CS5220: Matrix Multiply Project

Junyoung Lim, Max Ruth, Arjun Sharma

October 5, 2020

1 Introduction

We have tried a variety of ways to optimize matrix-matrix multiplication and are able to improve upon the basic algorithm approximately by a factor of three (64_block vs. basic in figure 1). The best optimized case we have obtained achieves only about 35-40% as efficient as BLAS algorithm (64_block vs. blas in figure 1).

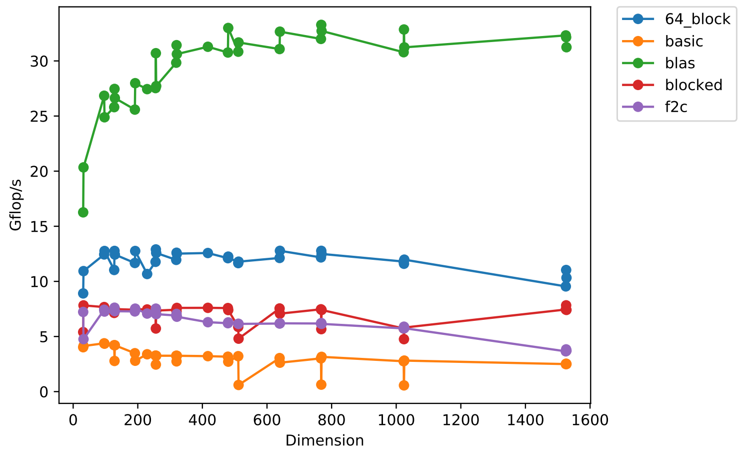

Figure 1: Flop rates for the given f2c, blocked, blas and basic algorithm compared with the our best algorithm, 64_block.

In the rest of the report, we describe the incremental steps that lead us to the `64_block` algorithm. In Sec. 2, we discuss the changes in compiling the code. Next, we describe the kernel that we developed in Sec. 3. The kernel code is extended to blocked code in Sec. 4 for large matrices. Finally, we conclude in Sec. 5.

2 Discussion: Compilation Details

The simplest design decisions that we made were on the compiler side. We used the GCC 8.3.0 compiler that was downloaded by default in the `build-essential` package. The three flags that we used for the compilation are the following:

- `-O3` — Enable more aggressive optimization than the flag `-O2` used by default.
- `-ffast-math` — Allows the compiler to reorder operations that may not actually be associative. We did not see a dramatic increase in the error of the result using this flag, although there would be a chance for that when floating point operations are seen as associative. Perhaps for more pathological matrix examples, we might have to worry about this flag.

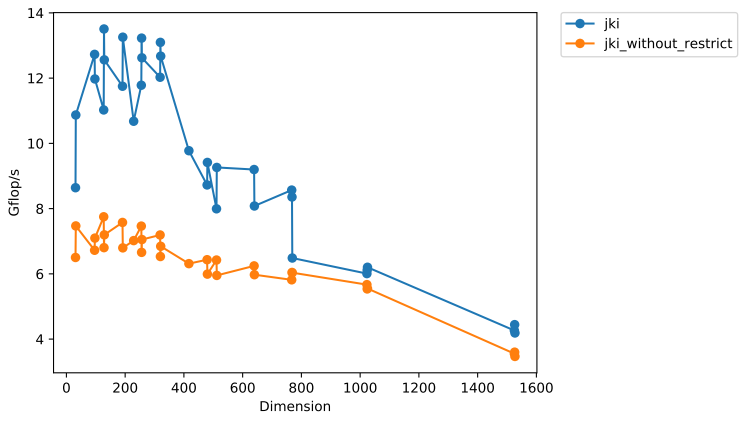

Figure 2: Flop rates for the inner `jki` kernel with and without the `restrict` keyword.

- `-march=native` — Optimizes the code for the specific machine architecture. When running on a desktop with the flag `-fopt-info-vec`, adding `-march=native` changed the vectorization from 16 bytes to 32 bytes wherever possible - allowing a potential doubling of the computation speed.

3 Discussion: the Inner Kernel

The full code for our best inner kernel (found in `dgemm_jki.c`) is given by:

```
void square_dgemm(const int M,
                  const double* restrict A, const double* restrict B, double* restrict C)
{
    int i, j, k;

    for (j = 0; j < M; ++j) {
        for (k = 0; k < M; ++k){
            for (i = 0; i < M; ++i) {
                C[j*M+i] += A[k*M+i] * B[j*M+k];
            }
        }
    }
}
```

There are two major differences between this code and that of `dgemm_basic.c`. First, this code has takes advantage of the `restrict` keyword when passing `A`, `B`, and `C`. This notifies the code that these matrices do not share pointers, and thus the output of `A[k*M+i] * B[j*M+k]` does not depend on the previous updates to the value of `C[j*M+i]`, allowing for GCC to vectorize the code appropriately.

The difference between the loop with `restrict` and without is show in Fig. 2. We see that the Gflop rate of the `restrict` code is approximately twice that of the code for smaller matrices. This is because the computation in these situations is limited primarily by the processor and not by memory. For larger matrices, the primary limitation on computation speed moves to be memory, and the relative performances narrow, as neither is better than the other for memory hit rates.

The second difference is the loop order. The `dgemm_basic.c` code loops through the matrix multiplication in the `ijk` order, meaning the outside loop is in `i`, the middle is in `j`, and the inner is in `k`. However, this loop order is not particularly good for memory locality, as each iteration of the inner loop requires drawing upon an element of `A` that is `8M` bytes away. Note this is particularly important for the inner loop, as it is incremented a factor of `M` times more iterations than the middle loop and `M^2` times more than the outer loop. Loading these elements quickly in succession is more difficult for this reason.

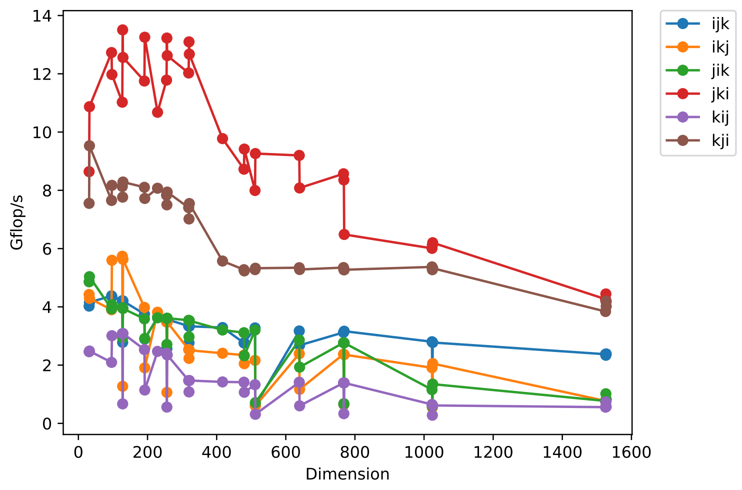

Figure 3: Flop rates for each of the six possible loop orders for the inner kernel.

To address the locality issue, we choose the inner loop to be over `i`. This is because only the `i` iteration is local in all three matrices `A`, `B`, and `C`. The middle loop would then be over `k`, which has good locality in `B`, but not in `C`. Finally, the outer loop would be over `j`, which has the worst memory locality.

We find that the numerical experiments shown in Fig. 3 confirm this intuition. The two loop orders where `i` is the inner variable are faster than the rest, with the `jki` being the fastest of them all.

We can learn a few other things from Fig. 3. For instance, it appears that when comparing between dimensions that are powers of two and their neighbors (such as the run 255, 256, and 257), the speed of the code can be significantly different. We believe this is likely related to associative caching. However, for small matrices that fit entirely into cache, it is unclear exactly why this would be an issue. We also see that these basic loop routines are strongest for small matrices, and tend to become significantly worse for large dimension. This could be interpreted via the roofline model, where smaller dimensions are bounded by flop rate (on the right side of the roofline) and the larger dimensions are bounded by memory efficiency (the left side of the roofline).

3.1 Register Allocate

Considering the jki loop order, the innermost loop reuses B[k,j]. Thus it can be register allocated such that it is fetched from the memory only once. [1] Practically this implies changing the algorithm to:
do k=1,M; do k=1,M;
r=B[k,j]
do i=1,M
C[i,j]+=A[i,k]*r
The benefit of register allocating is observed for the largest matrix size only in figure 4. For smaller matrices it is sometimes slightly detrimental. This is perhaps because for smaller matrices cache is big enough to hold B[k,j] in the memory by the time it has to be reused and writing it to another variable r reduces performance.[1]

4 Discussion: Blocking

We have seen that once the size of the matrices exceeds the size of the cache, there is a drop in performance of the inner kernel (e.g., Fig. 3). This drop can be attributed to a bad memory access pattern - i.e. we don't reuse values fast enough, and then we suffer capacity misses. To alleviate this, we work with blocks of the

[1]Lam MD, Rothberg EE, Wolf ME. The cache performance and optimizations of blocked algorithms. ACM SIGOPS Operating Systems Review. 1991 Apr 1;25(Special Issue):63-74.

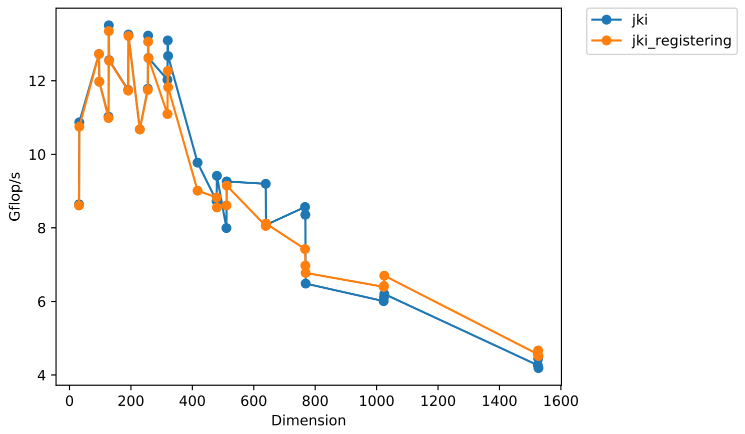

Figure 4: Flop rates with and without register allocating.

matrices which fit entirely in the cache. Performing a multiply on the blocks happens quickly, and we reduce the amount of memory traveling between the cache and RAM.

For our multiply, we found from some sources[1] that a good idea is to use "strips" of the matrix. We found that looping through vertical strips of `A` and `C` is ideal, which is likely due to the fact that the inner loop over `i` can extend over the whole of the matrix (and hence, we have memory locality).

The width of the columns, called `BLOCK_SIZE` in the code, must be tuned for the size of the cache and matrix. This is because we would like to use large amounts of our cache for prolonged fast computation time, but we cannot use more than the cache without having capacity misses.

To choose the optimal block size, we ran a variety of tests (see Fig. 5). We see that the optimal block size appears to be about 64 for the dimensions given. For block sizes lower than that (such as 8), we find that the performance never quite matches that of the larger block sizes. For too large block sizes (such as 128), we see that the performance is very strong at values in the middle dimensions, but drops strongly for large matrices.

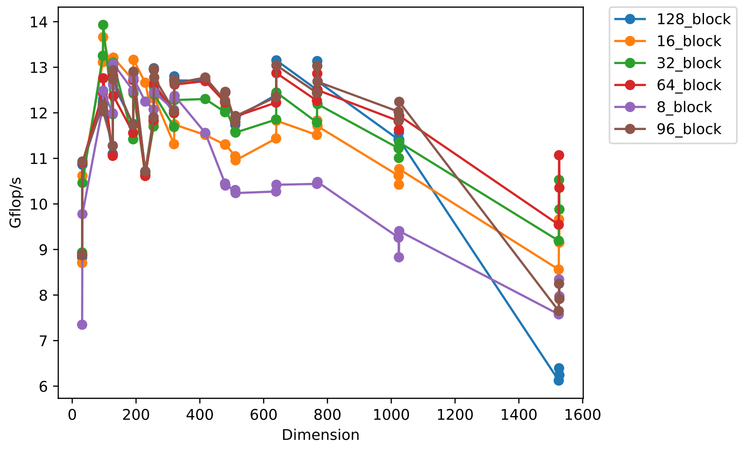

Figure 5: Flop rates with different block sizes, e.g. `64_block` indicates blocking with a size of 64.

4.1 Effect of register allocating with blocking

Figure 6 shows the effect of register allocating (section 3.1) when combined with blocking with 64 as the block size. The improvement due to register allocating is negligible. Neverthless, we keep register allocation in the final algorithm whose results are reported in figure 1.

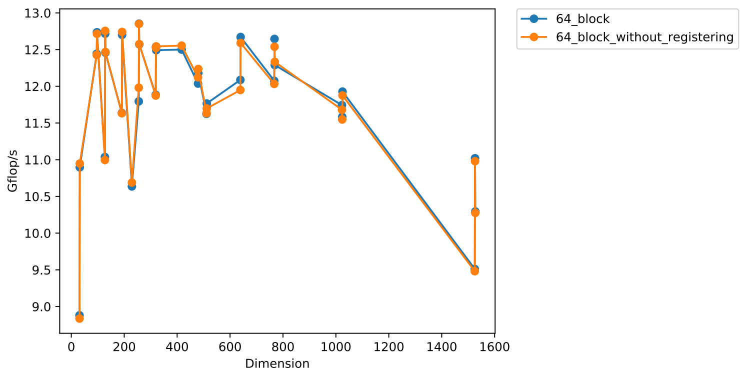

Figure 6: Flop rates with and without register allocation when combined with blocking with 64 block size.

5 Conclusion

As mentioned in the Introduction we are able to improve upon the basic algorithm by about 3 times. However, we are below BLAS by almost 2.5 times. Three changes provide the most benefit: using the `restrict` keyword, optimizing the loop order to exploit the memory structure, and blocking by performing calculations on sub-matrices that can fit in the cache. Implementation of register allocation has mixed results especially when combined with blocking.

We tried certain other things that have negligible or negative consequence on the performance. For instance, when we attempted to perform copy optimization, where we copied the whole of `A`, `B`, and `C` into structured aligned arrays. However, any potential improvement due to alignment was negated by the large amount of time moving memory. With more careful timing and storage, it is likely that alignment could be used to effect, but it would take much more attention to detail than simple copying. Another option would be to use the specific commands allowed by the Intel compiler, shown in the `kdgemm.c` example.

To improve the code further, we would likely start with a more careful profiling of the code using, e.g., `google-perftools`. This would allow us to more accurately assess how the code is performing. Further, we would update the code to use compiler intrinsics, which would allow us to take advantage of much faster code. Finally, it is possible that some sort of alignment of memory might help the code run faster, despite previous negative trials.