Xinyu Wu

xwu3

15418 Assignment 1

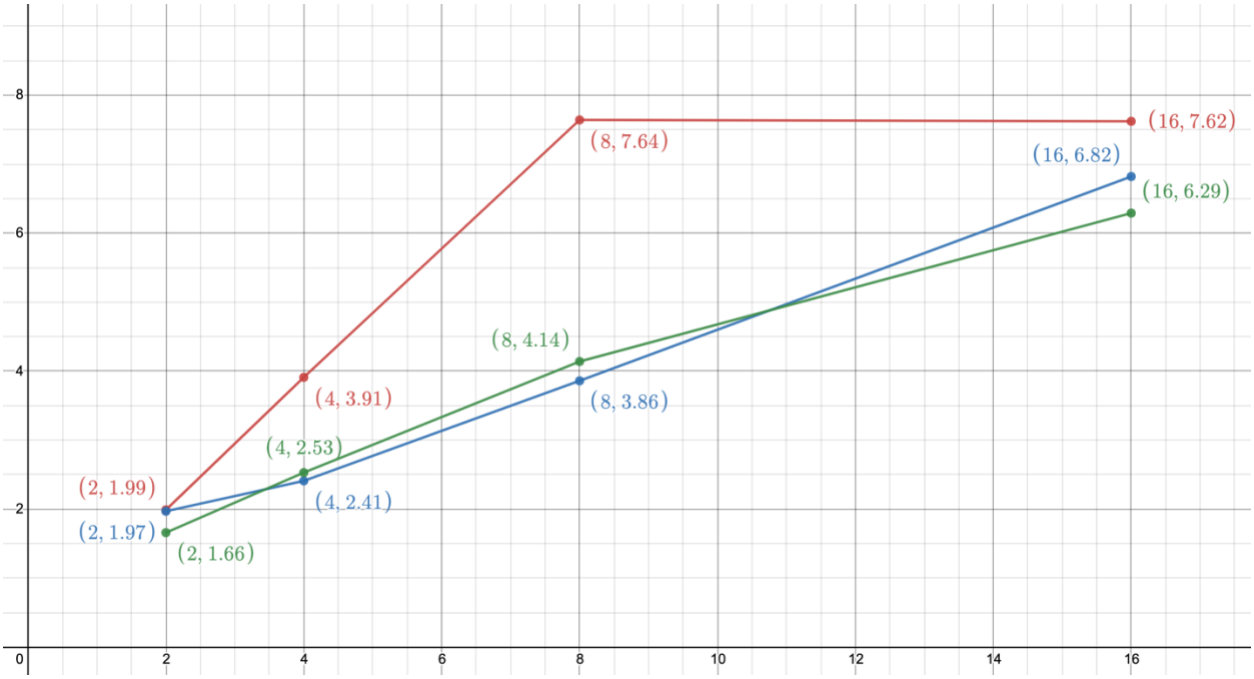
Problem 1

1. Graph of speedup as a function of cores:

(red: view0;

blue: view1;

green: view2)



- The speedup is somewhat linear, but not linear in terms of having a y-intercept of 0. However we can tell that the slope of speedup decrease a lot when we change from 8 threads to 16 threads. This is because we only have 8 cores, and using 16 threads will not increase its speedup by a lot anymore.
- It is not too linear because we have overhead everytime we use more cores.
- After I inserted timing code in the threads, the average times are below:

# of cores	Time (ms)
2	0.1293

4	0.0663
8	0.0337

- We can see that the time does decrease by approximately but not exactly a factor of 2, which supports the hypothesis.

2. Approach for maximizing speedup

Note that the speedup was limited by the fact that work was not evenly distributed among all the threads, therefore in many cases, many threads that are done with the work need to stop and wait for those threads which have larger jobs to finish. Therefore, in my approach, I used interleaving assignment rather than the blocked assignment approach previously. In this interleaving assignment approach, the assignment of amount of work between threads are more evenly distributed, and my speedup was about 7.61x for view 1 using 8 threads, and 7.45x for view 2 using 8 threads.

Problem 2

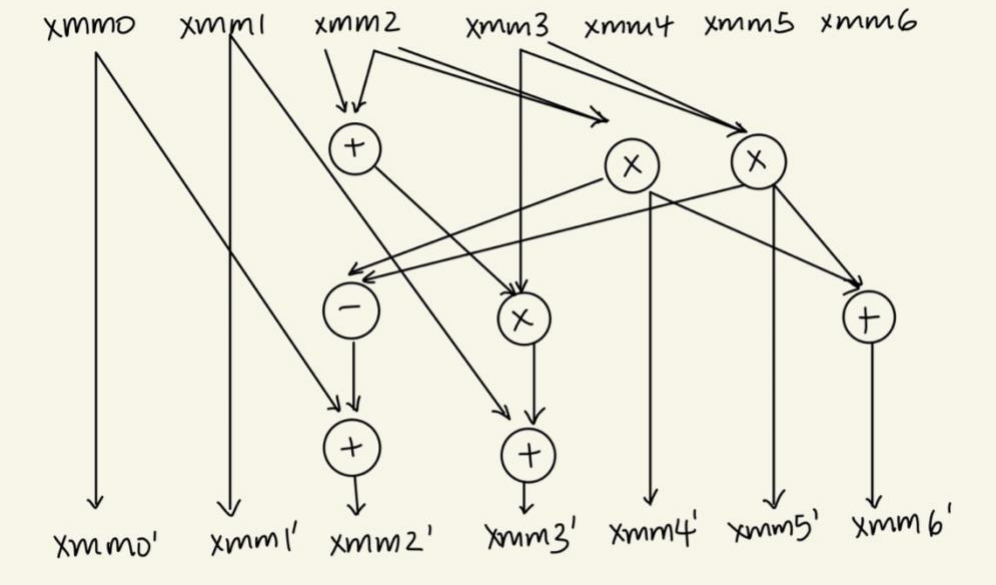
For 10000:
For vector width = 2: vector utilization = 78.703469%
For vector width = 4: vector utilization = 77.587343%
For vector width = 8: vector utilization = 77.191950%
For vector width = 16: vector utilization = 77.146414 %
For vector width = 32: vector utilization = 77.087252%

Vector utilization generally decreases as W goes up.
Because we can have more leftover in the last vector, meaning more empty lanes that are not operating.

The total vector instructions decrease as W goes up.
Because more data are operating at the same time in the same vector.

Problem 3

1.
```
        # This is the inner loop of mandel_ref
        # Parameters are passed to the function as follows:
        #   %xmm0: c_re
        #   %xmm1: c_im
        #   %edi:  count
        # Before entering the loop, the function sets registers
        # to initialize local variables:
        #   %xmm2: z_re = c_re
        #   %xmm3: z_im = c_im
        #   %eax:  i = 0
.L123:
        vmulss  %xmm2, %xmm2, %xmm4 # xmm4 = xmm2^2
        vmulss  %xmm3, %xmm3, %xmm5 # xmm5 = xmm3^2
        vaddss  %xmm5, %xmm4, %xmm6 # xmm6 = xmm4 + xmm5
        vucomiss        .LC0(%rip), %xmm6 # compare xmm6 and 4.f,
                                          # correspond to line if (z_re * z_re + z_im * z_im > 4.f)
        ja      .L126               # if xmm6 > 4.f break
        vaddss  %xmm2, %xmm2, %xmm2 # xmm2 *= 2
        addl    $1, %eax            # i++
        cmpl    %edi, %eax   # Set condition codes for jne below, compare i and count
        vmulss  %xmm3, %xmm2, %xmm3 # xmm3 = xmm2 * xmm3
                                    # correspond to line float new_im = 2.f * z_re * z_im;
        vsubss  %xmm5, %xmm4, %xmm2 # xmm2 = xmm4 - xmm5
                                    # correspond to line float new_re = z_re * z_re - z_im * z_im;
        vaddss  %xmm3, %xmm1, %xmm3 # xmm3 += xmm1
                                    # correspond to line z_im = c_im + new_im;
        vaddss  %xmm2, %xmm0, %xmm2 # xmm2 += xmm0
                                    # correspond to line z_re = c_re + new_re;
        jne     .L123       # if i != count, jump to .L123 and loop again
```

2.


3. According to the data flow chart, the critical path consists of three operations, say +, x, + going from the path from xmm2 to xmm3'. Since we know that both operations of

floating point addition and multiplication takes 4 clock cycles, the latency bound of mandel_ref would be 4+4+4 = 12 cycles/iteration.

The measured performance is 12.99 cycles/iteration, showing that the calculation is basically correct, except that in the actual performance, there are some other possible overheads so that the actual latency is a bit larger.

4. We have a total of 4 functional units for floating point addition and multiplication, and we have a total of 8 operations. This means that our throughput is 8/4 = 2 cycles/iteration.

   The measured performance varies from 13 cycles to 5 cycles, with most of the values fall between 5 and 6. This means that the throughput bound is not the bound that is limiting the performance for parallel versions. Rather, the parallel versions are still limited by latency.

5. The compiler can start using unit 5 which was not used before for floating point operation additions.

Problem 4

4.1

- the expected speedup is 8x since we have 8 cores. The measured speedup for only ISPC is about 5.09x. The parts that present challenges is the part of the image where calculations of Mandelbrot take very different amount of time. This means that in the parallelism we have with SIMD execution, we have very different performance for each hardware component, meaning that many components need to wait for others to complete. This is confirmed by the fact that when we run view 1 and view 2, the performance is worse.

4.2

ISPC + parallelism gives about 7.64x speedup. This is about 1.5x times faster than only ISPC, but ISPC + parallelism still varies over different views. It is worth the effort when we can cut dependencies between loops by calculating for two indices in the same iteration.

4.3

- When running the version of mandelbrot_ispc that does not partition that computation into tasks on view 1, the speedup is 4.41x for ISPC+parallelism.
Running mandelbrot_ispc with tasks on view one, the speedup is 7.16x.
- I prefer wide blocks, because there is more locality if we perform calculations within a wide block. I used 700 for block width and 100 for block height. The final speedup is 27.17x.

Problem 5

1. The speedup for a single CPU core is 4.70x. The speedup due to multi-core parallelism is 33.44x.
2. For my good input data, I spread the work evenly among the hardware threads, and I also chose the data very close to 2 to get as many iterations as we can have to perform more work. When the amount of work is evenly distributed, less and less of the threads will be waiting for other threads to complete. Rather, all threads will be working for the approximately same amount of time.

   The final speedup is 6.80x for ISPC only and 49.75x for task ISPC. My modification improves both SIMD speedup and multi-core speedup, because now they have roughly same amount of work to do and would not be idle waiting for other threads/cores to complete.
3. For my bad input data, I intentionally assign a very low iteration number (which is 1.f according to our chart) once every per 8 values, since we have 8 cores (meaning that we are performing 8 executions at one time). This way, all of the other threads/cores need to stop and wait for this single thread/core to finish, so all of them can proceed.

   The final speedup is 0.86x for ISPC only and 6.22x for task ISPC. My modification improve for multi-core speedup from ISPC only, because for multicore, each core has more execution that takes longer time. So generally the speedup is better than ISPC only.