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 **??**). The best optimized case we have obtained achieves only about 35-40% as efficient as BLAS algorithm (64_block vs. blas in figure **??**).
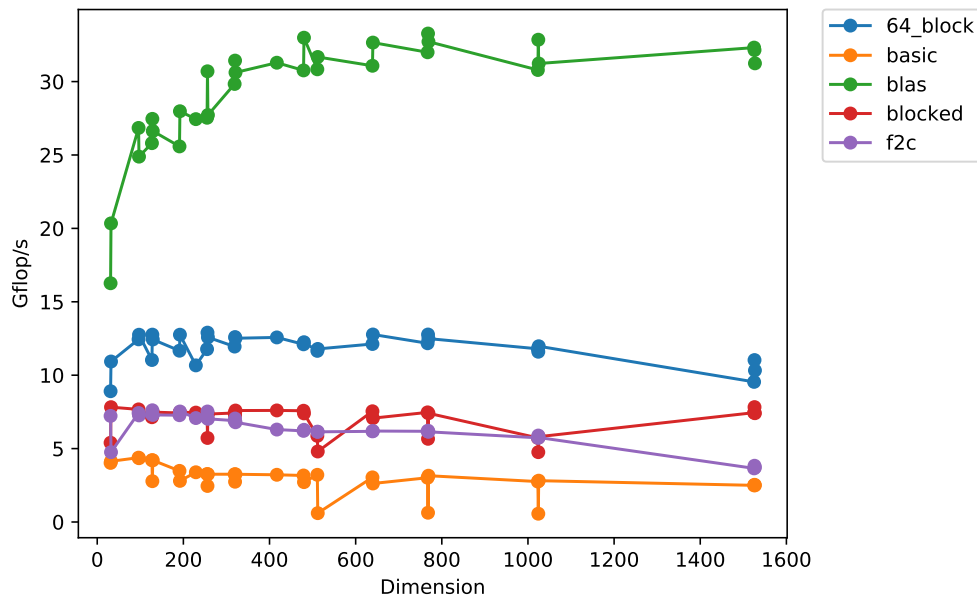

Figure 1: Flop rates for the given f2c, blocked, blas and basic algorithm compared with the our best algorithm, 64_block.

In the rest of the report describe the incremental steps that lead us to the 64_block algorithm.
References:
https://stackoverflow.com/questions/38190006/matrix-multiplication-why-non-blocked-outperforms-blocked

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.

- `-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.

Of course, there are likely many more compiler options that could have been considered. We did not experiment with the CLang, gfortran, or Intel compilers. From the

3   Discussion: the Inner Kernel

Plot: restrict vs. not restrict
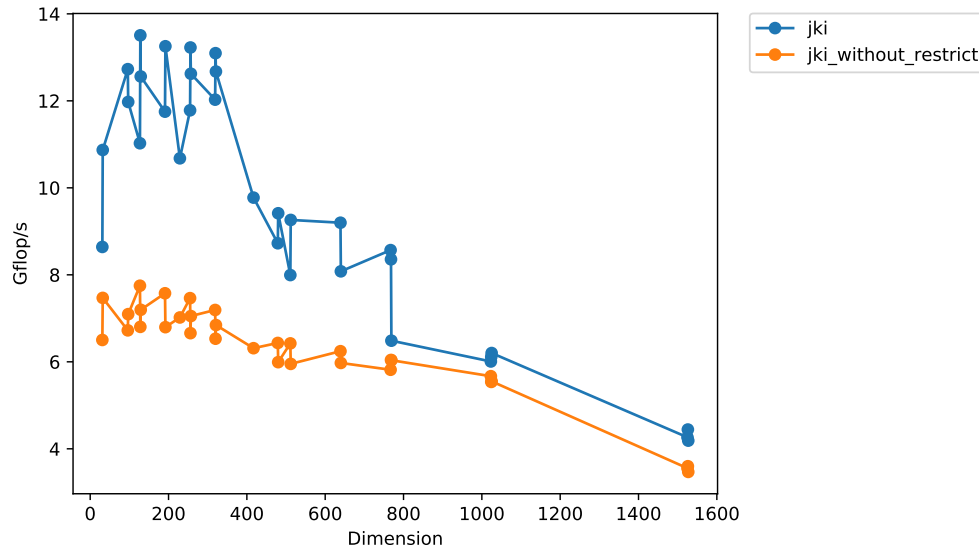
3.1   The `restrict` Keyword

Figure 2: Flop rates with and without the `restrict` keyword.

3.2   Loop Order

In the basic algorithm, matrix multiplication of two $M \times M$ matrices is performed in a sequence of three nested loops:
do i=1,M; do j=1,M; do k=1,M
C[i,j]+=A[i,k]*B[k,j]
The loop order written as above is ijk. The sequence of the i, j and k loops does not influence the accuracy of the calculation or the number of individual calculations. However, it has a profound effect on the time required due to the column major storage of arrays. This can be quantitatively observed in figure **??**. As the data is stored in the column major format, accessing C is faster if we first loop through the rows of a given column first. Hence, having i loop as the inner most loop is fastest i.e. the plots labelled as jki and kji in figure **??**. Same logic applies while accessing the elements of the array B and looping through the rows of B i.e. the k loop before the columns i.e. the j loop is more efficient than vice versa. Hence, jki is faster than kji in figure **??**. In the rest of the report jki loop order is assumed.
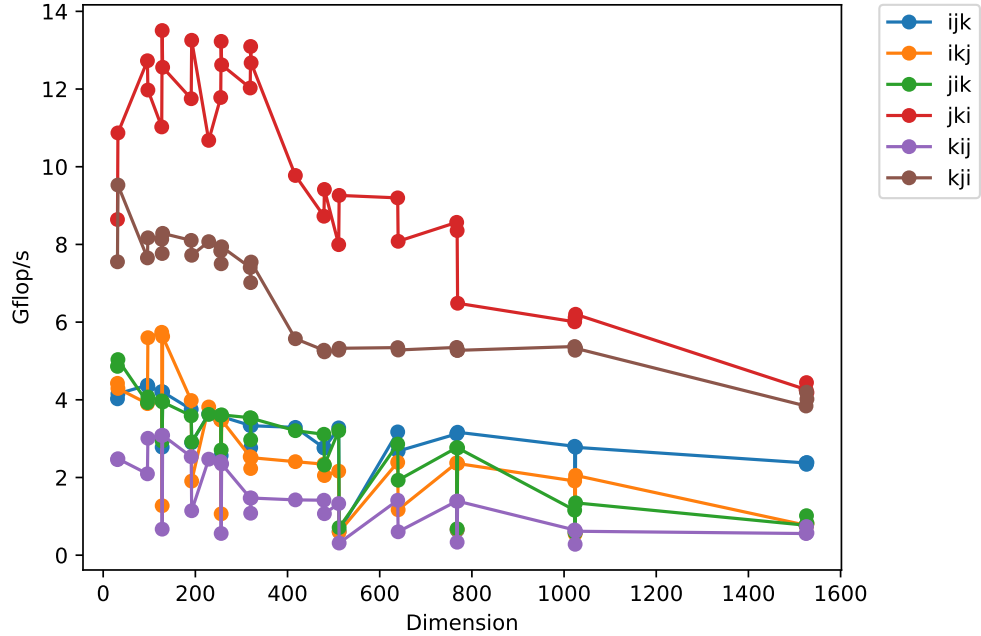
Figure 3: Flop rates for each of the six possible loop orders for the kernel.

3.3   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 **??**. 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.**??**[0]

4   Discussion: Blocking

If the calculations are performed on submatrices fetched from the original matrix that can completely fit into a faster level of memory hierachy such as cache, average access latency is reduced and the number of flops in a given time are increased.**??**[0] This is the idea behind blocking that allows an improved performance.

However, block size is a parameter here and can largely affect the performance as observed from figure **??**. Comparing figure **??** and **??** we observe that almost all the block sizes tested lead to a better performance. However, the most efficient block size depends upon the size of the matrix. As mentioned by Lam et al. **??**[0], the matrix size and cache parameters must be taken into account to obtain a most efficient block size (choosing a block size to fill whole cache or a fixed portion of cache is not the best strategy). While we have not tailored the block size for each dimension, from figure **??** we can see that blocking with 64 as the block size leads to near optimal performance for all the dimensions considered.

---

[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.
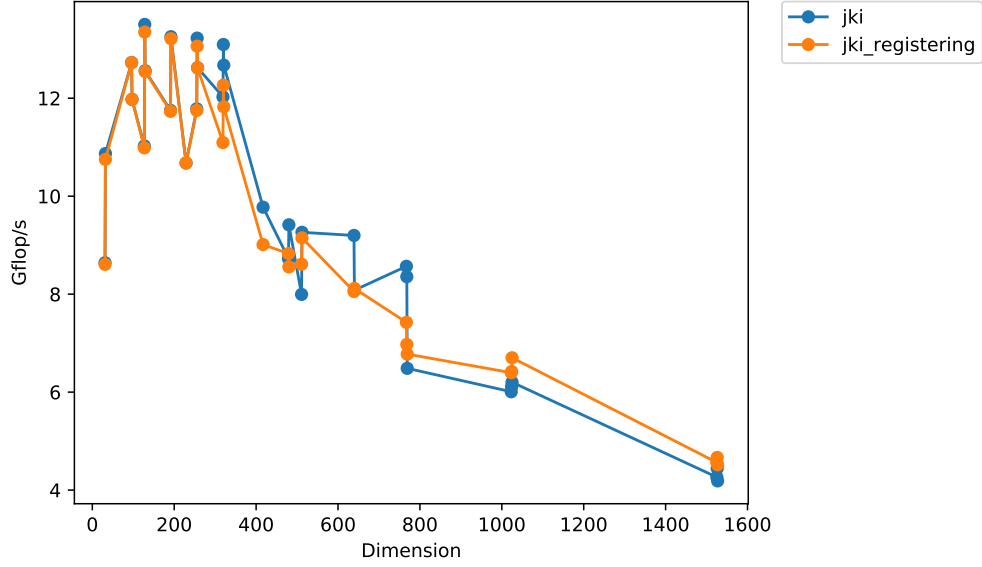
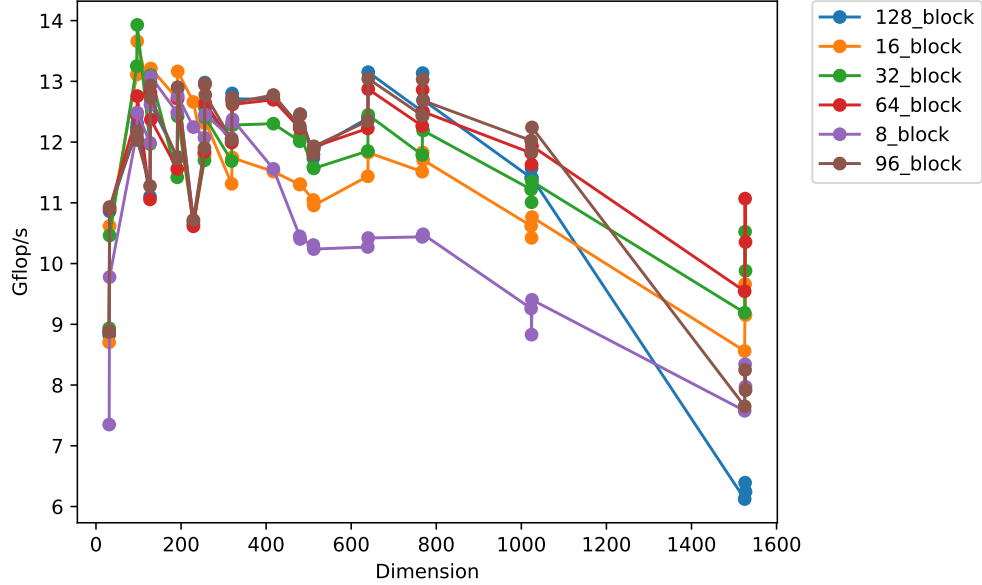Figure 4: Flop rates with and without register allocating.

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

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: (a) restricting the ....
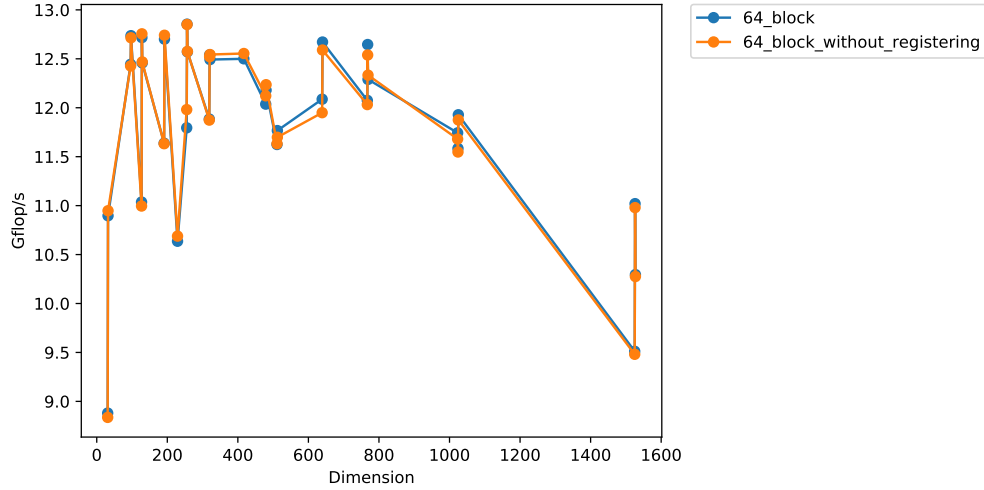
Figure 6: Flop rates with and without register allocation when combined with blocking with 64 block size.

(b) optimizing the loop order to exploit the memory structure and (c) 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. These include coping ... First, copying large amounts is bad. For instance, as a form of copy optimization, we attempted to copy 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.

Second, large improvements in the flop rate could be made with simple changes to the code and compiler. For instance, after the addition of three compiler flags, changing the loop order of the basic code, and adding the `restrict` keyword, the speed of the code improved by approximately an order of magnitude, especially for small matrices. However, when we tried writing large amounts of code (e.g., when trying the copy optimization previously described), the performance tended to go down.

- We should have more carefully profiled our code.

- Specific details about what is happening on a computer are important.

- BLAS is impressive, but weirdly slow for small matrices.

- Alignment is confusing