# CS 5220 HW1 Report

Name: Santiago Lai, Cornell id: zl345, NERSC username: Zhengnan

In this report, I will discuss each of the optimizations I attempted and the effect of those optimizations. In the end, I will also describe the change of performance when running my optimized code on a different machine.

**1. Loop reordering & Choosing block size:**

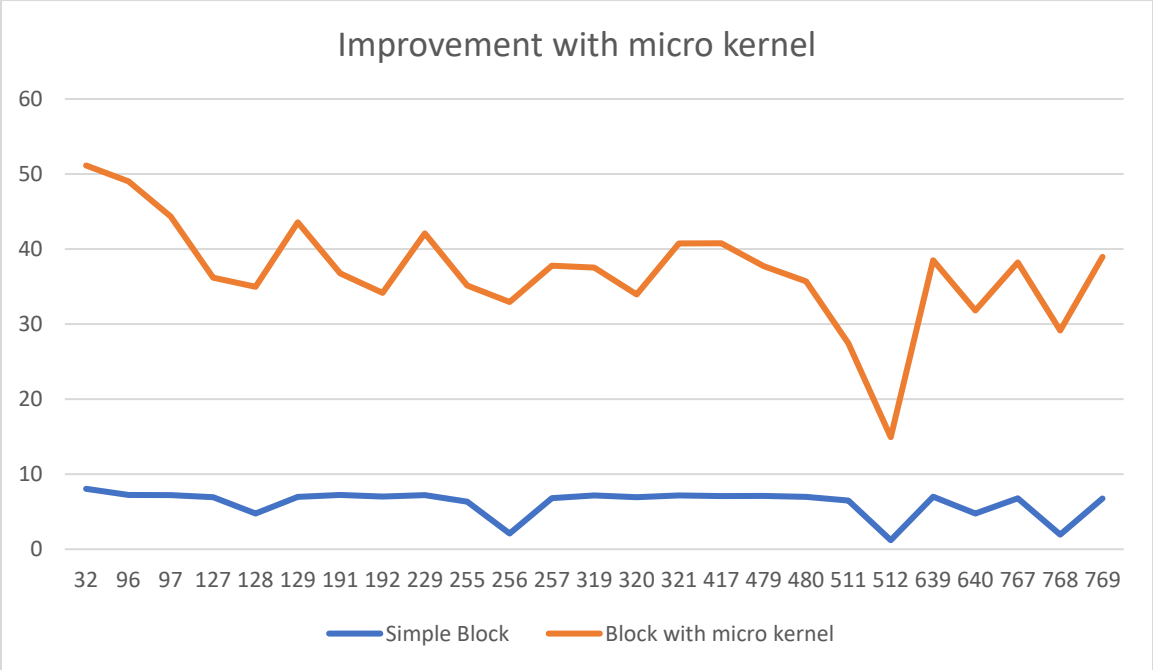
The first optimization that I attempted is loop reordering, with simply putting the k-loop inside to the outmost location. According to the handout, loop reordering might enhance the performance by potentially scheduling the operations in parallel. It turns out, though, that there is little improvement. I did not record the performance difference of implementing a single loop reordering for this reason.

I also considered the impact of block size on the performance of my code. Per handout, ideally the block size should just fit into a single L1/ L2 cache. Therefore, choosing a suitable block size might improve the performance as well. Since we have 32 KB for L1 cache and 512 KB for L2 cache (https://en.wikichip.org/wiki/amd/epyc/7763), the sizes S1 and S2 should satisfy 3S1^2 <= 2^15 and 3S2^2 <= 2^18, respectively. I solved the inequalities to get S1 = 64, S2 = 256, respectively, and used these sizes throughout my implementation.

**2. Micro Kernel (with SIMD):**

The second optimization that I tried is the micro kernel mentioned in recitation, paired with SIMD. Since Perlmutter has hardware support only for AVX 256, a vector can contain at

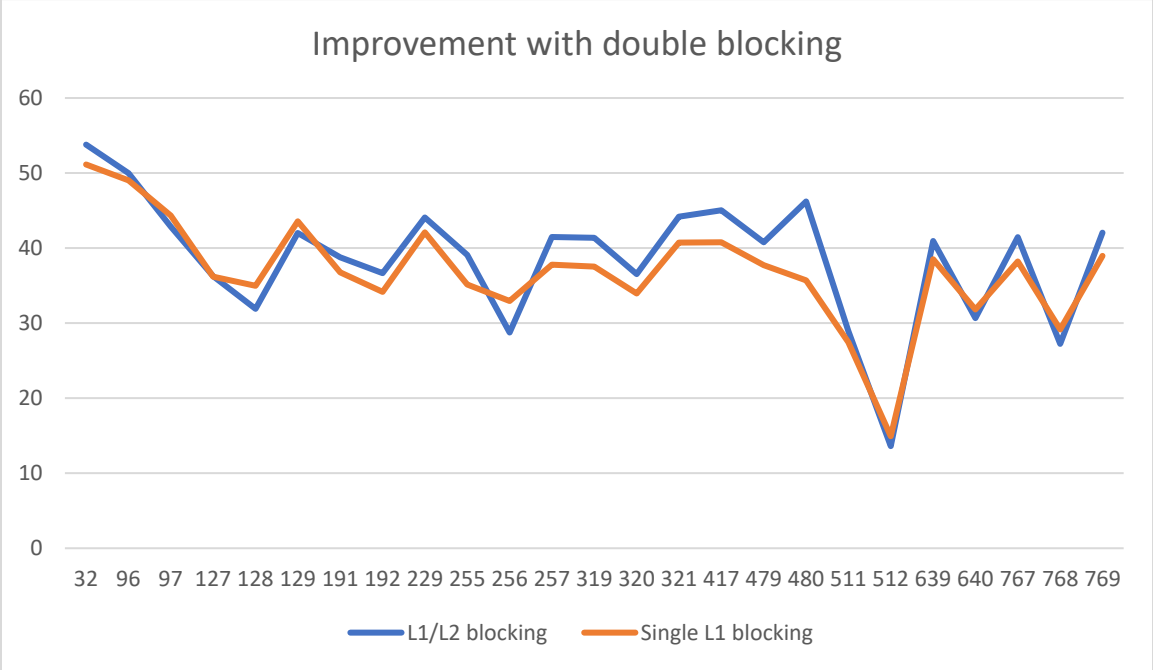
most 4 doubles. The block size that I chose at the start is a 4 by 4 block, i.e., computing 4 vectors using micro kernel each time. One noticeable thing in my implementation is that I made the vectors unaligned in order to avoid any segmentation fault issues. For entries in the original matrix that do not fit into a 4 by 4 block, I used the naïve approach to compute them directly. There should only be a linear number of those entries so it's fine not to optimize them. The improvement this time is significant compared to the initial implementation we had, achieving an average percentage of peak = 36.44. Please see the plot below for the performance with respect to different matrix sizes:




**3. Double Blocking:**

After implementing the micro kernel and getting a huge boost, I decided to follow the second idea in the handout and implement another level of blocking to improve the performance of the code. The code is exactly the same as the one for L1 blocking, with the size of block for L1 cache changing to the size of block for L2 cache. Ideally, since the

processor can now process blocks not only in L1 cache but also in L2 cache, there should be some performance improvement. The result of the optimization is summarized into the following chart:




There is some improvement when the matrix size is large, which is consistent with the effect of blocking: if matrix is small then there's nothing to block. The average percentage of Peak is 38.01, which is only of a little improvement.

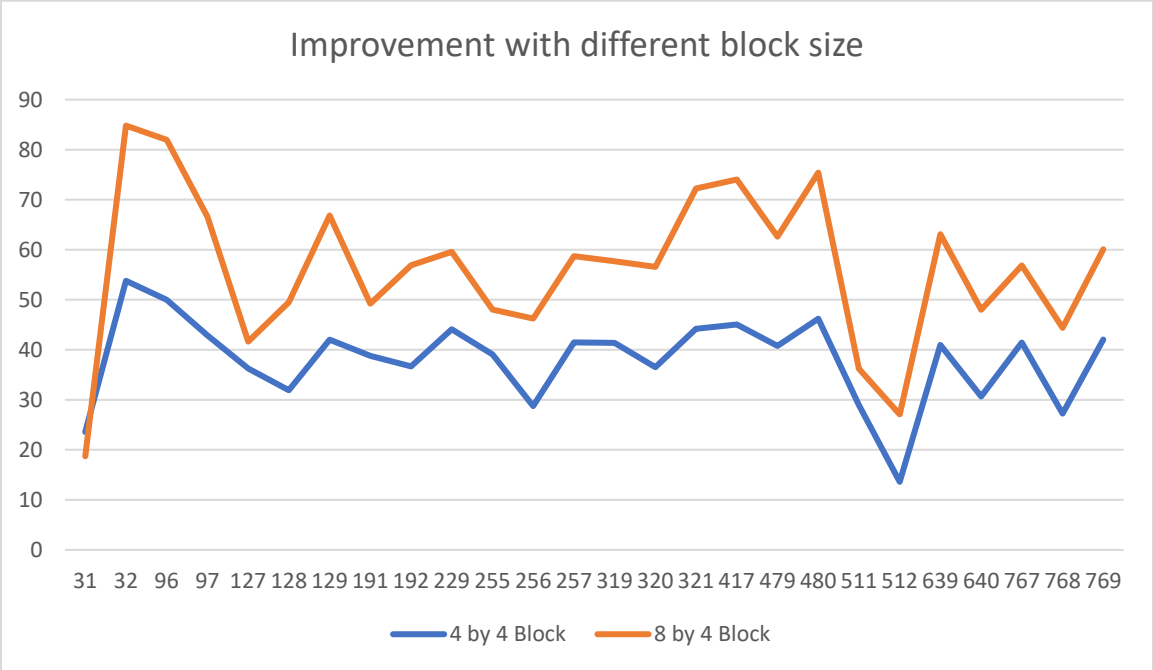
**4. Matrix repacking:**

I then noticed that in my micro kernel, the fetching of A is quick since it's reading continuous address, but the fetching of B might be slow, as each of the loading address differs by a column. As a result, I tried transposing matrix B to speed up the loading process. However, no matter transposing the whole matrix B at the start of the whole program, or transposing the block of B at the start of micro kernel, the average performance does not increase. In some scenarios, the performance is even lower with matrix repacking than

simply using micro kernel. The reason for the strategy to fail might be that when transposing the matrix, it is too expensive to load and copy a matrix (or a block).

**5. Micro Kernel, with different size:**

I saw on Ed discussion that we have access to 16 SIMD registers. This suggests that implementing the micro kernel with a 4 by 4 block might not be optimal, since it only uses 9 SIMD registers in my implementation. I switched the block size to 8 by 4, which now uses 14 SIMD registers and should be optimal. The performance is summarized into the following plot:



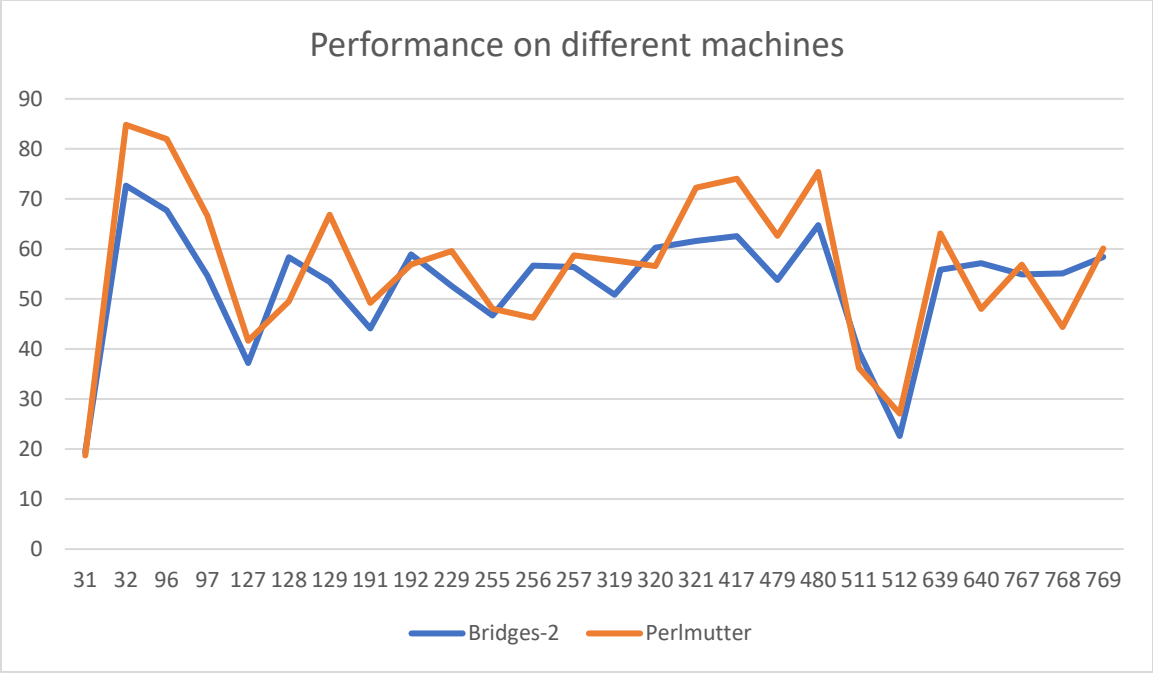

I had an average percentage of peak = 56.27! Additionally, we see that the performance of 8 by 4 blocks almost outperforms the one of 4 by 4 blocks at any matrix size. This is an amazing improvement, and it finishes the main part of my implementation.

**6. Block Sizes, revisited:**

At the last part of my implementation, I'm trying on different block sizes to see if there is any possible improvement on it. With a block size of 280 and 70 (which are theoretically closer to the upper bound), I got an average performance of peak = 30.58, which was a lot worse. This might imply that we should at least leave some room for the cache when considering block size. With a block size of 128 and 32 (or 64 and 16), the average performance of peak was roughly the same as the one I initially had. It was only when the blocks were too small (64 and 8) that I got an obvious decrease on average performance (47.52).

**7. Performance on different machine:**

For comparison, I ran my code on bridges-2 and got the following result:




The average performance of peak noticeably dropped to 52.92, which means that the performance was worse on brideges-2 than on Perlmutter.