CS 5220 Assignment 1 - Phase 2 Report

Group 24: Xiang Long (XL483), Yu Su (YS576), Joshua Cohen (JBC264)

1 Byte alignments and intrinsics

As per the group 22 peer review, we were suggested to try the __mm__malloc intrinsics for allocating buffers for copy optimization. The is possible for different byte alignments and so we experimented with different values:

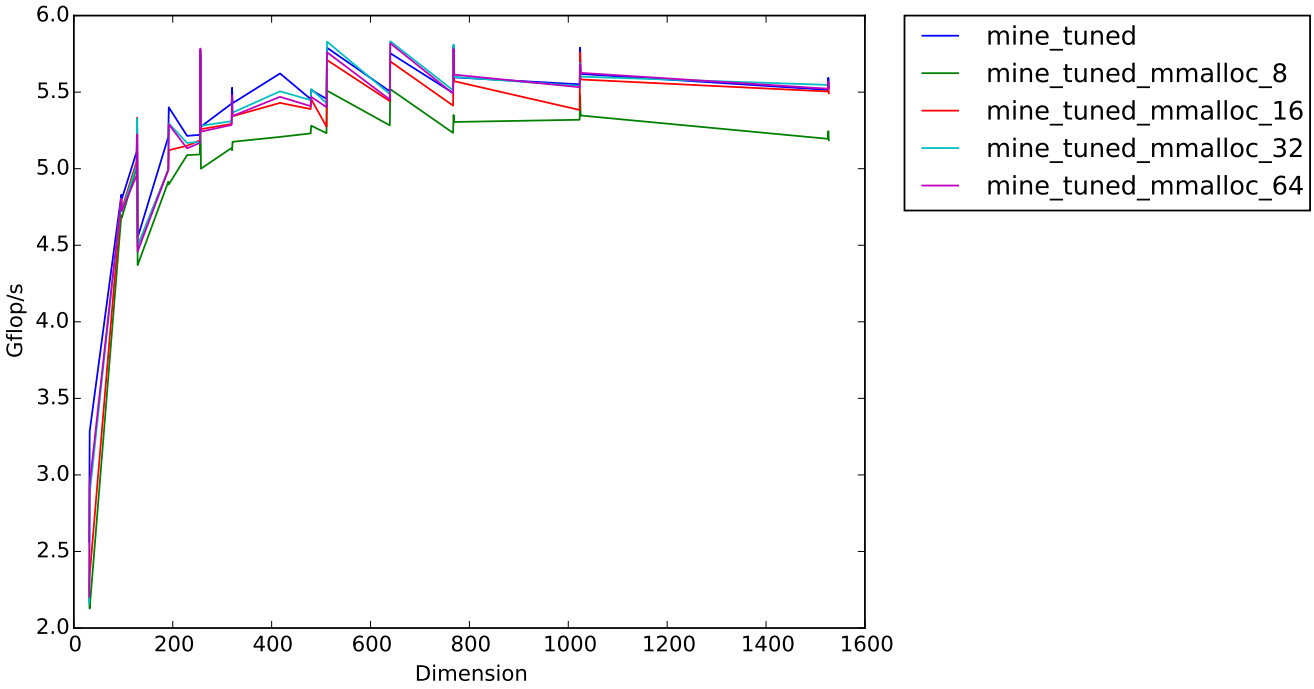

Figure 1: Comparison of different byte alignments against original version that used simple malloc

There was not much performance difference but arguably the byte alignment of 64 bits was the best out of the group, which makes sense as that is the word size of the processor.

2 Block size, revisited

We were also suggested to try to use a block size that takes account of only two matrices in the L2 cache, rather than our current three, the reason being the result C matrix does not need to be cached. We should also try to use a block size that is not a power of 2. As per this suggestion, the block size we should try is:

$$\text{BLOCK\_SIZE} = \sqrt{\frac{2^{18}\text{B in L2 cache}}{8 \times 2}} \approx 181$$

In our phase 1 report, we also calculated that if we did not choose powers of 2 as our block size, the appropriate block size for three matrices in the L2 cache should be 104 bytes. We compare this against the block size of 181 bytes and also our original 128 bytes:

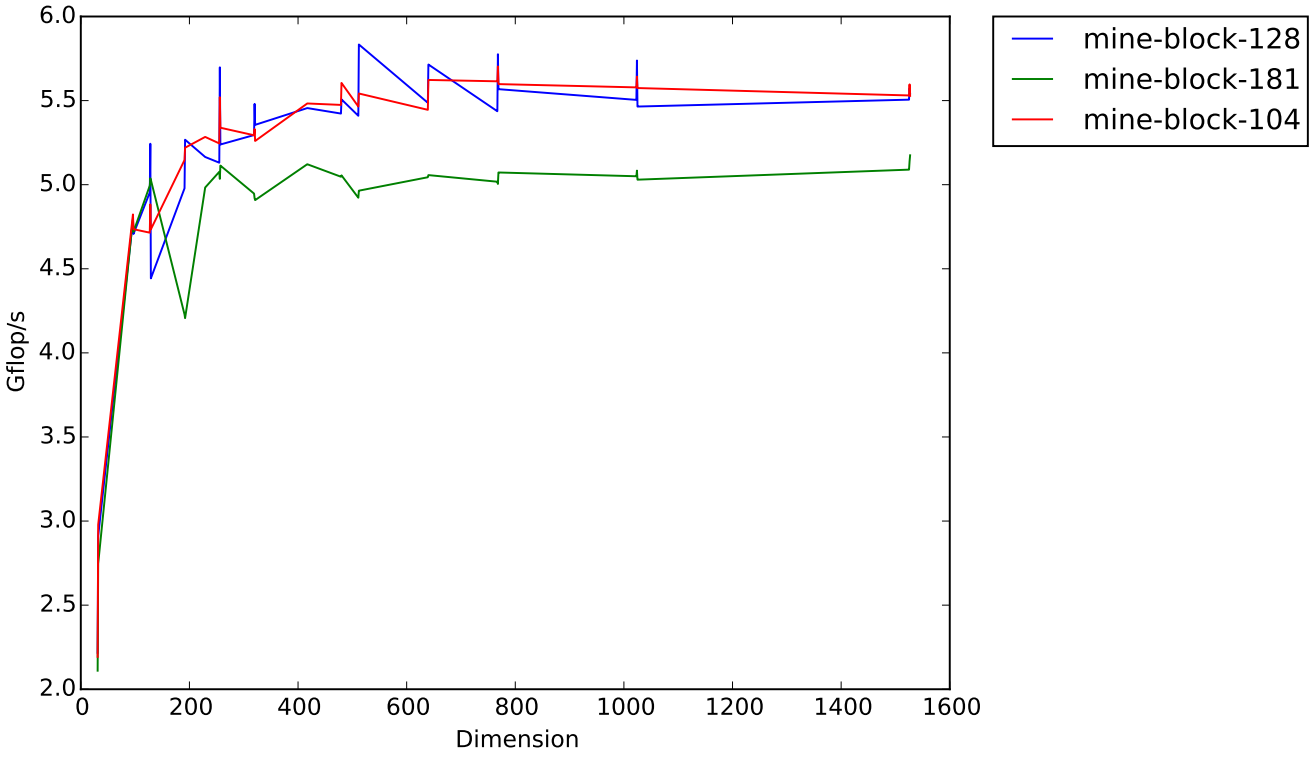

Figure 2: Comparison of different block sizes

It turns out that a size of 181 bytes clearly performs poorer than our original choice of 128 bytes, and the latter only has small differences with the size of 104 bytes. Since a block size of 104 bytes saves on cache space, we will change to this instead, but we shall keep trying to cache all three matrices.

3 Multilevel blocking

Since the Xeon Phi processors are also equipped with an L1 cache of 32KB, we could attempt to introduce a further level of blocking in view of this size. The block size would be:

$$\mathsf{BLOCK\_SIZE} = \sqrt{\frac{2^{15}\text{B in L1 cache}}{8 \times 3}} \approx 36.95$$

The multilevel blocking is implemented by first padding the matrix to a multiple of the second level (L2) block size, divide into blocks then recursively perform matrix multiplication on each of the blocks that further divides into the first leve (L1) block size. The performance of holding the second level block size steady at 104 and varying the first level block size is plotted below:

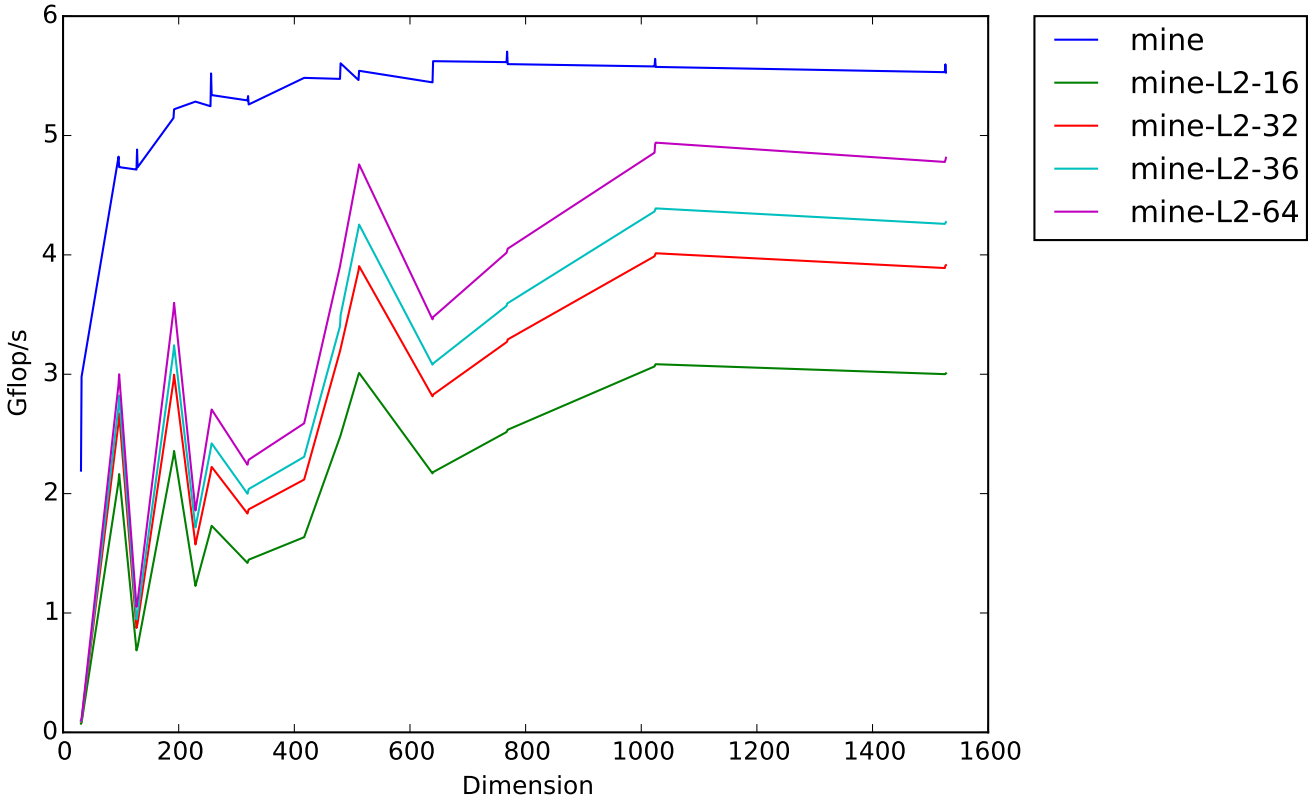

Figure 3: Comparison of different first level block sizes, second level block size constant at 104

So we don't seem to be gaining any performance, perhaps due to the overhead of allocating and copying many of these tiny blocks. On the other hand, we could also try to optimize for the L3 cache instead. Each Xeon Phi processor has 15MB of L3 cache shared between all six cores. Assuming we can utilize all 15MB on a single core, the block size should be:

$$\text{BLOCK\_SIZE} = \sqrt{\frac{15 \times 2^{20}\text{B in L3 cache}}{8 \times 3}} \approx 809.54$$

So this time we shall set the first level (L2) block size to 104 and vary the second level (L3) block size. The result of this is shown below:

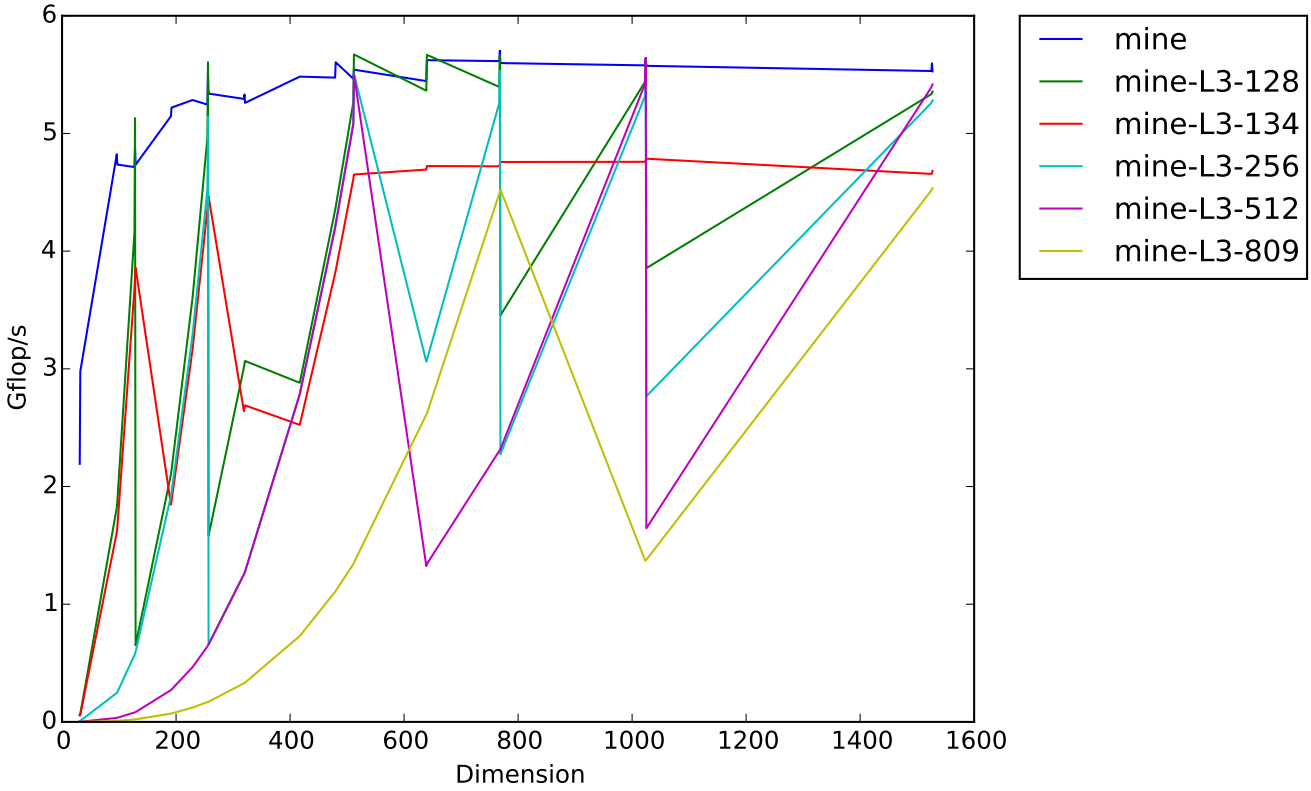

Figure 4: Comparison of different second level block sizes, first level block size constant at 104

We were still not able to obtain a performance gain.

4 Conclusion

Despite a series of further experiments, we were not able to obtain a significant performance improvement. However we made small modifications such as using the intrinsic __mm__malloc. The final performance of our solution is plotted below against all other implementations:

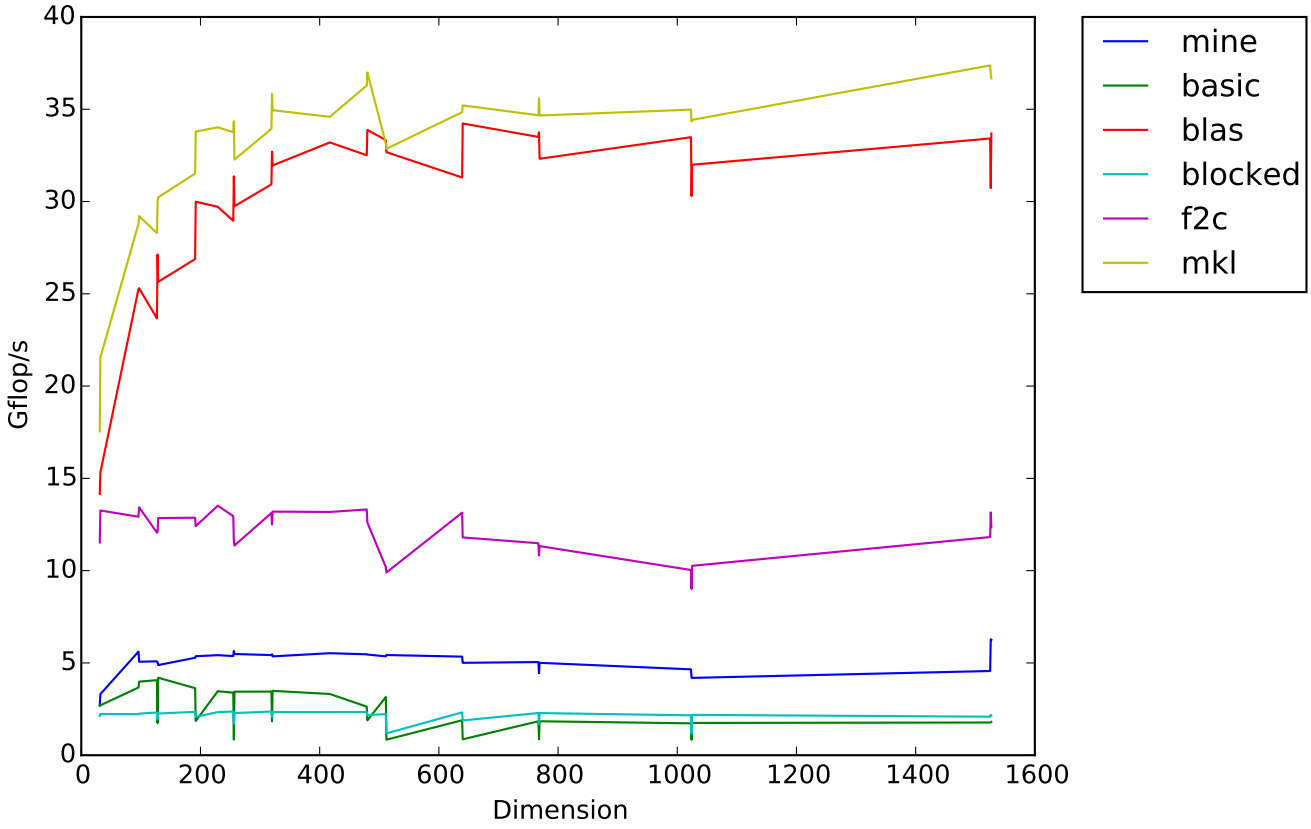

Figure 5: Comparison of our final implementation