accum_y += (sobel_kernel_y[2][1]*
        -:  108:                        src_ptr[(row+2-1) * WIDTH + (col+1-1)]);
195944004:  109:            accum_y += (sobel_kernel_y[2][2]*
        -:  110:                        src_ptr[(row+2-1) * WIDTH + (col+2-1)]);
        -:  111:
        -:  112:            // calculate total magnitude and clamp to 255
391888008:  113:            unsigned int total = sqrt(accum_x*accum_x + accum_y*accum_y);
195944004:  114:            dst_ptr[row * WIDTH + col] = total > 255 ? 255 : total;
        -:  115:        }
        -:  116:    }
```

Sam Weismuller
WES237B
8/19/2022

Neon:

```
    27997:  151:    for(int row = 1; row < HEIGHT-1; row++)
        -:  152:    {
391902006:  153:        for(int col = 1; col < WIDTH-1; col++)
        -:  154:        {
        -:  155:            // convert input rows to int16x8_t
195944004:  156:            uint8x8_t row1_char = vld1_u8(src_ptr + ((row-1) * WIDTH + (col-1)));
195944004:  157:            uint8x8_t row2_char = vld1_u8(src_ptr + ((row) * WIDTH + (col-1)));
195944004:  158:            uint8x8_t row3_char = vld1_u8(src_ptr + ((row+1) * WIDTH + (col-1)));
        -:  159:
        -:  160:            int16x8_t row1 = vreinterpretq_s16_u16(vmovl_u8(row1_char));
        -:  161:            int16x8_t row2 = vreinterpretq_s16_u16(vmovl_u8(row2_char));
        -:  162:            int16x8_t row3 = vreinterpretq_s16_u16(vmovl_u8(row3_char));
        -:  163:
        -:  164:            // combine rows 1 and 2 into one int16x8_t
        -:  165:            int16x8_t r_12 = vcombine_s16(vget_low_s16(row1), vget_low_s16(row2));
        -:  166:            // put two copies on row 3 in one int16x8_t to calculate x and y kernel in one operation
        -:  167:            int16x8_t r_33 = vcombine_s16(vget_low_s16(row3), vget_low_s16(row3));
        -:  168:
        -:  169:            // calculate x kernel accumulation for rows 1 and 2
        -:  170:            int16x8_t accum_x12 = vmulq_s16(k_x12, r_12);
        -:  171:            // calculate y kernel accumulation for rows 1 and 2
        -:  172:            int16x8_t accum_y12 = vmulq_s16(k_y12, r_12);
        -:  173:            // calculate x and y kernels accumulation for row 3
        -:  174:            int16x8_t accum_x3y3 = vmulq_s16(k_x3y3, r_33);
        -:  175:
        -:  176:            // combine accumulations for rows 1, 2, and 3
391888008:  177:            int accum_x = vaddvq_s16(accum_x12) + vaddv_s16(vget_low_s16(accum_x3y3));
391888008:  178:            int accum_y = vaddvq_s16(accum_y12) + vaddv_s16(vget_high_s16(accum_x3y3));
        -:  179:
        -:  180:            // calculate total magnitude and clamp to 255
391888008:  181:            unsigned int total = sqrt(accum_x*accum_x + accum_y*accum_y);
195944004:  182:            dst_ptr[row * WIDTH + col] = total > 255 ? 255 : total;
        -:  183:        }
        -:  184:    }
```

In both cases, the data access operations happened at half the rate of the rest of the loop. This implies that the compiler noticed the sequential nature of these memory accesses and unrolled them in order to batch them together, separating out expensive I/O operations from the faster register manipulation instructions.

In the Neon implementation, gcov also shows the degree to which the compiler removef the overhead of using the Neon instructions in C, compressing multiple lines of code into a single initialization line.

Part 3:

1. The __global__ flag indicates that a function is to be run on the GPU.

2. “foo<<4,32>>(out, in1, in2)” indicates that function “foo” will be run on the GPU in a grid of 4 thread blocks (essentially a collection of threads) that each contain 32 individual threads. If the arguments out, in1, and in2 are pointers they will need to point to GPU memory. “foo” can use variables like “threadIdx.x” and “blockIdx.x” to operate on different parts of the input data so they don’t do redundant work.