Russell Folk
CS 491/CS 521 Parallel Programming
Homework 1 Report

# Data Locality and Fine-Grained Parallelism

---

## Disclaimer

For these tests I changed the size of N to be 128 up from 64, this was to more readily see the effects of the various optimizations. Furthermore, all tests were run on the computer named koss in the grad lab. It is uncertain if I was the only user when these tests were run so results may vary.

## Optimize by Loop Permutation

The best results are determined by ordering the loops l, k, i, j which gives an average of 512.2 MFLOPS, up from the original implementation which was on average of 94 MFLOPS when running the default i, j, k, l ordering. The second best ordering is k, i, l, j which gives an average of 459.133 MFLOPS. The performance increase obtained with these orderings is 444.89% and 388.27% respectively. By far, as will be noticed throughout this report, correct ordering of the loop permutation gives the single best optimization available to be obtained.

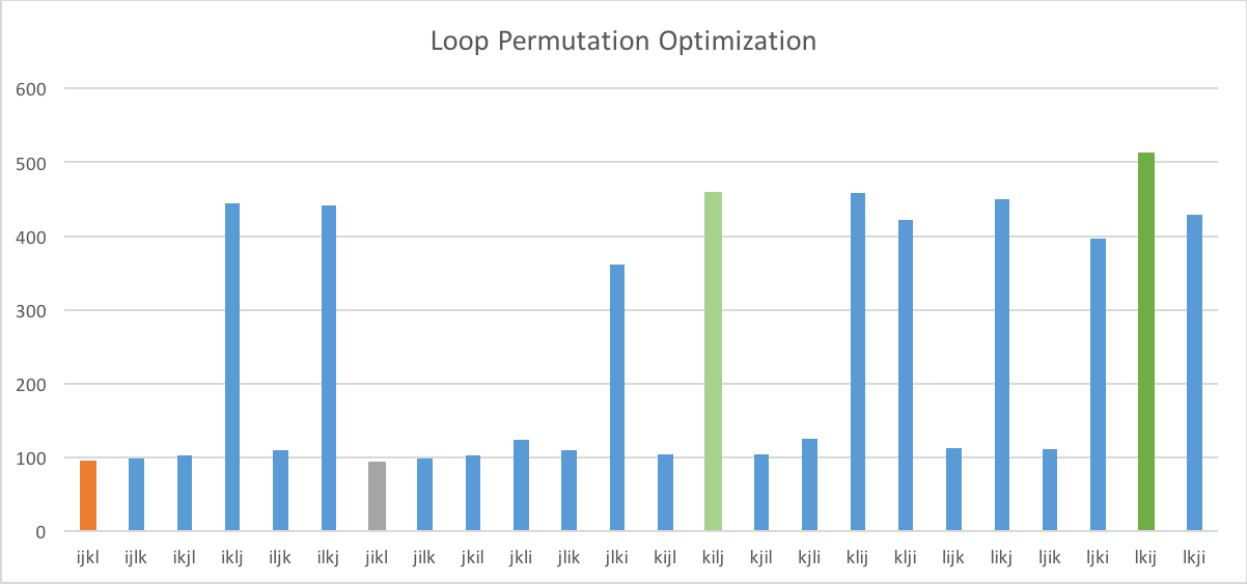


**FIGURE 1: LOOP PERMUTATION OPTIMIZATION**
**THE BAR IN ORANGE SIGNIFIES THE DEFAULT IMPLEMENTATION. THE BAR IN GRAY REPRESENTS A SUBPAR IMPLEMENTATION. THE BAR IN BOLD GREEN IS THE BEST PERFORMANCE OBTAINED. THE BAR IN LIGHT GREEN IS THE SECOND BEST PERFORMANCE OBTAINED THROUGH OPTIMIZATION.**

These tests were run in a very manual way by changing the order of the loops in emacs and executing all 24 permutations 3 times each and recording the results. I chose to use megaflops as the metric by which to judge improvement as runtimes I saw had more variation. The percentage improvement calculation is obtained by (newValue - oldValue)/oldValue*100.

I was mildly surprised that none of the permutations was significantly worse than the default configuration.

# Optimize by Loop Unrolling

---

## Disclaimer: loop unrolling permutations

I spent a lot of time testing loop unrolling including using various permutations of mixed unrolling. However, I did not observe significantly better unrolling combinations than what was achieved by only unrolling a single loop at once. E.g. unrolling the i-loop by 2 and unrolling the j-loop by 4 did not significantly improve performance. Thus, due to time constraints and lack of true differentiation, I did not include the results of these trials in this report.

The results obtained by optimizing via loop unrolling were very time intensive and not very useful over all. In actuality, only unrolling the j loop gave any significant improvements. At best I could only obtain about twice the performance (201.31%) by unrolling j 32 times. If only 16 unrolls were implemented only a 180.04% improvement was observed. *Please note that at this stage the original i, j, k, l ordering was maintained.* However, as referenced in figure 2, the gains to be had here are limited compared to loop permutation. We will investigate further how mixing the two works in favor.

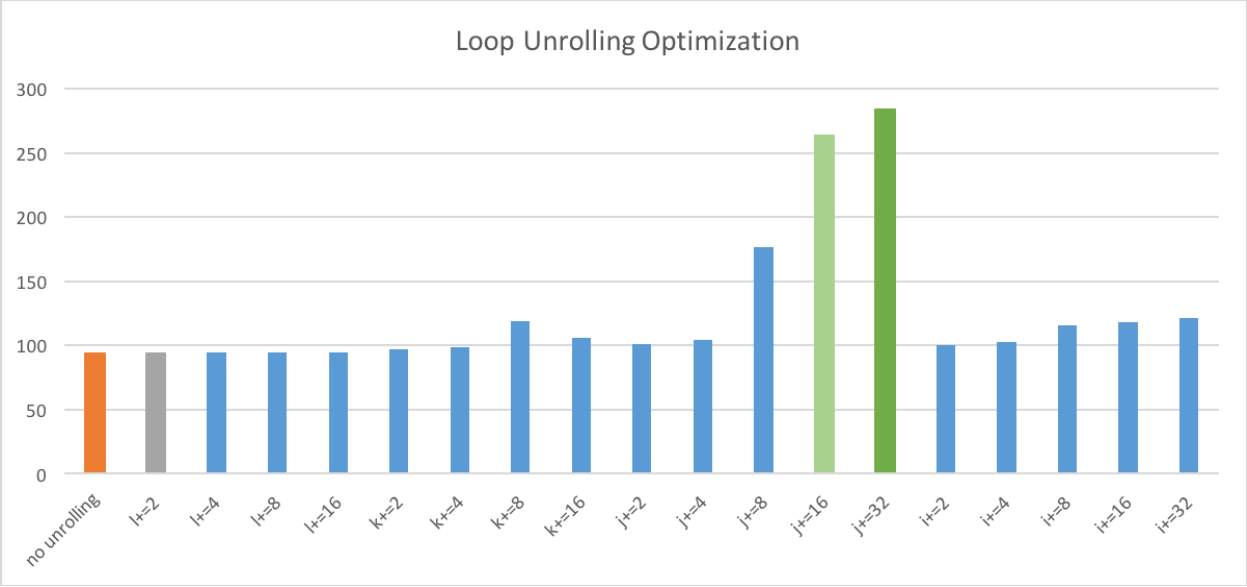


**FIGURE 2: LOOP UNROLLING OPTIMIZATION**
**THE BAR IN ORANGE SIGNIFIES THE DEFAULT IMPLEMENTATION. THE BAR IN GRAY REPRESENTS A SUBPAR IMPLEMENTATION. THE BAR IN BOLD GREEN IS THE BEST PERFORMANCE OBTAINED. THE BAR IN LIGHT GREEN IS THE SECOND BEST PERFORMANCE OBTAINED THROUGH OPTIMIZATION.**

These tests were run in a very manual way by changing the number of loop operations on a case by case basis then executing each series 3 times. I chose to use megaflops as the metric by which to judge improvement as runtimes I saw had more variation. The percentage improvement calculation is obtained by (newValue - oldValue)/oldValue*100. For all values included, refer to table 2 in the appendix.

# Optimize by Loop Blocking

---

## Disclaimer: loop blocking permutations

For these experiments I scripted a way to do nonuniform blocking, e.g., T = 1 for i, T = 2 for j, T = 0 for k, T = 0 for l. I ran these experiments over the course of a 12 hour period via the included `blocking_runner.sh` bash script. However, while interesting, I found the results to be rather unimpressive and due to the sheer amount of data, declined to enter the non-uniform results into this report.

I was rather surprised by two things in this set of experiments.

1. Having a T as small as 2 was enough to see an improvement. I had originally believed that the overhead that was generated by doubling the amount of loops would mean that we needed at least a T of 4 to see an improvement.
2. Having a T of 64 was still enough to observe a significant increase. As seen in figure 3, even with T = 64 we still observe that there is an improvement of about 150% over the original implementation. I had believed that due to the architecture of the system that we would have a loss of performance by 32 (especially as this is a 4 dimensional matrix).

Loop blocking, considering the minimal interaction required by the programmer, is a profitable venture. Not only does this allow for greater improvement than loop unrolling (the most programmer-intensive optimization) but it is scriptable to obtain the optimal value for T. As such, at best we obtained a 323.79% increase for T = 8 (the value I hypothesized would be best) and 280.77% increase for T = 32.

These tests were run using `blocking_runner.sh` with values for the various T's being 1, 2, 4, 8, 16, 32, and 64. I originally ran every permutation over night and after observing not very useful gains, I went with a uniform T and reran experiments 3 times each. I chose to use megaflops as the metric by which to judge improvement as runtimes I saw had more variation. The percentage improvement calculation is obtained by (newValue - oldValue)/oldValue*100. For all values included, refer to table 3 in the appendix.

# Optimization by combining methods

For these experiments I started with the single best loop permutation and applied other optimizations to the program. For loop unrolling I only chose to test the j and i loops, which in previous experiments proved to be the most promising. Finally, I tested all T values that were valid giving the amount of unrolling. For example, I could not unroll further than a given T. That is, I could not have T = 1 and j += 2 or j += 4.

The results tended to follow what I had speculated given the previous experiments. Referring to figure 4, we see that there are 14 combinations that give at least 500% improvement over the unoptimized implementation. As such, the single best performance I was able to obtain was using a combination of loop permutation (l, k, i, j) and loop unrolling (j+=16). By doing this I

obtained a 548.92% increase over the original code. By incorporating with this loop blocking with T = 16, I obtained the second best result which gave a performance increase of 545.67%.

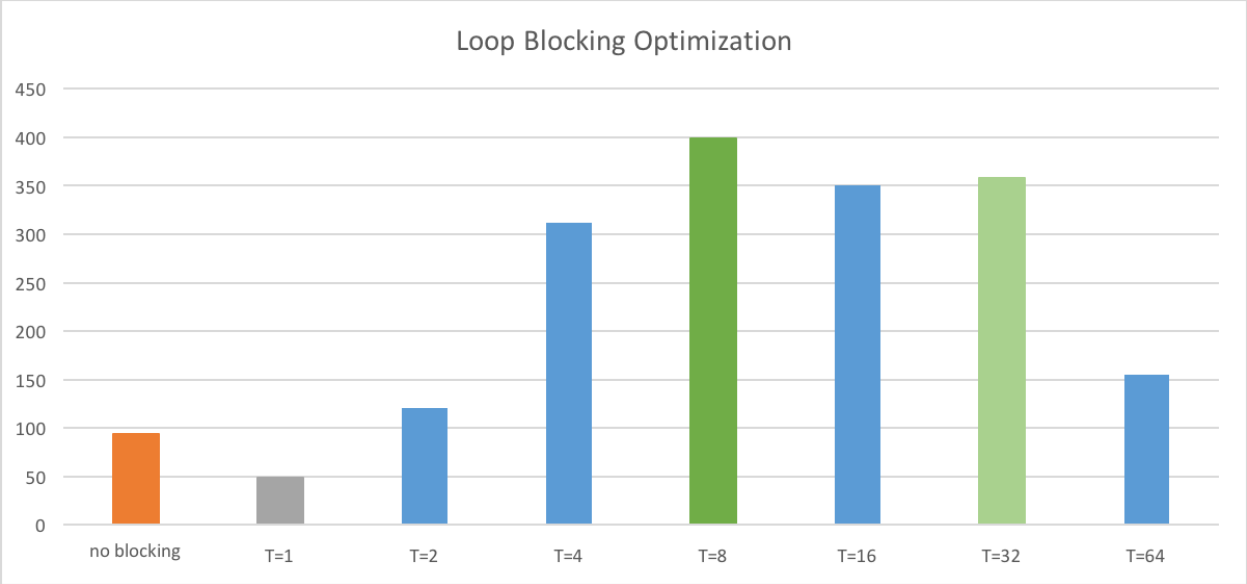


**FIGURE 3: LOOP BLOCKING OPTIMIZATION**
**THE BAR IN ORANGE SIGNIFIES THE DEFAULT IMPLEMENTATION. THE BAR IN GRAY REPRESENTS A SUBPAR IMPLEMENTATION. THE BAR IN BOLD GREEN IS THE BEST PERFORMANCE OBTAINED. THE BAR IN LIGHT GREEN IS THE SECOND BEST PERFORMANCE OBTAINED THROUGH OPTIMIZATION.**

I was rather surprised that loop blocking and loop permutations together (without the benefit of loop unrolling) did not score higher. I had falsely assumed that given the fact that individually they were the most promising optimizations, together they would be the most promising.

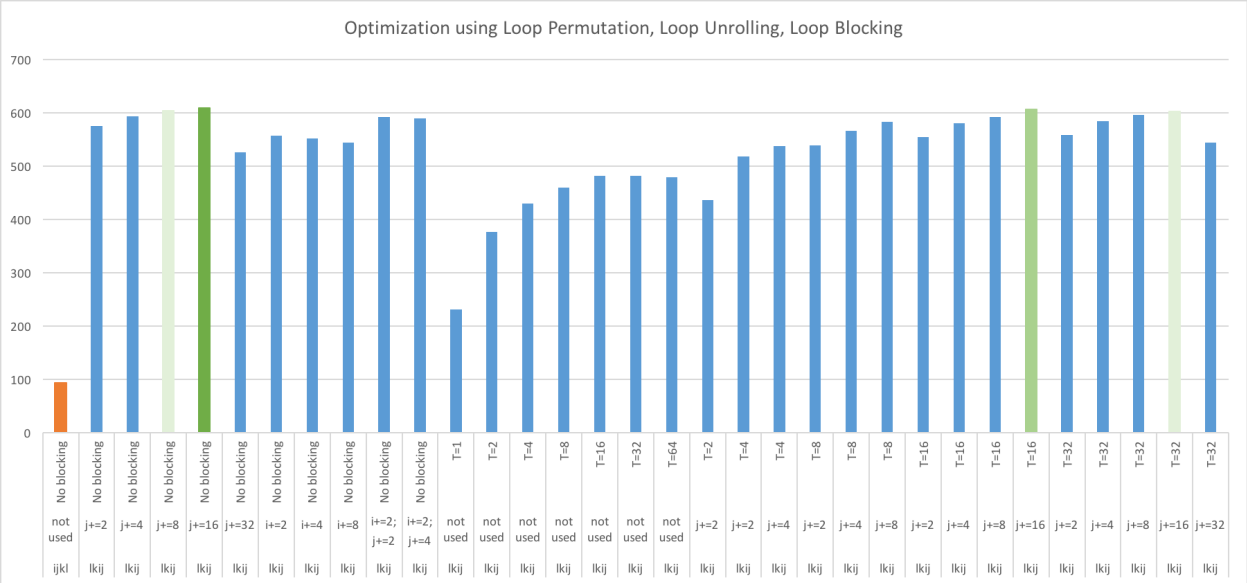


**FIGURE 4: MIXED LOOP OPTIMIZATION**
**THE BAR IN ORANGE IS THE DEFAULT IMPLEMENTATION. BARS IN GREEN SIGNIFY THE BEST PERFORMANCE OBTAINED LESSENING AS THE BOLDNESS DETERIORATES.**

I had rightly assumed that loop unrolling using j would be very effective because the original experiments matched that with the increases I saw by making j the final permutation in the loops. Furthermore, I was satisfied to see that my initial thoughts on unrolling were vindicated where j += 8 and j += 16 were the best options.

These results were obtained manually by implementing each of the combinations I was interested in and then culling later results if the trend was decreasing, such as it was when j += 32 (thus no j += 64). All results of run experiments can be found in table 4.

In conclusion, however, given the amount of time spent optimizing and the relatively small gains to be had, I believe that most of a programmers job should be spent in obtaining a good loop permutation and allow the compiler to optimize the rest. It is my understanding that in most cases, the compiler can rightly determine the correct amount of unrolling to perform. Blocking I suppose could be useful, but only when combined with a good amount of unrolling. Without adequate unrolling, unblocking with permutation is less effective than permutation alone.