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:

Vector size per register 256 bits

Number of vectors per core 4

Size of cache line 64B

L1 cache size 64 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

$$\sqrt{\frac{64 \cdot 1024}{8 \cdot 3}} = 52.3$$

L2 cache size

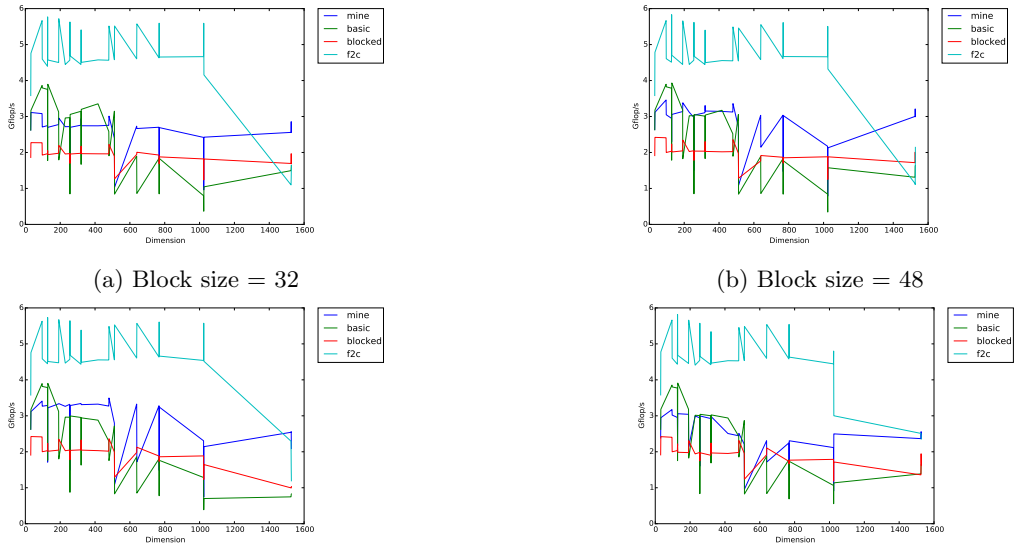
$$\sqrt{\frac{256 \cdot 1024}{8 \cdot 3}} = 104.5$$

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 48, 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.



(a) Block size = 32

(b) Block size = 48

(c) Block size = 64

(d) Block size = 96

Figure 1: Plots for different block sizes

Logic – Matrix Multiplication Kernel

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:

```
1       for (int j=0; j<n; j++) {
2           for (int i=0; i<n; i++) {
3               for (int k=0; k<n; k++) {
4                   C[j*n+k] = C[j*n+k] + A[k*n+i]*B[j*n+k];
5               }
6           }
7       }
```

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).

Priliminary Results

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 48 (block size) is slightly better than 16, 32 or 64. The comparison of performances are in Figure 1.

Next Steps

There are a few things we plan to do to enhance the performance of our code.

1. Nested block multiplication.
   We are going to set up three hierarchy of nested blocks to fit in three layers of caches. Basically any large scale matrices could be divided into blocks of size 800, and each of the block could be further divided into blocks of size 96, which then can be divided into blocks of size 48.

2. Transpose the left matrix. One issue with our model is the cost of obtaining a row of A is quite expensive. A viable way to avoid this cost is to pre-compute the transpose of A, which should be $mathcal O(n^2)$ cost. However this potentially allows we get access to rows of A much easier, as rows of A are columns of A^T, in which elements are stored closely in memory.

3. Employ the built-in function `_assume_aligned()`, which supposedly notifies the compiler that the data entries are closely located in the memory. This should avoid unnecessary cost of repeated reading from memory, particularly for instance a row in A^T in the aforementioned plan.