#### **Loop efficiency**

1. INSERT GRAPH]els of the cache.
2. The program runs faster when the data array is in the outside for-loop. This has to do with the cache. The data array is significantly longer (and thus takes up more memory) than the filter array. When the data array is on the outside and the filter array on the inside, the filter array is small enough to be stored on the top levels of the cache. This means less time is needed to access the next index of the filter array and thus the program runs faster. In addition, with the filter array stored in the top levels of the cache, a portion of the data array could be stored in the lower levels of the cache. Thus, part of the data array is also stored in the cache meaning less time is needed in the outside loop to access the next index of the data array. When the filter array is in the outside loop and the data array in the inside loop, it runs slower because the data array is too big to be stored in the cache. This means more time is taken in accessing the next index of the data array for each iteration of the loop.
3. Looking at the relative performance versus filter size for both programs, we see two different trends. For the filter first program (filter array in the outer loop), the number of operations per second is relatively constant barring the data plot for a filter size of 1. This makes sense because with the filter array on the outside and the data array in the inner loop, the cache is not big enough to hold all of the data array. Thus, changing the length of the filter size would have no effect on total runtime because the program is slowed down by the memory accesses to the next index of the data array. In contrast, the data first program (data array in the outer loop) yields a different trend. We see an increase in number of operations per second until the filter size hits 16. At a filter size of 32, there is a huge drop in performance and a less drastic increase in performance for the rest of the experiment (up to a filter size of 1024). This also makes sense because up to a filter size of 16, the filter array fully fits in the cache making access of each index very fast. When the filter size hits 16 and then 32, we see performance drop because the filter no longer completely fits in the cache and thus the program takes longer to access the next index of the filter array in the inner loop. We see the gradual increase in performance starting at a filter length of 32 because the cache is more completely filled with the filter array and thus the program is able to execute more operations per second.

#### 

#### **Loop Parallelism**

* 1. For both parallel programs, the speedup increases linearly up to four threads. By eight threads, both programs yield no speedup (horizontal line on the graph). This would be considered sub linear speedup. Looking at the hardware used for this experiment, we used a CPU that has 8 cores. Thus, we can expect that any number of threads greater than 8 would result in no increase in efficiency because the CPU would not have enough cores to handle that many threads. We see this in the graph. We would also expect linear speedup through 8 threads specified. Looking at the graph, it appears that the last four virtual cores do not perform as efficiently as the first four. That is why we see a flat line in efficiency for the filter first program and a smaller slope for the data first program.
  2. Looking at the graph, the y-axis represents the runtime of a serial implementation of the code divided by the parallel runtime for a given number of threads. Looking at the results, the filter first program shows a constant scale up for the first four threads. From there we see a significant drop-off in runtime. This is sub linear scale up. This makes sense because as mentioned above, the CPU seems to run most efficiently with a maximum of four cores specified thus as the input size (in this case filter length) is increased, the relative runtime remains constant. It appears again that the second four virtual cores are not as efficient as the first and thus we see a drop-off in runtime at 8 threads specified. We see an even greater drop-off at 16 threads because the CPU does not have enough cores to handle each thread, and thus the parallelism of the program cannot maintained for the increase in filter length. The data first program shows a slight increase in efficiency from one thread to four threads. This can be explained because the increase in filter length from 32 to 64 and 64 to 128 was not significant enough slow the overall runtime of the program and with the added number of threads, the program was able to run faster even with the increased input size.
  3. The data first implementation yielded faster results than the filter first implementation. This makes sense if we consider how the parallelism of each program processes the output array. With the data first program, the data array is split so that each thread receives a unique section of the total data array. Because the output array is based on the index of the data array, no sharing occurs and each thread reads from its section of the data array and writes to its specific section of the output array in parallel. In contrast, the filter first array uses threads that try to access the same part of the output array at the same time. This sharing causes the program to have to wait for one thread to complete before another can run causing the overall program to be slower than with a data first implementation.
  4. The non-ideal speedup can probably be attributed to startup costs. This is why we see a decrease in speed from four cores to eight cores. The increase in threads means more time was spent at startup to create the extra threads and thus the overall program runs slower than the ideal speedup because of the initial cost of creating the extra threads. The non-ideal scale up can probably attributed to interference when different parallel threads attempted to read from the same data area in memory. This would cause each thread to wait for those before it accessing the same area in memory causing the overall program to run slower and thus yield non-ideal scale up. Skew was probably not much of a factor as each parallel thread ran same loop and thus probably finished around the same time.

#### **An Optimized Version**

1. The most efficient implementation of an individual optimization was when taking advantage of the loop unroll optimization. This optimization reduces the loop overheads by completing two operations per each iteration instead of one. This was effective in cutting down on total runtime for the program. Combining the loop unroll with the loop collapse optimization led to the overall most efficient program. In another test, I optimized with a Loop Unroll optimization by a factor of four instead of two. This yielded a more efficient program than the original loop unroll optimization.
2. I also tested the loop collapse optimization. This optimization collapses nested for loops into a single for loop cutting down on total runtime of the loop. This optimization did not significantly decrease runtime of the program overall. In combining it with the unroll optimization however, the program was ran at its most efficient than either of the optimizations on their own. Loop fusion did not really apply to this program because it deals with two consecutive for loops and my program deals with a single nested for loop. It also appeared that loop fission and loop tiling did not completely apply to the program.