CSE 260 Assignment Report - Blocked Matrix Mutiplication

Xinyu Xie, A53285829; Yiming Liao, A53274851

Section 1. Results

In assignment 1, we use our knowledge of the memory hierarchy and vectorization to optimize a matrix multiply routine. With the provided simple starter code, we try to optimize that starter code to achieve higher performance. We will show the details of our optimization in the section 2. In this section, we will show the performance of our implementation and explore some interesting points of our results.

1.1 Performance Study

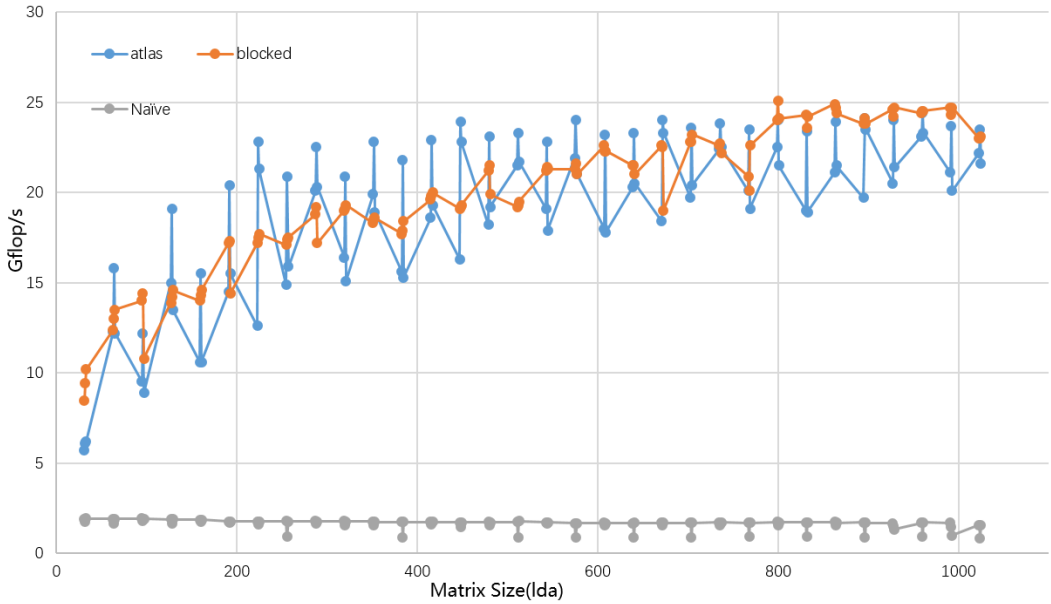

Figure 1. Performance vs Matrix size from 32 to 1024 of atlas, block and naive respectively

We can see from the picture above, our implementation have a huge improvement compared with the naive algorithm and also achieve the performance which is close to the performance of Atlas. And we find that when the matrix size is relatively small or large, our implementation can even have better performance than Atlas. Generally, we find that the performance of our implementation is increasing when the matrix size get larger.

1.2 Comparison with Naive Code on 6 Interesting points

In order to explore more features of our implementation, we select six interesting points to compare the performance between the Atlas and our implementation. The points we selected is N=32,64,128,256,512,1024,and N is represent the input matrix size.The figure shows as following:

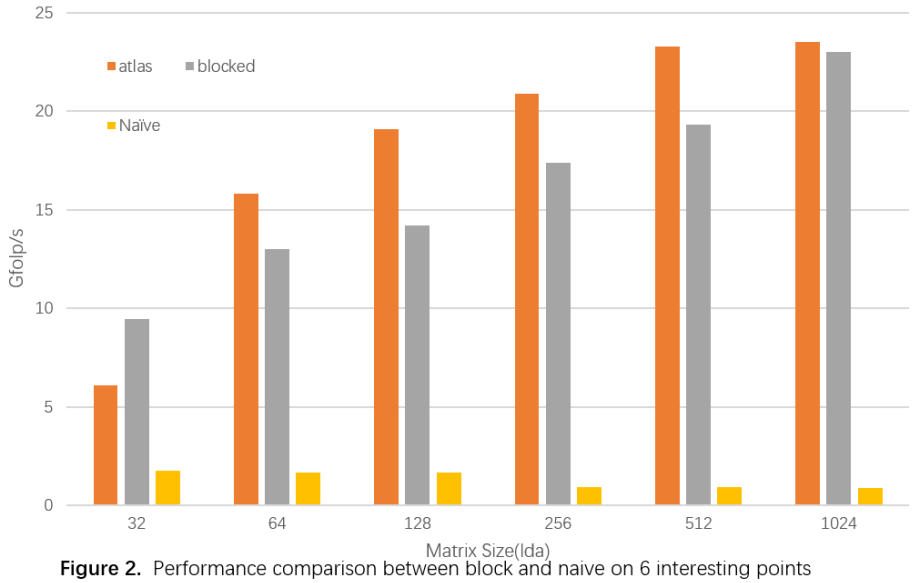

Figure 2. Performance comparison between block and naive on 6 interesting points

We can see from the picture that when the matrix size is 32, our implementation have better performance than atlas! It's really cool and we thought that it is because atlas is a widely used library so it may focus on improve the performance in genreal case, and our implementation tune the different block size to fit the cache size of our machine so we can get good performance. And the 3rd-level block of our implementation is 48, which is close to 32, we think this may be a point help us achieve high performance when the matrix size is small.

As the matrix size gets larger, the performance of our implementation and the Atlas both become better, while the performance of the naive algorithm become worse. We think it's become when matrix gets large, Atlas and our implementation can make better use of the 3-level cache with register tiling and waste less memory during matrix calculation, however, the naive algorithm just calculate more times when the matrix size become larger, so it becomes slower.

1.3 Irregularity of Performance

An interesting thing we find from our result is that, for Atlas algorithms, the performance is higher when the matrix size is a multiple of 32, compared with the performance on the matrix size plus one or minus one. For example, the performance of Atlas when matrix size is 256 is better than the performance when the matrix size is 255 and 257. However, for the naive algorithm, the situation is exactly the opposite. That is, the performance is lower when the matrix size is a multiple of 32. The performance of the naive algorithm when matrix size is 256 is worse than the performance when the matrix size is 255 and 257. As for our algorithm, there is no significant difference between the three close matrix size. It's really interesting. we think it is because the implementation of atlas algorithm may focus on the size for the multiple of 32 in order to get higher performance in general case. As for the naive algorithm,there is a significantly higher dTLB miss rate when the size is a multiple of 32, which is caused by the super alignment between page access. As a tile

Section 2. Optimization and Analysis

Overview

In our implementation, we try to optimiza the matrix multiplication program with 3-level cache blocking, vectorization, register tiling, loop unrolling, the 'strict' keyword and compile optimization. In this section, we are going to analyza how these methods work on this problem in detail, accompanied with experimental results. With these optimizations, the final performance of our program achieves a high score very close to Atlas results. At the beginning, as an overview, we list the parameter setting in our implementation.

- 3-level block size: 1152, 384, 48
- Register tiling: 4×12
- Register utilization: 12 registers for C, 3 registers for B and 1 register for A
- Compile options: -O4 -DMY_NEW_CODE -avx2

2.1. Cache Blocking

Cache Blocking is a technique to rearrange data access to pull subsets (blocks) of data into cache and to operate on this block to avoid having to repeatedly fetch data from main memory.

Cache Blocking is an important class of algorithmic changes involves blocking data structures to fit in cache. By organizing data memory accesses, one can load the cache with a small subset of a much larger data set. The idea is then to work on this block of data in cache. By using/reusing this data in cache we reduce the need to go to memory (reduce memory bandwidth pressure).

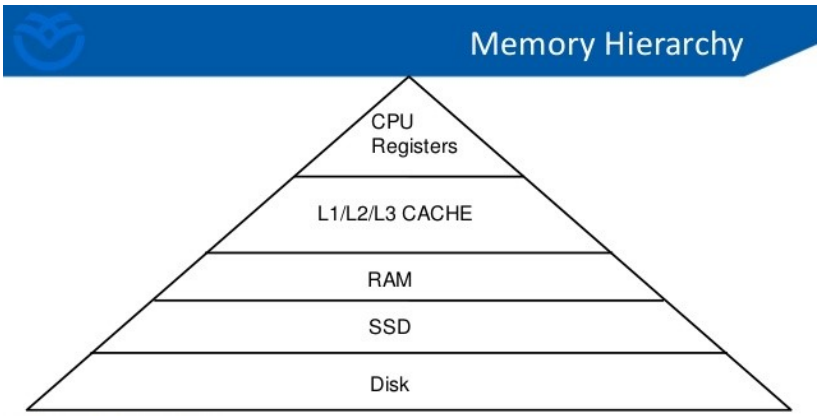

Figure3 Menory Hierarchy

We know that the cache has 3 levels so we do 3 level blocking and try to fit each cache level with different block size. If each level block size could exactly fit in each level cache, then we could achieve good performance. Thus, we do a few experiments on the different level of block to find the better block size for each level.

2.1.1. Case Study

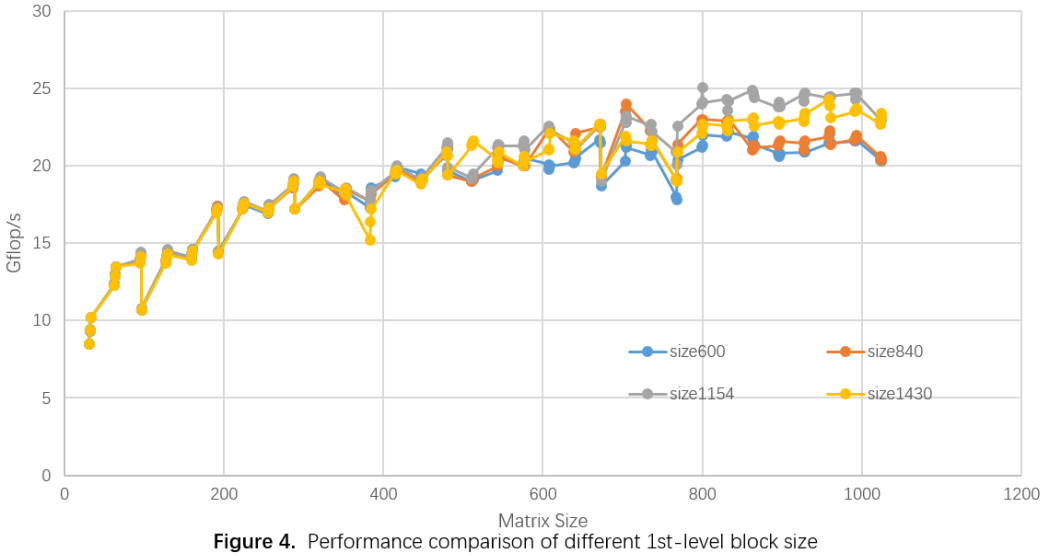

Figure 4. Performance comparison of different 1st-level block size

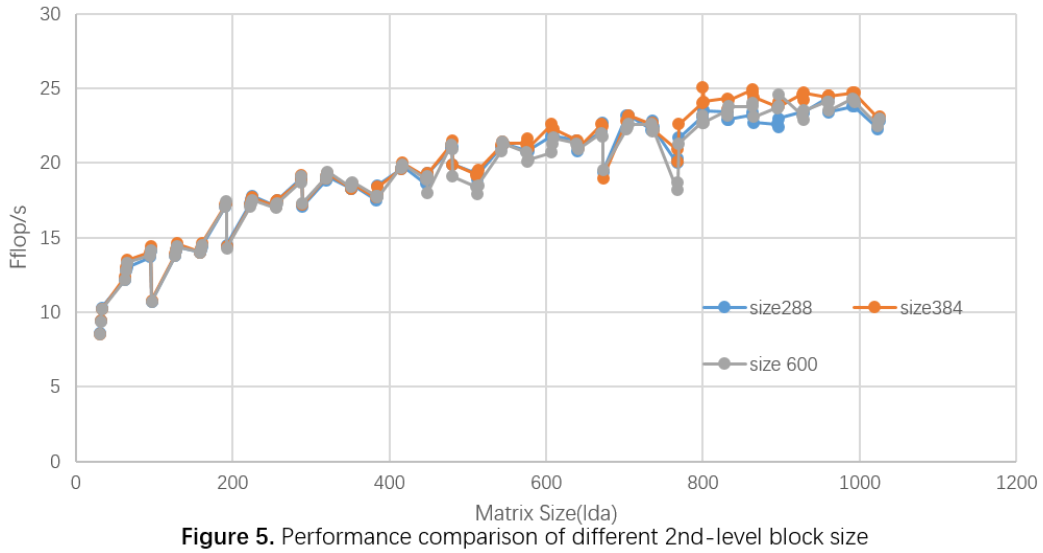

Figure 5. Performance comparison of different 2nd-level block size

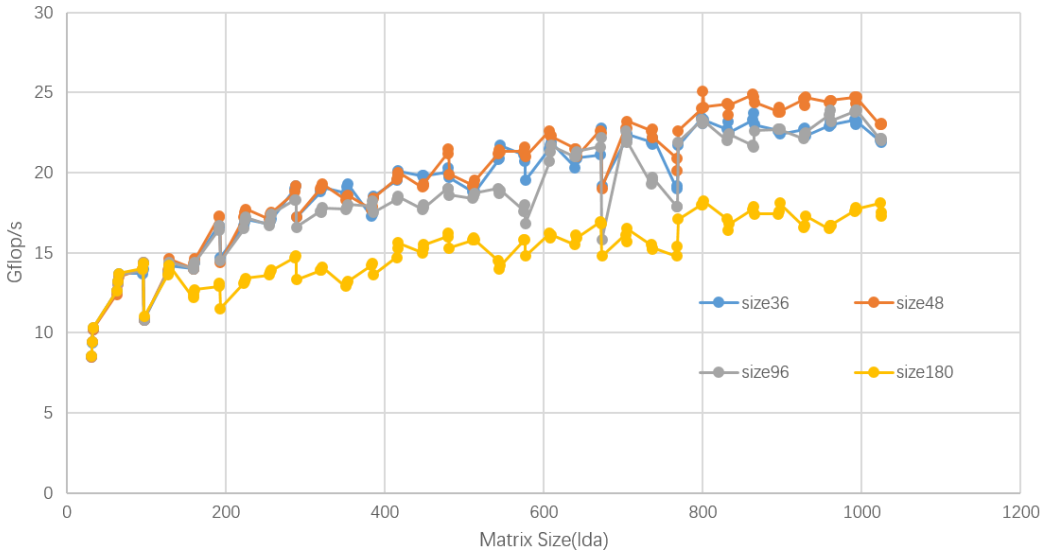

Figure 6. Performance comparison of different 3rd-level block size

The above three pictures show our experiment results on different level of block with different block size. We choose the optimal block size for each level block based on the experiment.In fact, we find that the block size for the 1-st level and 2-nd level have no particularly large impact on the final performance, you can see it from the first two pictures. While the block size for the 3-rd level block should be chosen carefully.

2.1.2 Insight of block size tuning

The assumption of our tuning technique is that if the closer block size to cache size, the better the performance we can achieve. So the cache size is the optimal value for our block size. Then we try to use binary search to find the exact value fo the cache size. At the same time, because the size of sub-sub-matrix we used in Register Tiling is 4×12, the block size we chose must be a multiple of 12.

2.2 Vectorization and AVX2 Intrinsics

Advanced Vector Extensions (AVX, also known as Sandy Bridge New Extensions) are extensions to the x86 instruction set architecture for microprocessors. It extends the capability of Streaming SIMD Extensions, especially for floating point data and operations. AVX2 is an extention of AVX which introduces fused multuply-accumulate(FMA) operations. In our implementation, we use AVX2 operations and enable it during compiling. The improvement of AVX2 instructions, or saying SIMD, is based on the idea of instruction level parallelism. For programs requiring to perform the same operation on multiple data points, SIMD enables machines to do same operations on multiple processors simultaneously, which increases utilization of processors computation resource. Because in the matrix multiplication problem, instructions are quite simple and highly repetitive, it's reasonable to adopt SIMD. In this work, SIMD is closely related to vectorization that divides the whole task to multiple vector multiplications which can be solved parallelly. We use AVX/AVX2 instructions during register tiling procedure which does vector mutiplications.

2.3 Register Tiling and Loop unrolling

Register tiling

Register tiling is an optimization idea that trys to make maximum use of registers, which are the fastest memory in our hirearchy. Also like cache blocking, when computing sub-matrix multiplication results, it's possible to break it down into smaller 'blocks' which can be handled with registers. Combined with vectorization, we manually assign vectors to different registers so as to increase parallelism in computation. In AVX2, there're totally 16 256-bit registers. As shown in Figure 7, suppose we are updating values of a sub-sub-matrix of size \\(4 \times 12\\) which is a part of the third-level matrix block, we can load this target sub-sub-matrix into 12 registers and the 2 mutiplier matrix into the rest registers vector by vector. Then during computation, we can reuse the 12 registers for target sub-sub-matrix without frequently loading and storing new values which might be done if we don't manually regulate, and thus reduing running time.

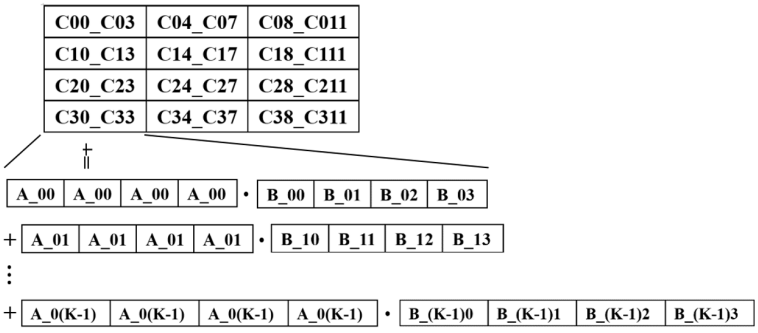

Figure 7. Register tiling. The register is 256-bit length which can load up to 4 double-type variables.

Loop unrolling

The goal of loop unrolling is to improvement running speed of a program by reducing cost in loop control and, combining with register tiling, reducing cost of load/store operations which is the mainly trade-off of computing sub-matrix multiplication. This optimization actually shows a boost in performance as shown in Figure 8 which in turn proves that load/store operations of register cost heavily in this problem. We

implement this strategy simply by duplicating outer loop body(which then reducing outer iteration steps), which typically menas loading several sub-vectors of target matrix **C** into corresponding registers at one time and thus those registers used for computing C can be reused for many times without reloading at each iteration until completing computation. Also since there're totally 16 registers, we also tried to unrolling inner loop to some extent by doing same things on matrix **A** or **B**.

Algorithm 2 Loop Unrolling (our implementation)

Require: Matrix $A^{M\times K}, B^{K\times N}, C^{N\times K}$, where $M \bmod 4 = 0, N \bmod 12 = 0$; 16 256-bit registers available

```
for i = 1, 2, ..., N/4 do
    for j = 1, 2, ..., M/12 do
        Load C[4i, 12j : 12j+3] ... C[4i+3, 12j : 12j+3], C[4i, 12j+4 : 12j+7]
    ... C[4i+3, 12j+4 : 12j+7] and C[4i, 12j+8 : 12j+11] ... C[4i+3, 12j+8 :
    12j+11] into 12 different registers
        for k = 1, 2, ..., K, do
            Load B[k, 12j : 12j+3], B[k, 12j+4 : 12j+7], B[k, 12j+8 : 12j+11]
        into 3 registers
            for ii = 1, 2, 3, 4, do
                Fill a register with A[4i + ii, k]
                Compute and update all c vectors in register
            end for
        end for
    end for
    Store register values of updated c vectors into cache
end for
return C
```

In our final implementation, considering limitation of register number and register tiling strategy, we realize a \\(4 \times 12\\) tiling to fully utilize the 16 registers as shown in Figure 7. In detail, in outer loop totally 12 registers are utilized for 12 \\(c\\) vectors of size \\(1 \times 4\\) at each iteration; in inner loop, 3 \\(b\\) vectors and 1 \\(a\\) vector are loaded into registers and the \\(b\\) vectors are reused within one iteration while \\(a\\) vector reloads for 4 times.

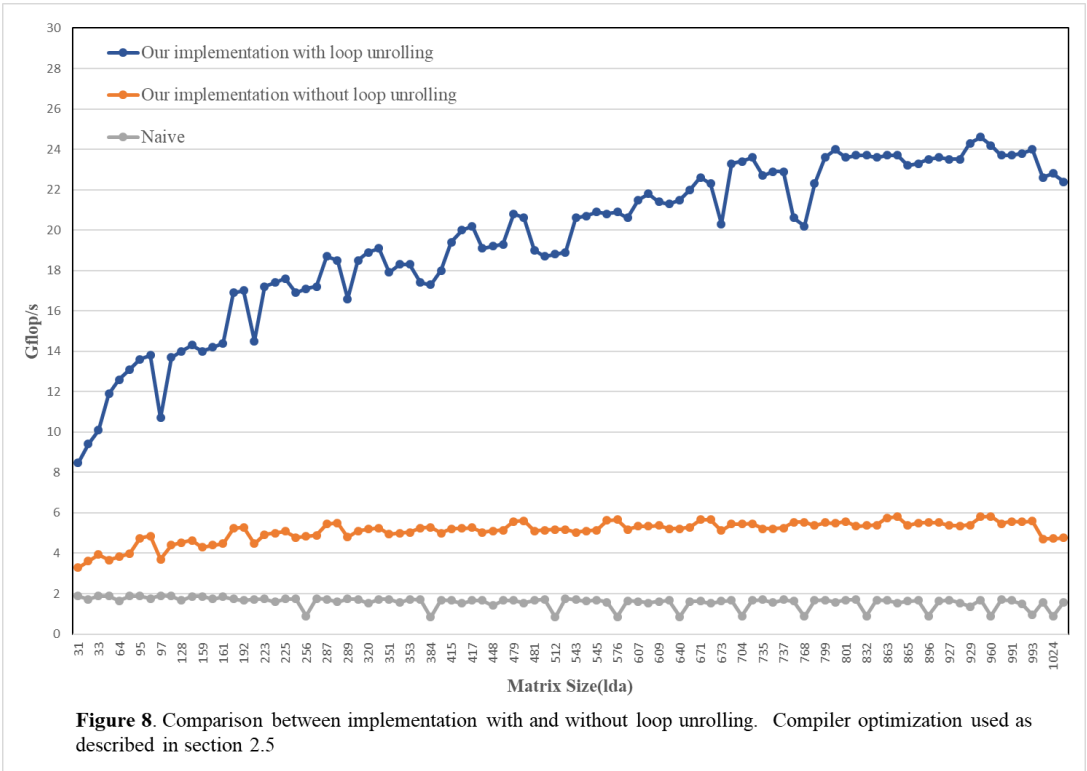

Figure 8. Comparison between implementation with and without loop unrolling. Compiler optimization used as described in section 2.5

2.3.1 Insight of loop unrolling

The reason for great improvement in performance, as briefly described above, is that it reduces register load and store cost to a large extent. Supposing we are computing \\(C ^{8 \times 8} = A^{8 \times 8} \cdot B^{8 \times 4} \\), without loop unrolling, we use 3 registers to store vectors of A, B and C respectively and update

the content of registers at each iteration, which in sum requires \\(8 \times (2 \times (1 + 4 \times 8)) = 528\\) loads and \\(8 \times 2 = 16\\) stores. But if we unroll the loop, for example, by duplicating body of outer loop of 8 times (which actually eliminates the most outer loop), the program would require only \\(8 \times 2 + 2 \times (4 \times 8) = 80 \\) loads and \\(8 \times 2 = 16\\) stores. Because we load 8 \\(c\\) vectors one time, each \\(a\\) vector and \\(b\\) vector can be reused for 8 times without reloading, which results in highly improvement.

2.3.2 Case study

During implementation, we tried several different matrix tiling methods. The key idea is to use as many registers as possible and minimize the load/store operations. As mentioned before,in our final implementation, we realize a \\(4 \times 12\\) tiling to fully utilize the 16 registers as shown in Figure 7. A previous design was a \\(8 \times 4\\) tiling which loads 8 \\(c\\) vectors and 1 \\(b\\) vectors once and 4 \\(a\\) vectors twice within each iteration, which uses totally 13 registers. Here we compare these two tiling to futher show how register tiling, combined with loop unrolling, contributes to the performance.

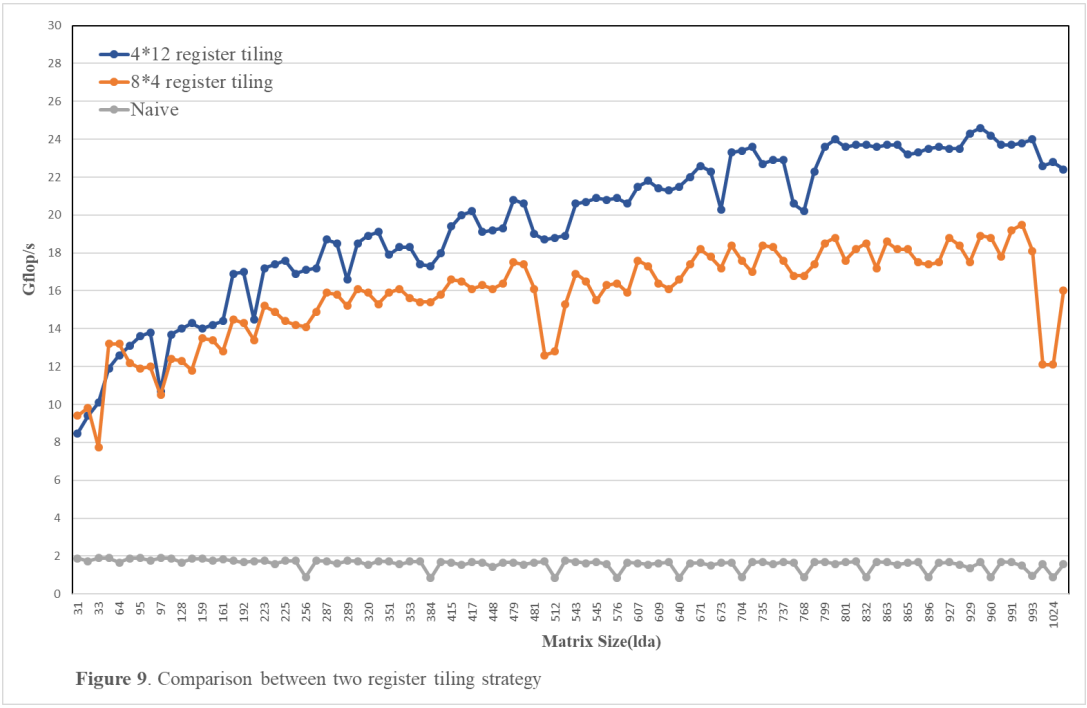

Figure 9. Comparison between two register tiling strategy

As shown in Figure 9, \\(4 \times 12\\) tiling demonstrates more and more significant advantages over \\(8 \times 4\\) tiling as matrix size grows. Considering the analysis of loop unrolling in 2.3.1 section and algorithm described previously, we know that as to the \\(4 \times 12\\) tiling for a \\(C^{M \times N} = A^{M \times K} \cdot B^{K \times N} \\) problem, it toally requires \\((M/4 \cdot N/12) \cdot [12 + (K \cdot (3 + 4)] = (M \cdot N) /4 + (7 \cdot M \cdot N \cdot K)/48 \\) register load operations, while as to the \\(8 \times 4\\) tiling, the number is \\((M/8 \cdot N/4) \cdot [8 + (K \cdot (8 + 1)] = (M \cdot N) /4 + (9 \cdot M \cdot N \cdot K)/32 \\). Hence the \\(4 \times 12\\) tiling needs \\((13 \cdot M \cdot N \cdot K)/96 \\) fewer register load operations, which in return results in the performance gap as matrix size grows up.

2.4 Removing False Dependency (restrict keyword)

As to C99 standard, *restrict* is a new keyword used in pointer declarations which is a declaration of intent given by the programmer to the compiler. It says that for the lifetime of the pointer, only the pointer itself or a value directly derived from it (such as pointer + 1) will be used to access the object to which it points. This limits the effects of pointer aliasing, aiding optimizations. Obviously, in this simple matrix mutiplication

problems, we don't have any pointer aliasing and thus all pointers declared in the program can be aided with a *restrict* keyword so that the improvement can be further improved. The following table demonstrates the effects of this keyword.

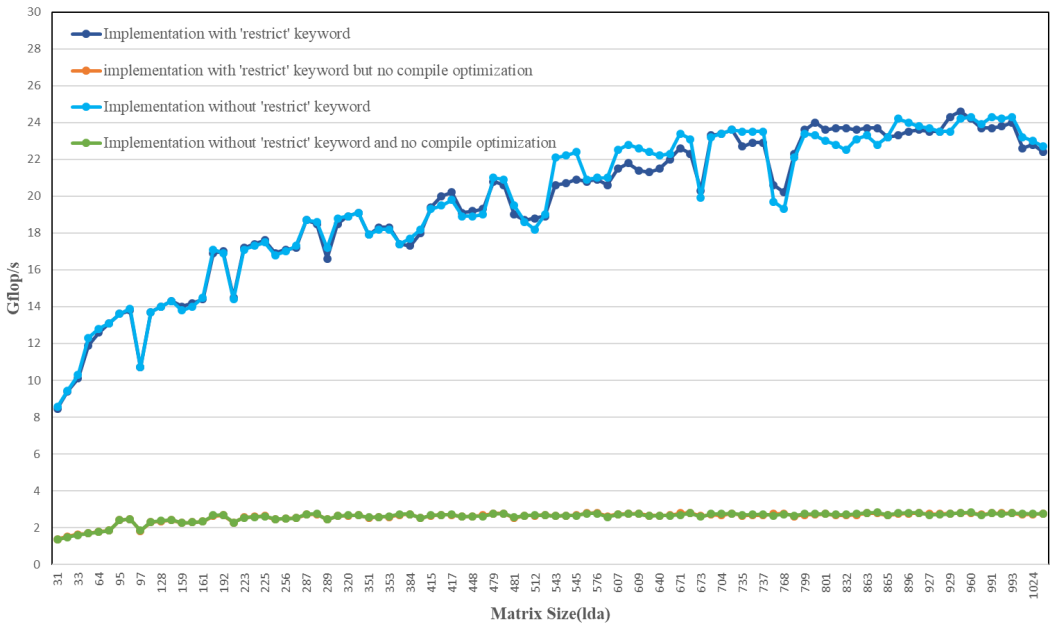

Figure 10. Effect of 'strict' keyword on performance.

Unfortunately, from our experiments, hardly any improvement this keyword makes. At first we guessed that it was because the compiler optimization option auto fixed false dependency, but after we removed compiler optimization options, it's still hard to tell that the *restrict* keyword works. Perhaps we wrongly use this keyword, or false dependency does not really matter a lot in this problem. But after all, we keep the *restrict* keyword in our final implementation (at least it does not harm).

2.5 Complier Optimization

Compiler plays a very important role in programming performance. All the optimization strategy cannot work very well without turning on optimization of compiler. Actually the effects of compiler optimization is really marvelous that almost increases the performance by 8-9 times. As shown in Figure 11, after implemented all ideas mentioned above, the performance does not boost until we enable the compiler optimization option, and thus the performance is getting very close to Atlas implementation, and even better at some periods. Although there're so many options for optimization, in our final implementation, we use only a few, as shown below, which gives the most improvement according to experiments.

make MY_OPT="-O4 -DMY_NEW_CODE -avx2"

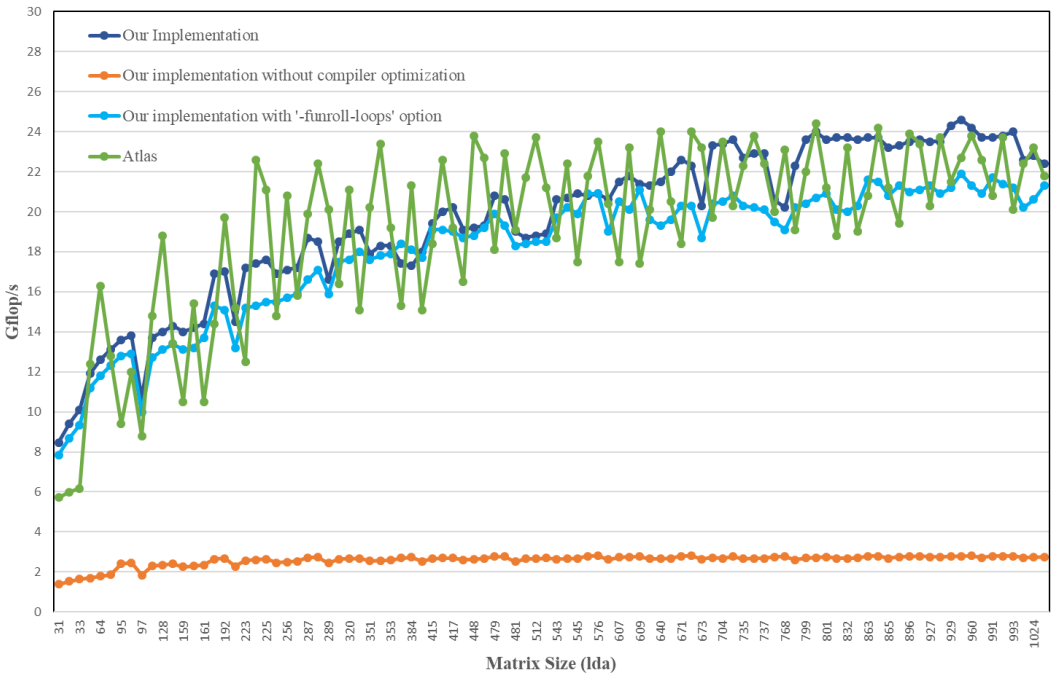

Figure 11. Performance comparison of compile optimization.

Interestingly, we did try other optimization options, but did not see much improvement and even decline. For example, since we paded the input matrixes by copying them to new arrays, we would want to enable the '*-funroll-loops*' options when compling which can unroll the loops in padding step. However, as shown in Figure 11, this instruction does not help at all. This is strange and we guess the reason might be unrolling a too large loop result in lots of jamming, which redues the parallelism of program.

Section 3. Problem and Future Work

Though overall our implementation reaches a high performance on matrix mutiplication very close to Atlas, there're still some problems and further work can be done.

1. If we look closely at the figures, we will see that there exists significant dips at certain matrix sizes, like 511, 512, 1023, 1024. As discussed on piazza, we realize it's because these matrix size reaches the edge of pages and thus there're significantly higher dTLB miss rate when size=1024 compared to 1025, which is caused by the super alignment between page access that fools the 4-way associative data TLB cache. (8B * 1024/row = 8K/row = 2 pages/row). Therefore accessing Row 1 and Row 3 will lead to the same cache set. As a tile usually contains 32 rows, TLB conflict is quite likely to happen. We think this problem can probably solved by compile the program in 'compact' mode, but haven't tried it. Also, there may be some other ways to reduce the effects of TLB miss rate at these points, which can be further studied.
2. AVX512 is also a further optimization choice, which is sure to make improvement because larger registers are available which means much fewer load/store cost.
3. Finally, if time permits, we would like to study more about the compiler optimization. Because from experimental results, we fing it contributes the most to our work, but we know really little about how it works and why it works. There're so many options to be selected. We know functions of part but have no idea about the rest. If we can get deep knowledge of compiler optimization and write code to cooperate with it, the program will be optimized further.

Reference

1. Markus Puschel, "How to Write Fast Numerical Code", https://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring15/slides/10-simd.pdf (https://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring15/slides/10-simd.pdf) describes vectorization with SSE (paper Chellappa, Franchetti, Puschel, "How to write Fast Numerical Code: A Small Introduction"
2. Lam et al on, "The Cache Performance and Optimizations of Blocked Algorithms", https://suif.stanford.edu/papers/lam-asplos91.pdf (https://suif.stanford.edu/papers/lam-asplos91.pdf)
3. Brian Van Straalen's notes on vectorizing with SSE intrinsics
4. Overview of AVX: https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions (https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions)
5. Koharchik and Jones, "An Introduction to GCC Compiler Instrinsics in Vector Processing", Linux Journal http://www.linuxjournal.com/content/introduction-gcc-compiler-intrinsics-vector-processing?page=0,0 (http://www.linuxjournal.com/content/introduction-gcc-compiler-intrinsics-vector-processing?page=0,0)