CS 5220: Project 1 Final Report

Group 018: 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.

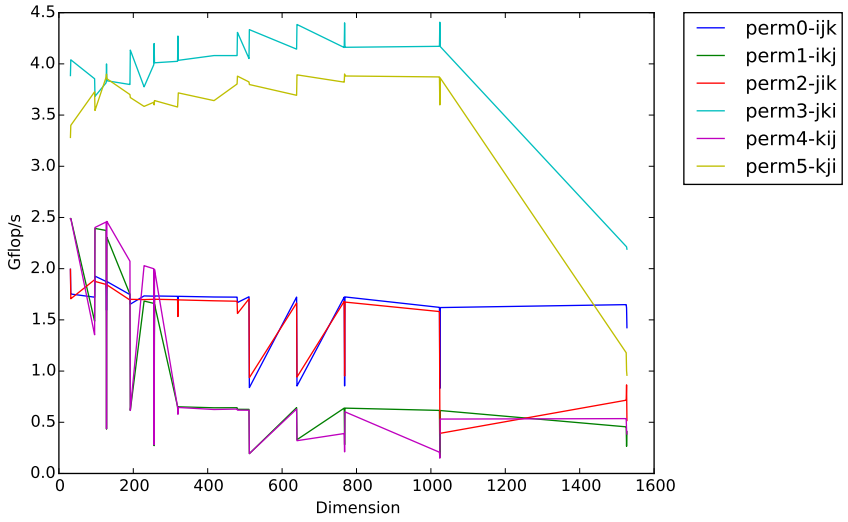
This was the state of affairs as of the initial report. Alas, the only thing that has changed is me trying to hand-vectorize the code. It seems I flew a little too close to the sun, and I never bring sunscreen...Sections 5 and 6 discuss what went on here, and why it failed so miserably.

2 Loop Permutations and Compilers

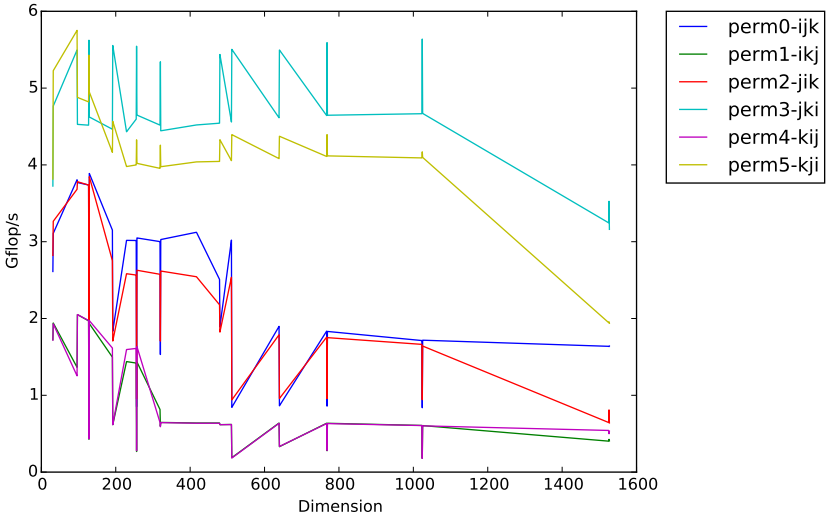
We experimented with different permutations of the three loop variables `i`, `j` and `k`. Apparently, `j, k, i` (starting from the outermost loop) gains the most advantage by reading in continuous blocks of matrix `C` and `A`. The loops are as follows:

```
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;
        }
    }
}
```

The code for producing these permutations can be found in `dgemm_basic-permutations.c`, where you can change `#define PERMUTATION` at the top to be 0, 1, 2, 3, 4, or 5 depending on which permutation you want to run. When compiling with `gcc`, we get the following results:



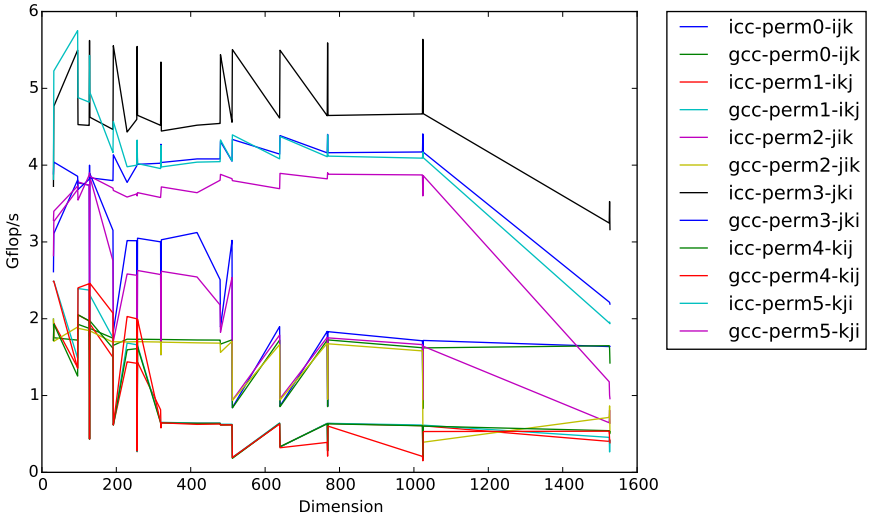
noting that the peak is somewhere between 4 and 4.5 Gflop/s. With the same code being compiled by `icc`, we get the following trend:



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

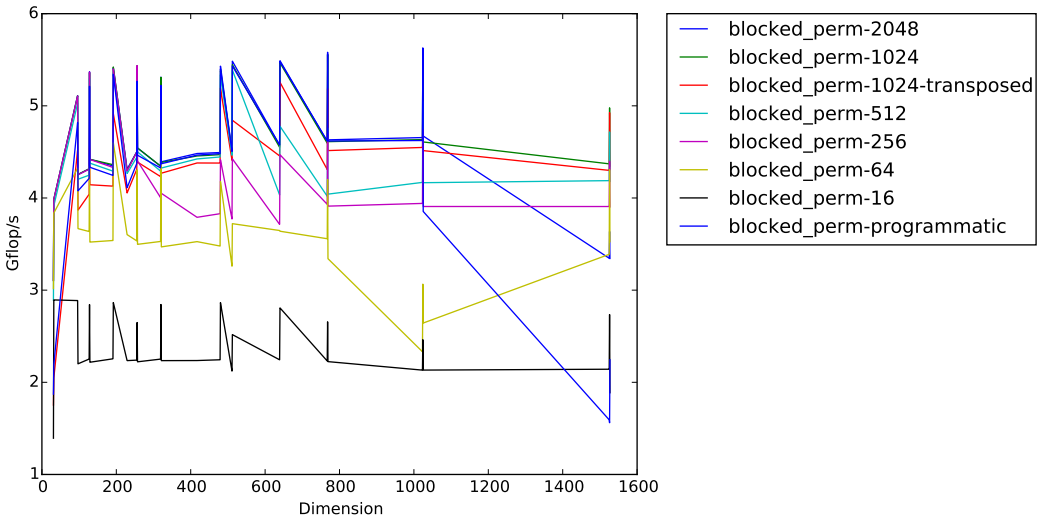
   $$\sqrt{\frac{\texttt{L3\_SIZE}}{24}} = \texttt{BLOCK\_SIZE}$$

   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.

   The code that was originally purposed for this was in `dgemm_basic-permutations.c`. However, inside of the top of the method `square_dgemm` at the bottom of the file you can see the corresponding `sysconf` calls that are attempting to acquire the right information. It seems that the calculation we seek to compute for number of per-matrix-block bytes is invalid. We tried computing having 2 full blocks plus one page (4096 bytes on

the cluster), but that produces equally disatisfying results. The computation in that code is assuming that we need to fit all three matrix blocks in cache at the same time, which is likely not the case.

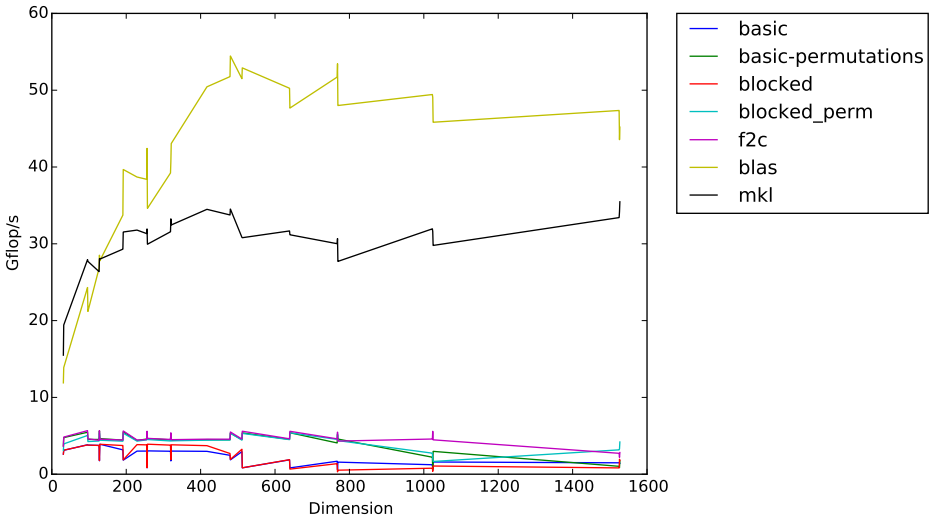
The corresponding plot:



The plot coloring reset at the bottom for `blocked_perm-programmatic`, this is the blue line that degrades *much* faster after a matrix dimension of 1024, for clarity.

4 Next Stages

As a final comparison of some of the methods described above, you can see from the benchmarks that our algorithm is really nothing to be proud of at this point:



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 (also located in `dgemm_blocked-ours.c`) 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.

5 Final Report

So I decided that I thought I would be cool and hand-vectorize the code. I did some bad research, and thought that the nodes could execute `AVX512` instructions, which is far from true. It is only the Phi boards that can do these. With the help of a truly wonderful guide released by Intel here:

https://software.intel.com/en-us/articles/benefits-of-intel-avx-for-small-matrices

I felt confident that I could adopt this strategy for the 512 bit case, since their code was for `float`, and ours is using `double`, it is effectively the same thing.
However, as verified from this extremely valuable resource of available instructions,

https://software.intel.com/sites/landingpage/IntrinsicsGuide/

it would need a little tweaking (e.g. there is no `_mm512_broadcast`, but instead you could use `_mm512_set1_pd(double a)`), but the logic would be the same. After writing this kernel, I realized I had no way to test it unless I could find a way to offload this to the Phi boards. I tried. But failed. It turns out that manually managing the memory for the Phi boards is rather complicated, and it seems that the `#pragma offload *` with associated `in, out, inout` is generally designed for less control than what I am used to (`CUDA`). But it also seems that, if I dig a little more, it can be done. If it means anything, the following were excellent resources that the class might benefit from:

1. https://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features

   This might actually be the most comprehensive resource out there, given the authoring entity.

2. http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/

   I already put this on Piazza. Very thorough.

3. https://cvw.cac.cornell.edu/MIC/

   This one actually may be a little outdated though...

4. https://portal.tacc.utexas.edu/documents/13601/901837/offload_slides_DJ2013-3.pdf/4b27de31-e8c4-4848-b100-c4b670b48148

   This was shown in class.

After a long while, I finally gave up on this, and just went back to working with the nodes. The two main problems I ran into were

1. Vectorizing and 8x8 kernel for doubles with 32 vector registers

   This worked, in the end, but I made a small but impactful mistake of adding up the wrong things. If the first element of a matrix is named a, we need to broadcast this into 2 registers in order to do the full 8x8 kernel. When you add things up, with how my code is, you need to add the left half top down and the right half top down, and I was doing left right.

   By halves, I mean that you treat the left half of the matrix as the first 256 bits, and the right half of the matrix as the last 256 bits.

   The relevant code exists in `dgemm_blocked_perm.c`.

2. Sub-blocking based off of an 8x8 kernel that requires exactly 64 values.

   The loop structure for this sub-blocking routine took me a *lot* longer than I expected. I'm not entirely convinced this was the best way to do it, but I only promised myself that I would get it functional. I was only able to debug the hand-vectorized code after I got this working.

At this time I must note that I didn't dream up the compiler flags used in the final result. I am currently working next to my partner for a different class, who gave me his compiler flags so that I could see what was going on.

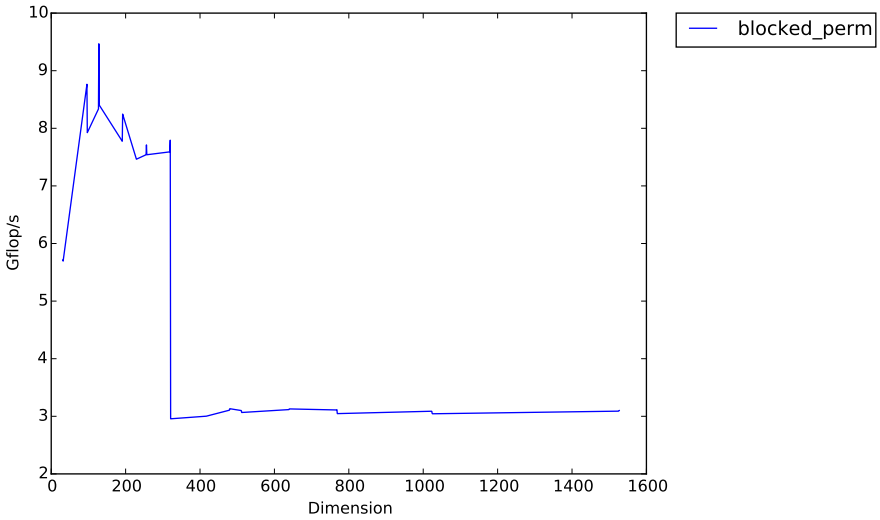
a) The flags:

   i) -O3
   ii) -fast
   iii) -axCORE-AVX2
   iv) -unroll-aggressive
   v) -mtune=core-avx2
   vi) -ipo
   vii) -xHost

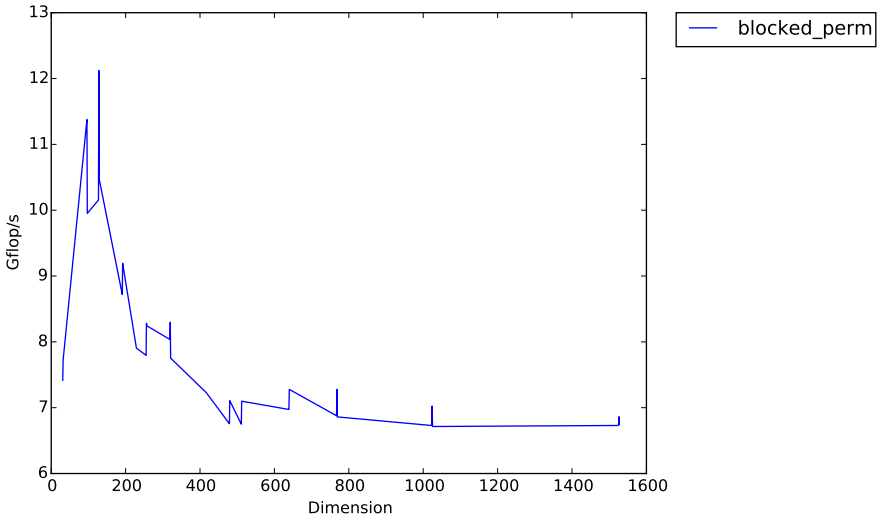
b) The group:

   Group 22, I will spare the explanation of the flags because you probably have already seen them, and their final report explains them as well.

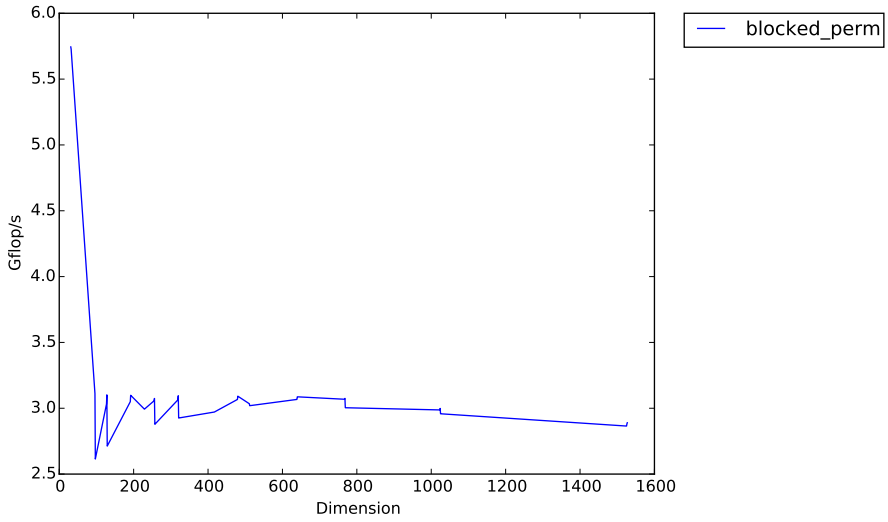
Finally, we have some results. My initial "smart" strategy to fit things according to `BLOCK_SIZE` described at the top of `dgemm_blocked_perm.c` seems to be invalid.
My hand-vectorized code, with that logic:



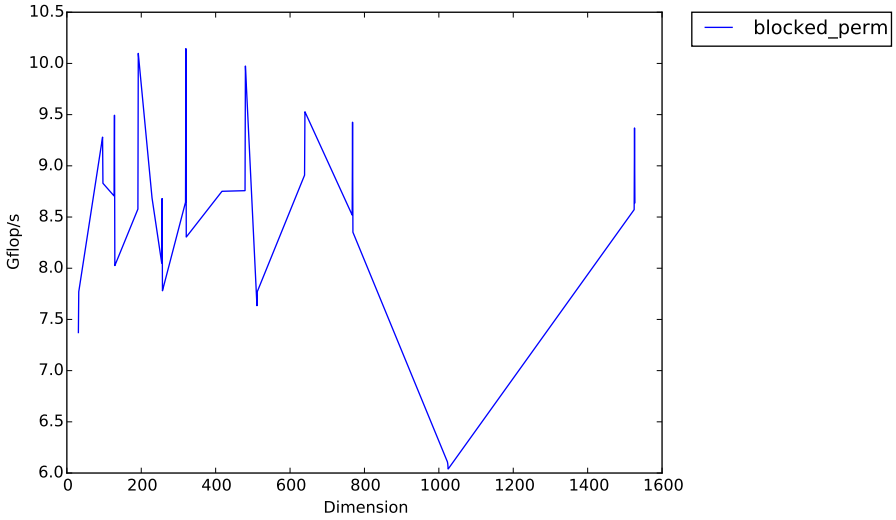
Well...that's depressing. That dropoff there is where my vectorized code actually starts being used. If the input matrix dimensions were less than `BLOCK_SIZE`, then I short-circuit and just do naive. So, if in the `basic_dgemm` we short-circuit always (and just do naive), then we get



Not so surprisingly, this does significantly better. Because I don't know how to vectorize well, only vectorize functionally.
So, if we change up the block-size to something smaller, say 64 just for shiggles, then we would expect to get much better performance. My vectorized code:



Guess not. For reference, this would be the intel vectorized code with a `BLOCK_SIZE` of 64:



Oh snap! The 1024 drop being somewhat expected here, since I must admit that a block size of 64 is really not a great choice.

6 Summary

So had I more time, I would have liked to experiment a lot more with vectorized code. Specifically, I don't take advantage of the fused-multiply-add capabilities, and very much would have liked to. Good thing we have another assignment!!! I realize full-well that the reason my vectorized code does so poorly is not because the 8x8 kernel is bad itself, but indeed because I am not very careful about how I am copying data over to the kernel matrices. This has a significant impact, which overwhelms any potential increase I may have had from vectorizing the code myself. If there even was one in the first place.

Better luck next time, champ.