CS 5220: Project 1 Initial Report

Group 018: Guantian Zheng (gz94), Stephen McDowell (sjm324)

1 Introduction

Our initial findings for this assignment are somewhat discouraging. We initially had a bit of trouble figuring out how all of the parts of this assignment fit together. At this time, we feel that we have a good understanding of how we are supposed to be performing the tasks expected of us, but not much performance has been achieved at this point.

1. The first stage of diagnostics was to begin examining how the different permutations of `i`, `j`, and `k` can have an impact on the speed of the basic matrix multiply.

2. The next natural piece to examine *at a high level* was comparing how `icc` and `gcc` compare with one another. During this phase we used the basic permutations to help us understand which was superior, but have not gone as far as changing the compilation flags – yet!

3. We then began examining how these permutations and best-suited compiler can affect the blocked strategy provided in the initial framework, in conjunction with adjusting the `BLOCK_SIZE` variable to see its effect on the blocking efficiency.

4. Where we are currently is trying to query the hardware to programmatically determine the optimal `BLOCK_SIZE`, as well as toying with the idea of transposing the `A` matrix in memory.

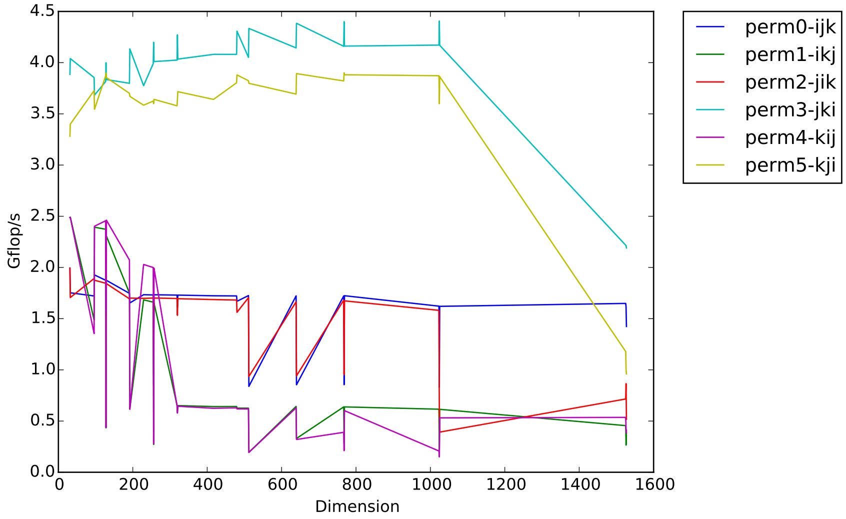
Part of the reason why we have not made as much progress as we would have liked to is because we only have two group members. This was not really made apparent to us until tonight.

2 Loop Permutations and Compilers

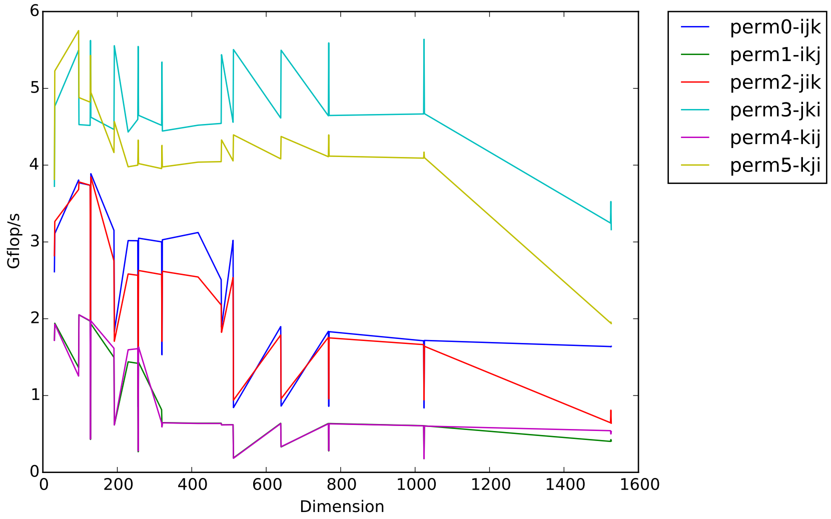
We experimented with different permutations of the three loop variables i, j and k. Apparently, j, k, i (starting from the outermost loop) gains the most advantage by reading in continuous blocks of matrix C and A. The loops are as follows:

```
for(j = 0; j < M; ++j) {
    for(k = 0; k < M; ++k) {
        double b_kj = B(k, j);
        for(i = 0; i < M; ++i) {
            C(i, j) += A(i, k) * b_kj;
        }
    }
}
```

When compiling with gcc, we get the following results:

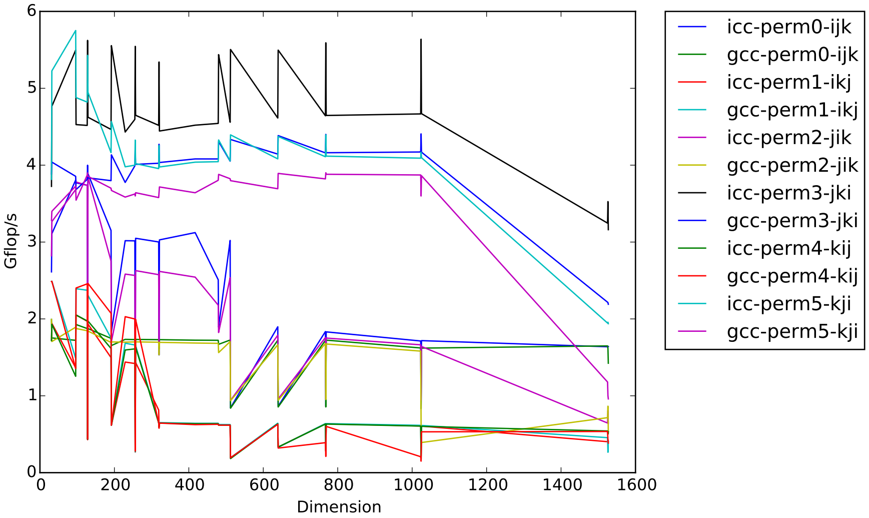


noting that the peak is somewhere between 4 and 4.5 Gflop/s. With the same code being compiled by icc, we get the following trend:



The difference in performance is not entirely that surprising, given how much of a proponent of icc Prof. Bindel is. The benchmarks shown above gave a lot of freedom to the compilers, though (the Makefile provided has optimization level 3 for both). Perhaps a better diagnostic would have been to turn off all optimizations and compare. But this seems to be a somewhat unrealistic goal, since the point of using the compiler in the first place is that it knows how to speak computer much better than we do.

Lastly, the two plots have been merged in one to demonstrate the superiority of `icc` when it comes to tuning the basic matrix multiply:



3 Tuning Blocked Matrix Multiply

Building on previous results, we decided to adopt the optimal `j-k-i` loop for per-block computation (`basic_dgemm`), while searching for an appropriate block size. We had a two-pronged approach here, which was to

1. Guess different block sizes and effectively perform a manual binary search:

   The results of this search, which will be graphed below, indicate that *for the cluster* it seems to be the case that `BLOCK_SIZE = 1024` is optimal.

   The testing for this has been done in the file `dgemm_blocked_perm.c`.

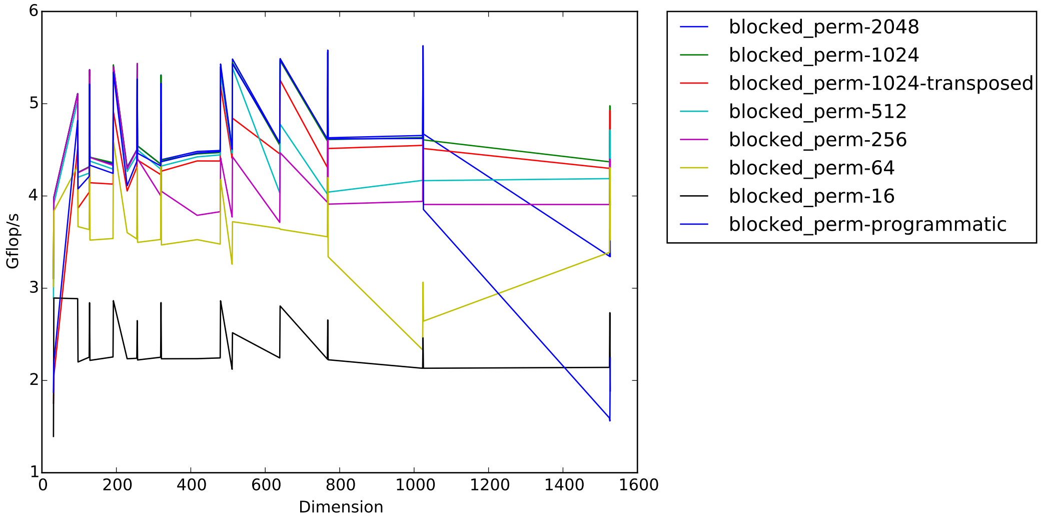
2. Try to programatically find the optimal size.

   This approach has hit somewhat of a dead-end, in that it cannot beat the manual guessing strategy tried first. The basic idea here is that we have 2 whole matrix blocks that need to fit into memory (`C` and `A`), and at each execution of the first nested loop (`k`) need only grab one element from `B`. We made the likely silly assumption that this would mean reading an entire page from memory, so in order to find the ideal `BLOCK_SIZE` we need to solve

   $$\sqrt{\frac{\texttt{L3\_SIZE}}{24}} = \texttt{BLOCK\_SIZE}$$

   In code, we have probably made a mistake or fallacious assumption somewhere, or maybe even are just calling the wrong methods with the wrong parameters. The plotted result below was with a `BLOCK_SIZE = 991`, where we hardcoded in 15MB L3 cache to the equation above. Perhaps this is somewhat reasonable, in the sense that it might make the most sense to take the result of the equation above and round up to the nearest highest power of 2 (which is 1024) since we are working with the cache.

See the plot on the next page:



4 Next Stages

We have been playing around with transposing things in memory, but generally think that we are not supposed to venture here – in order to perform such a task with the provided parameters we have to do something like:

```
double* bad_idea = (double *)(&(*A));
naive_transpose(M, bad_idea);
```

We also found that the transposed results really only start having an effect on large matrices since we are looking at an $O(n^2)$ operation in the naive case in order to transpose A, and then have to transpose it back. Asymptotically speaking, as the matrix multiply is $O(n^3)$ we will begin to see better results.

We plan to try new methods such as better copy optimization, compliler flags and annotations, and also want to explore the cache-oblivious variants of this project if possible.