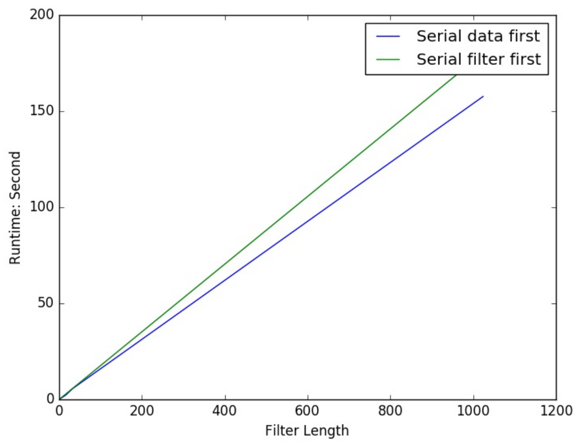
Loop Efficiency

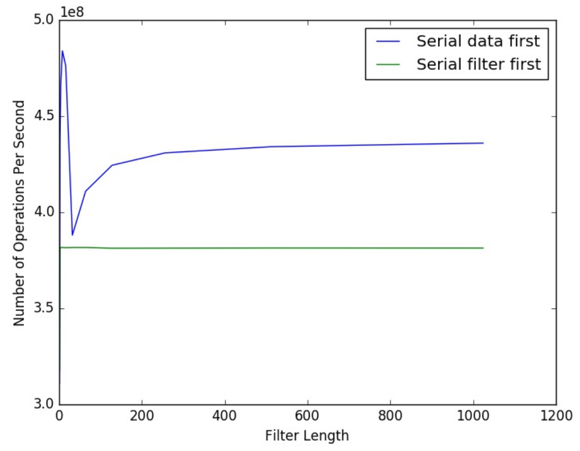
1.a: Plot the runtime as a function of filter size.

Answer:



1.b: Normalize the runtime to the filter length and plot the normalized runtime.

Answer:



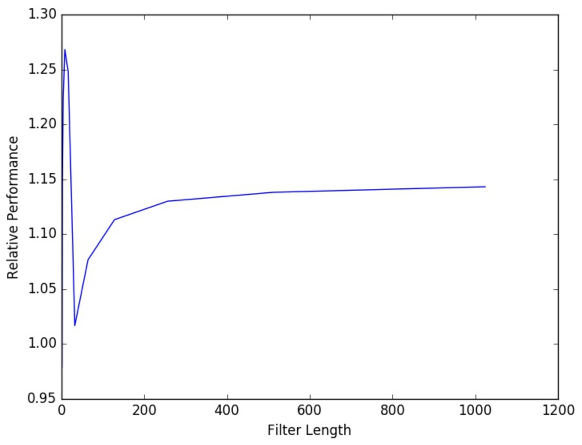
1.c: Is it more efficient to have the data or the filter in the outer loop? Why?

Answer:
From the graphs above it's easy to notice that when the data is in the outer loop, the program runs faster. My explanation for this situation is that having the data in the outer loop will

provide better cache utilization. Because the size of filter is much smaller than the size of data which means that it can be stored in the cache so that the processor doesn't have to often access it from memory. On the other hand, if the data is in the inner loop, the cache will be updated frequently which is inefficient.

1.d: How does the relative performance scale with the filter size?

Answer:

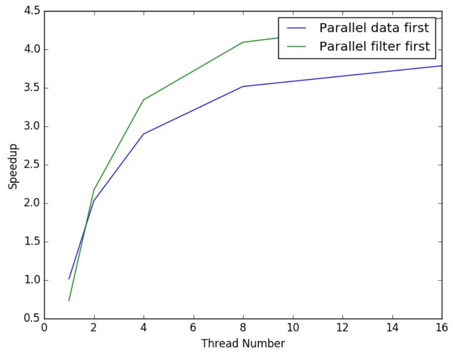


According to the graph above, we can notice that when the size of filter is small, with the increment of filter size, the relative performance of data first loop comparing to filter first loop is also improving. But when the filter size reaches a specific value, the relative performance drops which makes the advantage of data first loop not so obvious. This special value may be the size of cache since when the filter size is larger than the cache size, the data first loop also needs to update the cache frequently.

Loop Parallelism

2.a: Generate speedup plots for 1,2,4,8, and 16 threads.

Answer:

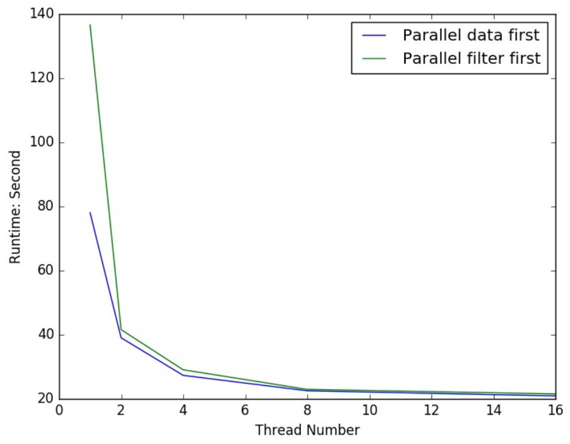


2.b: Describe the results. What type of speedup was realized? When did speedup tail off? Why? What hardware did this run on and how did that influence your result?

Answer:
According to the graph in 2.a, in the beginning the speedup increases with the number of threads. But when the number of threads becomes larger than 8, the speedup remains almost unchanged. The reason is that the CPU we use has 8 cores so using more than 8 threads will not make any significant difference.

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

Answer:



From the graph above we can know that the data first is a little faster because the cache utilization issue still exists even if we split the loop.

3.b: For the faster, estimate the value of p in Amdahl?s law based on your results for 1 to 8 threads.

Answer:
According to Amdahl's law $Speedup = \frac{1}{(1-p)+\frac{p}{s}}$, we have $p = \frac{1-Speedup}{\frac{Speedup}{s}-Speedup}$. In our case, $s = 8$ because we have 8 threads. From the graph in 2.a we can know that the speedup for 8 threads is about 3.5. Therefore $p = \frac{1-3.5}{\frac{3.5}{8}-3.5} \approx 0.816$.

3.c: To what do you attribute any non-ideal speedup? To startup costs, interference, skew, etc?

Answer:

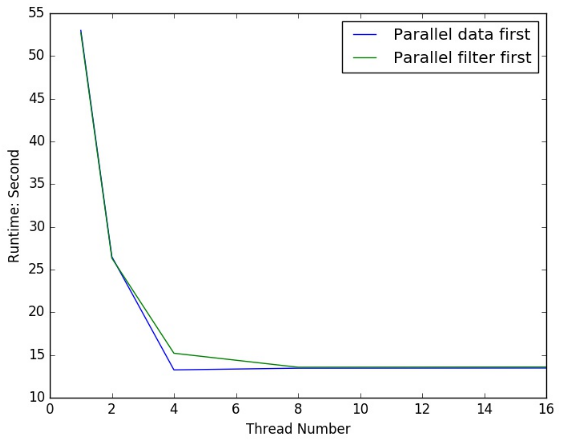
The reason for non-ideal speedup is the startup costs. We need to allocate resources such as data and filter for the program. And we also need time to create extra threads.

An Optimized Version

1: Consider the following code optimizations. For each, implement the optimized code and describe your result. For optimizations that improve run time, explain why. Similarly, provide an explanation if an optimization is ineffective.

Answer:

- a. We unroll the outer loop by a factor of 2 and the inner loop by a factor of 4. The result below shows that the performance is greatly improved comparing to the previous result. The reason is that this optimization reduces the number of iterations.



- b. We try both static scheduling and dynamic scheduling with different chunk sizes. But the performance is not improved. The reason is that this optimization is not suitable for our task. The static scheduling makes sure that each iteration of loop takes almost the same time. And the dynamic scheduling allows each iteration of loop takes different times. In our case, the data array and filter array can be distributed almost equally among threads so the custom scheduling makes no difference on the efficiency.