Project Objective

Our project handles matrix-matrix multiplication on a single processor. Our goal is to optimize the computation performance and maximize the arithmetic intensity in the calculation. Our approach is based off the naive method and try modification strategies such as dividing the problem into blocks.

Logic – Overview

Generally, multiplication between an i-by-k matrix and a k-by-j matrix of double precision floating point numbers takes $2ijk$ FLOPS in total. If two matrices are square and of size n, the FLOPS count is $2n^3$.[1] Therefore to increase the overall arithmetic intensity, judging for the definition

$$A.I. = \frac{\text{\# FLOPS}}{\text{\# Bytes transfer}}$$

our ultimate goal is to reduce the volume of data transferred between memory and cache.
To have a better understanding of the processor we're working with, we collected some key data of the computing node of the cluster as follows[2]

Vector size per register 256 bits

Number of vectors per core 4

Size of cache line 64B

L1 cache size 32 KB

L2 cache size 256 KB

L3 cache size 15 MB (estimate)

To avoid unnecessary memory accesses, a reasonable and viable way is to do matrix multiplication by blocks, in which we divide the matrices into smaller sized blocks, such that a few blocks can fit in the cache to finish the computation.
Since each double precision floating point number takes 8 Bytes (64 bits) to store, and to finish a matrix-matrix multiplication $C = A \cdot B$ we need to fit three matrices of identical dimension in the cache, we could find that the maximum size each level of cache can accommodate are:

L1 cache size

$$\sqrt{\frac{32 \cdot 1024}{8 \cdot 3}} = 36.9$$

L2 cache size

$$\sqrt{\frac{256 \cdot 1024}{8 \cdot 3}} = 104.5$$

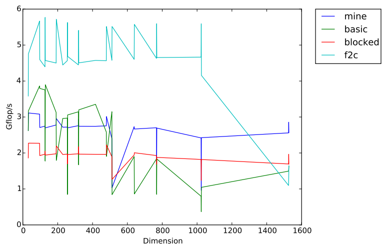
L3 cache size

$$\sqrt{\frac{15 \cdot 1024^2}{8 \cdot 3}} = 809.5$$

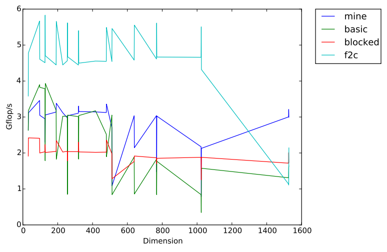
However, in practice, since we want to maximize the efficiency of passing data from memory to cache by making full use of each cache line read, we want the dimension of the matrices to be multiples of cache line size, which is 64B or 8 floating point numbers. Thus the actual most reasonable size to fit in L1, L2, and L3 caches are 32, 96, 800, respectively.

[1]In terms of efficiency, we are not considering models like Strassen algorithm, which are more efficient but too complicated to optimize under the scope of this course.

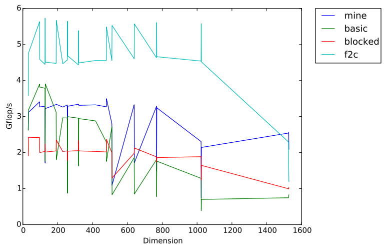
[2]We used the `membench` generated line plot to observe the size of each level of the cache.



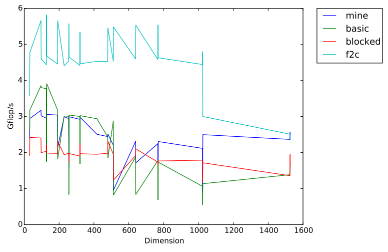
(a) Block size = 32



(b) Block size = 48



(c) Block size = 64



(d) Block size = 96

Figure 1: Plots for different block sizes

Varying Kernel Loop Orders

One of the most important factors influencing the overall performance of block matrix multiplication is the kernel algorithm. In the C code we assume the matrices are in column-major order, which means a matrix is stored in a one-dimension array, and the $(j * n + i)$th element represents `M[i][j]`. When handling $A \cdot B$, we compute the inner product of all rows in A and all columns in B. Therefore to minimize the times of memory access, a reasonable way is to keep a column in B unchanged in memory while going through all rows of A. Thus we came up with this code for the kernel:

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

Note that the size of matrices computed in this kernel should be reasonably small. We wish to make the best efficiency in utilizing all four vectors fully loaded with data, which means the size we are thinking about should be multiples of 4 (each vector can hold 256 bits, which is 4 floating point numbers).

Tuning Block Sizes

So far we have tuned the block size. Based on our analysis, L1 cache should be able to accommodate three matrices of 48, and L2 cache should be able to accommodate matrices of 96. Judging from our plots, we observed that the performance of 16, 32, 48 or 64 varies, but none of them exhibits outstanding performance. The comparison of performances are in Figure 1.
It can also be observed that with Block size 96, the performance is worse than the others. Since L1 cache is

Tuning Block Sizes continued on next page...

8-way set associative, and we have 32 KB for L1-cache, with cache line of 64B, which leads to

$$\frac{32 \cdot 1024\text{B}}{64\text{B} \cdot 8} = 64\text{lines}$$

which means if a column of a matrix spans 96 lines, one cache read wouldn't be able to read the whole column, thus resulting in the poorest performance among the four.

Code and Plots

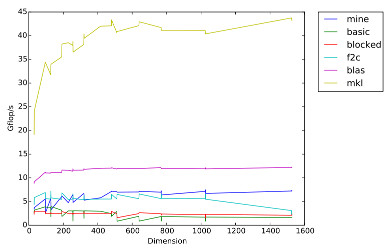
- We do not have individual trials for varying loop orders. All approaches are combined with blocking
- `dgemm_mine2.c` is our code for this approach.
- `dgemm_mine<n>_IJKijk.c` files are our code for different attempts, where n is an identifier of which step we are at (for instance, in this section $n = 2$), and `IJKijk` or any other order of I, J, K, i, j, k indicates the exact loop order.
- Figure 1 are selected plots for this section.

Copy Optimization

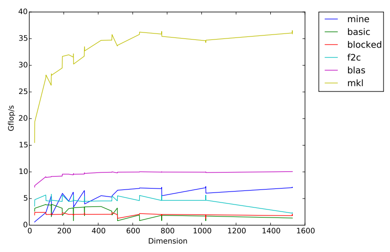
How we would like to revisit our loop ordering approach. Based on our analysis, fixing j, or putting j in the outermost loop should best facilitate reading columns of data. However the disappointing performance drives us to reconsider the mechanism of C accessing a column of such matrix.
Hypothetically when the matrix is created, it should be stored as a 1D array where all elements' address are adjacent. However without knowing the compiling details, the compiler might have to use a safer approach – reading individual elements instead a column as a whole. If so, our effort will not save any extra memory access.
To fix this, we will take advantage of the memory allocation function in C, `malloc` or `_mm_malloc`, to enforce a row or a column of elements stored in adjacent spaces, and more importantly, make the compiler aware of this by using the funcion `__assume_aligned`.
In terms of the cost of memory allocation. The worst case performance would be $\mathcal{O}(n^2)$. However in comparison with the $\mathcal{O}(n^3)$ overall cost, this operation would not be a burden to us.
Figure 2 illustrates our attempts with various blocking size (32, 48, 64, 96) and the copy optimization idea. Notably the performance of our code consistently reaches around 7 GFLOPs/sec when the matrix size is large. The fact that our code outperforms the `dgemm_f2c` approach further proves the success of the copy optimization idea.
Note that among the varied block sizes, 32 shows better result than the other ones. This observation matches our analysis in the previous section, where we summarize that the L1-cache should best accommodate three matrices of 32. The lower FLOP rates for trials with other sizes support our conclusion.

Code and Plots

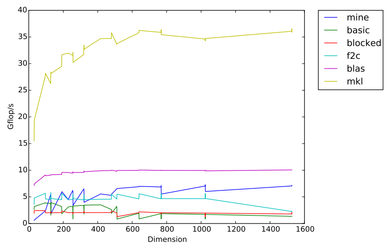
- `dgemm_mine3.c` is our code for this approach.
- In the code, the constant `BLOCK_SIZE` allows varying the block size. Right now it is set to the most optimized value 32.
- Figure 1 are selected plots for this section.



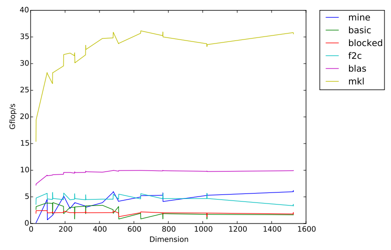
(a) Block size = 32



(b) Block size = 48



(c) Block size = 64



(d) Block size = 96

Figure 2: Plots for different block sizes with copy optimization

Direct Access vs Iterator

Accessing an element in a 1D array `A` by its indices i, j, or k, which is `A + i*BLOCK_SIZE + j`, requires a integer plus and multiplication. We would not think the integer instructions are comparable to double-precision floating point operations, but considering it will potentially operate $\mathcal{O}(n^3)$ times throughout the process, avoiding it would be beneficial to the overall performance.
Hence in this approach we used iterator instead of indices to access an element in an array. Each time we increment the pointer of the block by 1. And under the assumption that all elements are stored in adjacent spaces after copy optimization, we could guarantee this method is safe.
The result is illustrated in Figure 3. When the block size is set to 32, the average result we receive from last section is around 7.0 GFLOPS/sec, whereas after using the iterator, the performance increases to 7.3 GFLOPS/sec (average of three trials). The increase is not significant, but good enough to prove that we made a reasonable move.

Code and Plots

- `dgemm_mine4.c` is our code for this approach.
- `timing_mine.csv` is the spreadsheet of FLOPS rate of this approach. This is also our best performance among all attempts
- Figure 3 is the plot of this approach.
- `timing_cp_opt_best.pdf` illustrates a peak performance of around 9 GFLOPS/sec, which is definitely our highest. However we could not repeat this result.

Conclusion and Further Thoughts

In this project, we studied the performance of matrix-matrix multiplication on a single core. We started from an naive approach, where the peak FLOPS rates were less than 3 GFLOPS/sec. We investigated several approaches, including dividing into sub-blocks, varying loop order of kernel and blocks, and copy

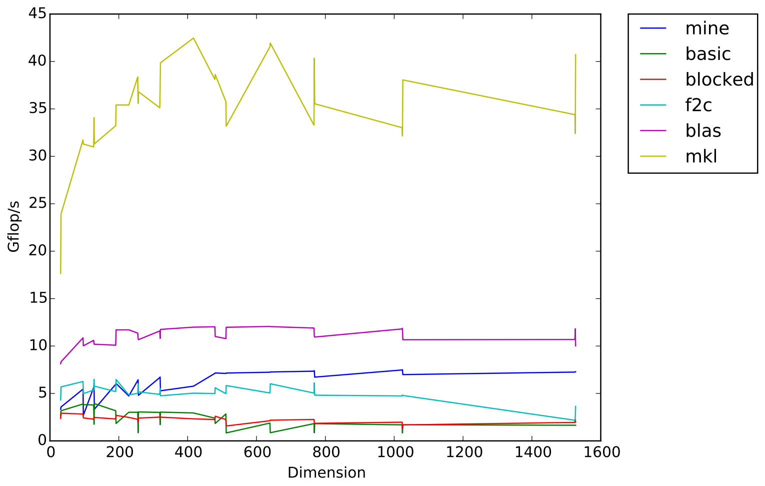


Figure 3: Plots for copy optimization with iterator

optimization. In the end we managed to enhance the overall performance to around 7.3 GFLOPS/sec. The conditions we finalize are:

- Divide into blocks of 32, which fits best the L1-cache of the cluster node we work on;
- Alter the loop order of the kernel such that each row may be reused as much as possible;
- Apply copy optimization, which significantly enhances the performance;
- Employ iterator, which slightly increase the FLOPS rate.

There are a few other approaches that we would love to try, including testing different compiler flags and optimization levels. We note that we haven't intensively investigated the kernel algorithm, which certainly contains huge potential to be improved. In the end, there are opportunities of combining different algorithms together, such as incorporating Strassen method in some part of the algorithm. This would remain further study.

Acknowledgement

We received much help and constructive discussion with

Prof. David Bindel

Colin Ponce

Eric Lee

Junteng Jia and Eric Gao (Group 22)

Ji Kim and Cem Iskir (Group 8)

Ze Jin, Sam Tung and Patrick Cao (Group 9)

We would gratefully appreciate those people's generous help and suggestions.