Hossein Souri

Project 1

EN 601.320/420/620
September 30, 2020

Loop Efficiency

a. Plot the runtime as a function of filter size (don't forget to label axes and units).

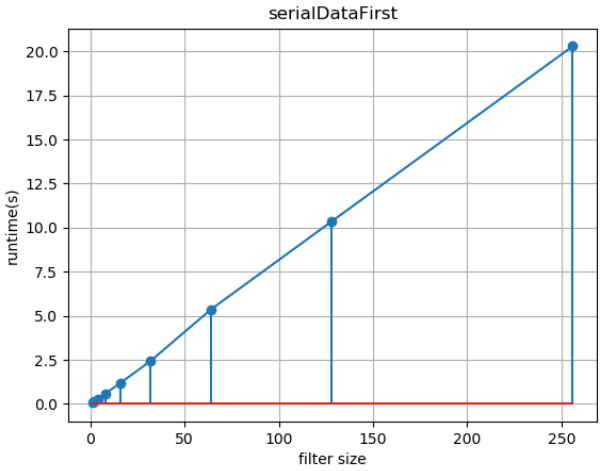

Figure 1: serialDataFirst runtime

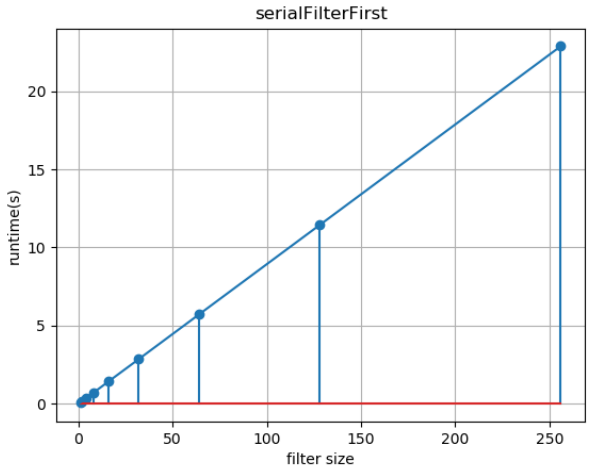

Figure 2: serialFilterFirst runtime

b. Normalize the runtime to the filter length and plot the normalized runtime, i.e. number of operations per second.

Answer: Please see figures 3 and 4.

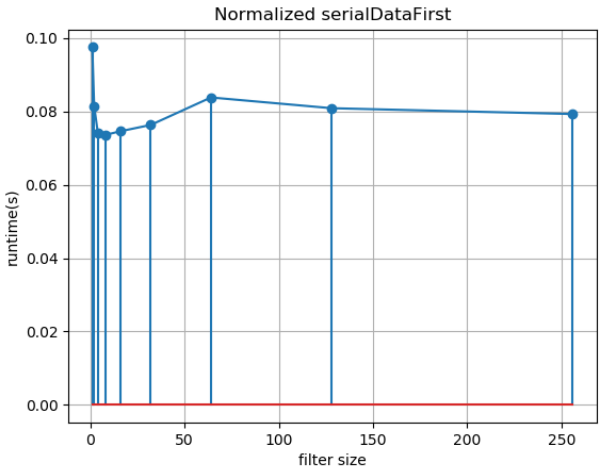

Figure 3: Normalized serialDataFirst runtime

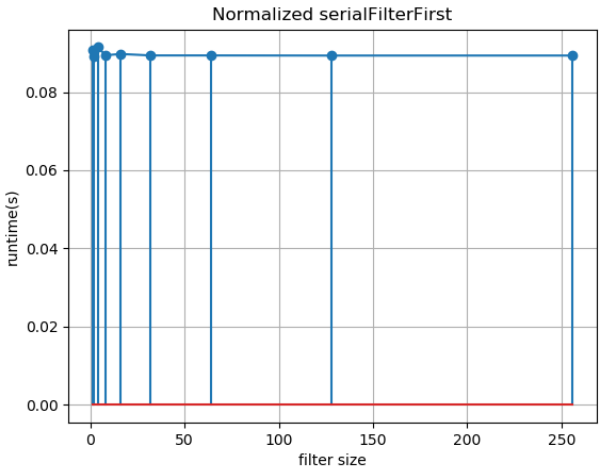

Figure 4: Normalized serialFilterFirst runtime

c. Is it more efficient to have the data or the filter in the outer loop? Why?

Answer: The data is better to be in the outer loop because as we can see in the previous figures the runtime for the case that data is in the outer loop is relatively better than the other case where the filter is in the outer loop. The reason is that the memory access is better in that case. In short, memory access patterns resulting in better cache hit rates.

How does the relative performance scale with the filter size? Explain the trend.

Answer: Considering relative performance to be the performance of one implementation divided by the performance of another, in general data first function is faster than the filter first. But when we deal we small filter size (1 and 2), the performance of the filter first is better but as filter size increases we see that the data first is better. After a specific filter size (here 128) we see that the relative performance is constant.

1 Loop Parallelism

1.1 Speedup

a. Generate speedup plots for 1,2,4,8, and 16 threads. You will have to specify the number of threads to run using omp_set_num_threads().

Answer: The plots are shown in figure 5 and figure 6.

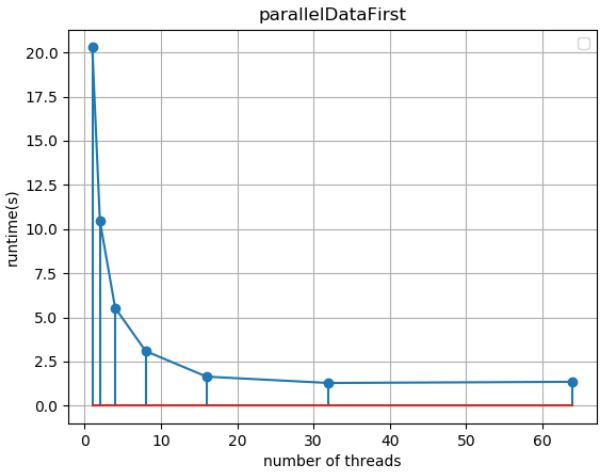

Figure 5: parallelDataFirst runtime

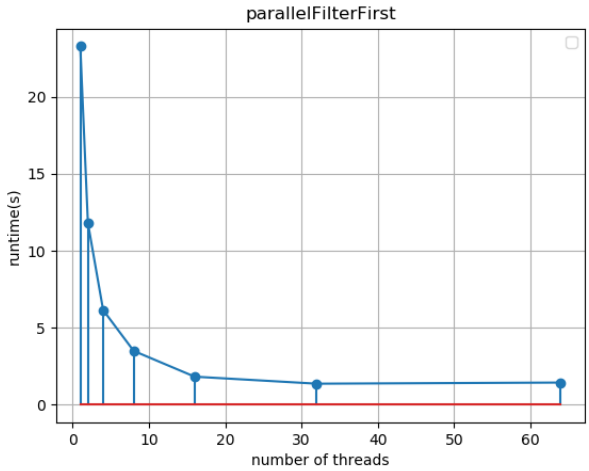

Figure 6: parallelFilterFirst runtime

b. Describe the results. What type of speedup was realized? When did speedup tail off? Why? What hardware did this run on and how did that influence your result?

Answer: Clearly we see that the speedup is exponentially growing. The reason is that in each step we double the number of thread which directly cause a double performance rate. We see a tale off when we have 32 thread or more. The reason is that my machine has 32 processors and having more than 32 threads does not improve the performance (My workstation has 8 cores, 2 threads per core, and 4 sockets which result in 32 processors).

1.2 Parallel performance

a. Which version of the parallel code was faster? Why?

Answer: According to my numbers the parallelDataFirst is faster. In this case the outer loop is to get a hold of much larger chunks of things that can be done in parallel. Therefore, each outer loop iteration represents a significant, parallel chunk of work which translate into a increase in speed.

b. Which was more scalable? Why?
Answer: The parallelDataFirst is more scalable since it performs better and also tails off less sharply as the dataset gets larger.

c. Estimate the value of p in Amdahl's law based on your results.

Answer: Based on Amdahl law:

$$S = \frac{1}{1 - p + \frac{p}{n}}$$

Here, n is the number of processors and S is the speedup. Therefore we can compute p using the following:

$$p = \frac{n - \frac{n}{S}}{n - 1}$$

$$S = \frac{T(1)}{T(n)} = \frac{20.289314}{1.647943} = 12.3658$$

Therefore:

$$p = \frac{16 - \frac{16}{12.37}}{16 - 1} = 0.9804$$

d. To what do you attribute any non-ideal speedup? To startup costs, interference, skew, etc? Explain your answer

Answer: Here the startup effect the speed up as we have multiple processors.

2 Loop Unrolling

Compare the relative performance of these functions on a filter of length 256 with the largest data size from part 2 and 8 threads.

Answer:
You can see the comparison in figures 7 and 8. In figure 8 I just used a larger data set to magnify the difference between the functions. We can see that like previous parts data first functions perform better and unrolling the outer loop has better performance.

a. Can you correlate the performance improvement with the fact that you unrolled the loop 8 times? Perhaps by instruction counting. Explain.

Answer: The performance improves by unrolling one loop even though the difference is not that much. And unrolling the loop doesn't improve the performance 8 times.

b. Can you conclude whether it is better to unroll the inner loop or outer loop? If yes, why?

Answer: I did the experiments multiple times and average over 100 experiments to have high confidence of the numbers. Based on the numbers unrolling the outer loop helps more and

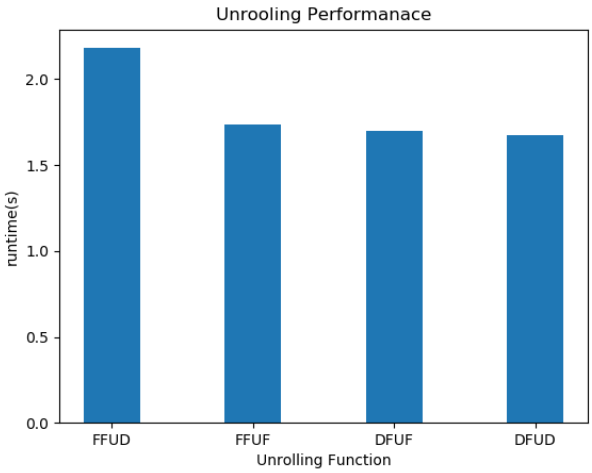

Figure 7: Unrooling Performanace

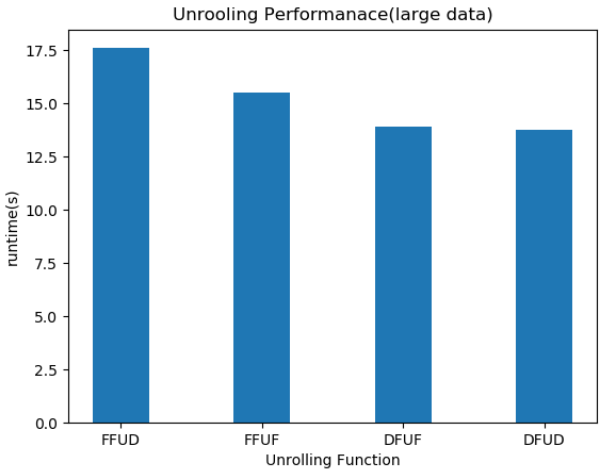

Figure 8: Unrooling Performanace for large data

performs better. The reason why unrolling the outer loop helps is that we have larger chunks of things that can be done in parallel.

3 Compiler Optimizations

For a filter of 256, 8 threads, and the same data size, compare the performance of the program running at compiler optimization level -O0 and -O3. What is the relative performance?

Answer: As you can see in the figure 9 the relative performance for the best parallel is:

$$\frac{1.824273}{0.713025} = 2.5584$$

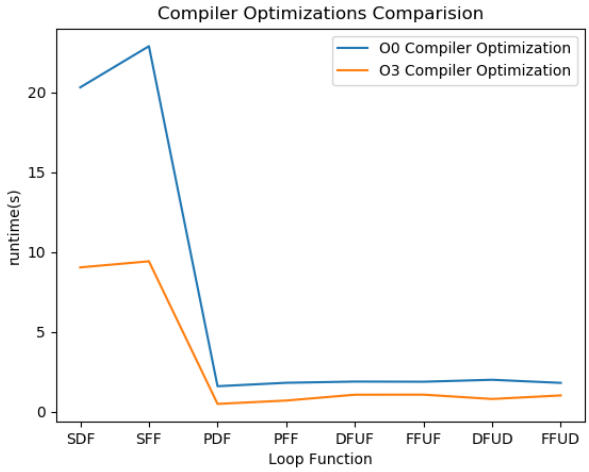

Figure 9: Unrooling Performanace

b. To what optimizations do you attribute the performance differences? gcc users might refer to https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html Explain.

Answer: -O3 turns on the many useful optimization flags related to loops. Here it turns on -floop-unroll-and-jam, -ftree-loop-vectorize, -fsplit-loops, -ftree-loop-distribution, and -fversion-loops-for-strides.