ECS231 Homework 2 PartII

Jiaping Zhang[*]
University of California, Davis

April 2016

1 Problem 1: Block Matrix-Matrix Multiplication

1.1 Approach

First, we compare the basic matrix-matrix multiplication and block matrix-matrix multiplication to know how their performances are.Then, we try different block sizes to see whether there is an optimized block size for the block matrix-matrix multiplication.

1.2 Results

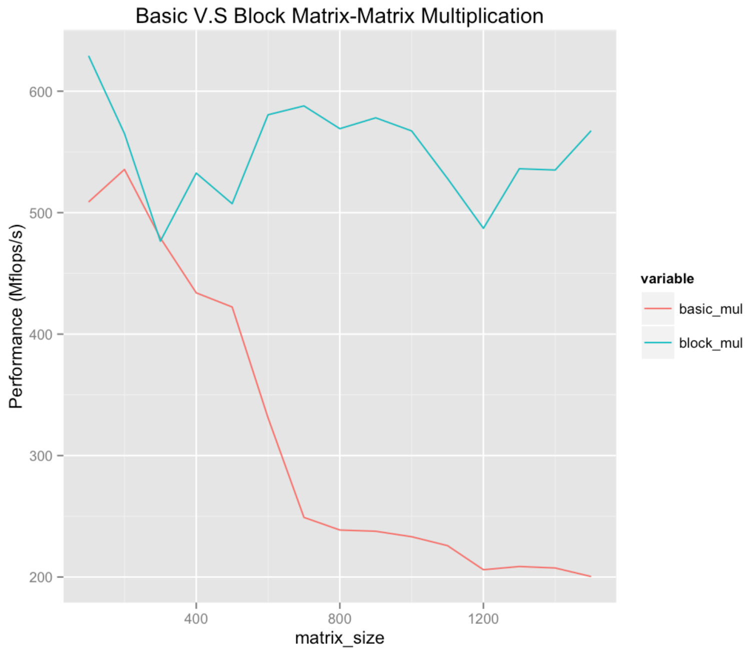

Figure 1: Compare Basic and Block Matrix-Matrix Multiplication

1.3 Analysis

From figure 1, we can see that the block implementation of matrix-matrix multiplication has much better performance than the naive implementation, especially when the matrix size is getting larger. The performance of the basic implementation drops significantly after the matrix size is larger than 200. Thus, it

[*]SID: 999240764

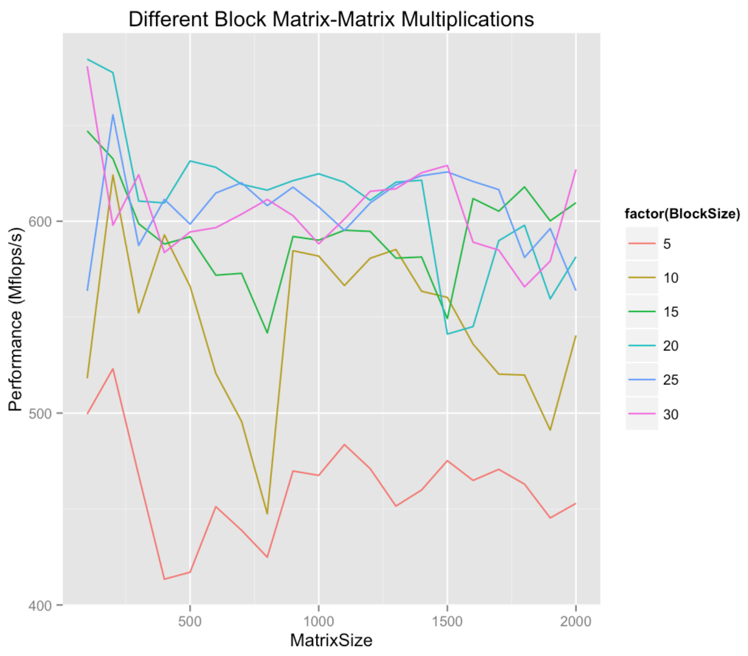

Figure 2: Different Block Size Matrix-Matrix Multiplication

confirms that the block matrix-matrix multiplication is much more efficient due to the better leverage of the cache to exploit locality in the memory hierarchy.

From Figure 2, we can see that there is significant difference of performance when the block size is small, i.e 5,10. The performance improves and becomes more stable when the block size is around 15. However, the performance doesn't improve a lot when the matrix size is larger than 20. This suggests that the block size of 20 is sufficient to get desirable and efficient performance.

2 Reference (code)

2.1 Problem 1

```
####################C code########
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>


void square_dgemm_basic (const unsigned M, const double *A, const double *B, double *C)
{
  unsigned i, j, k;

  for (i = 0; i < M; ++i) {
       for (j = 0; j < M; ++j) {
            const double *Ai_ = A + i;
            const double *B_j = B + j*M;

            double cij = *(C + j*M + i);

            for (k = 0; k < M; ++k) {
                 cij += *(Ai_ + k*M) * *(B_j + k);
            }

            *(C + j*M + i) = cij;
       }
  }
}

void
basic_dgemm (const unsigned lda,
             const unsigned M, const unsigned N, const unsigned K,
             const double *A, const double *B, double *C)
{
  unsigned i, j, k;

  /*
    To optimize this, think about loop unrolling and software
    pipelining.  Hint:  For the majority of the matmuls, you
    know exactly how many iterations there are (the block size)...
  */

  for (i = 0; i < M; ++i) {
       const double *Ai_ = A + i;
       for (j = 0; j < N; ++j) {
            const double *B_j = B + j*lda;

            double cij = *(C + j*lda + i);

            for (k = 0; k < K; ++k) {
                 cij += *(Ai_ + k*lda) * *(B_j + k);
            }
```

```
          *(C + j*lda + i) = cij;
       }
   }
}

#define BLOCK_SIZE ((unsigned) 16)
void do_block (const unsigned lda,
          const double *A, const double *B, double *C,
          const unsigned i, const unsigned j, const unsigned k)
{
     /*
       Remember that you need to deal with the fringes in each
       dimension.

       If the matrix is 7x7 and the blocks are 3x3, you'll have 1x3,
       3x1, and 1x1 fringe blocks.

             xxxoooX
             xxxoooX
             xxxoooX
             oooxxxO
             oooxxxO
             oooxxxO
             XXXOOOX

       You won't get this to go fast until you figure out a 'better'
       way to handle the fringe blocks.  The better way will be more
       machine-efficient, but very programmer-inefficient.
     */
     const unsigned M = (i+BLOCK_SIZE > lda? lda-i : BLOCK_SIZE);
     const unsigned N = (j+BLOCK_SIZE > lda? lda-j : BLOCK_SIZE);
     const unsigned K = (k+BLOCK_SIZE > lda? lda-k : BLOCK_SIZE);

     basic_dgemm (lda, M, N, K,
                  A + i + k*lda, B + k + j*lda, C + i + j*lda);
}

void square_dgemm_block (const unsigned M, const double *A, const double *B, double *C)
{
     const unsigned n_blocks = M / BLOCK_SIZE + (M%BLOCK_SIZE? 1 : 0);
     unsigned bi, bj, bk;

     for (bi = 0; bi < n_blocks; ++bi) {
          const unsigned i = bi * BLOCK_SIZE;

          for (bj = 0; bj < n_blocks; ++bj) {
               const unsigned j = bj * BLOCK_SIZE;

               for (bk = 0; bk < n_blocks; ++bk) {
                    const unsigned k = bk * BLOCK_SIZE;

                    do_block (M, A, B, C, i, j, k);
               }
```

```
            }
        }
    }

    int main(int argc, char *argv[])
    {
      int N = atoi(argv[1]);
      double *A = (double *)malloc(N * N * sizeof(double));
      double *B = (double *)malloc(N * N * sizeof(double));
      double *C1 = (double *)malloc(N * N * sizeof(double));
      double *C2 = (double *)malloc(N * N * sizeof(double));
      unsigned i, j;

      /* You'll definitely change this... */

      int seed = time(NULL);
      srand(seed);

      for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            A[i*N+j] = rand();
            B[i*N+j] = rand();
        }
      }

      clock_t t1;
      t1 = clock();
      square_dgemm_basic(N,A,B,C1);
      t1 = clock() - t1;
      double time_taken1 = ((double)t1)/CLOCKS_PER_SEC;
      double perf1 = 2*pow((double)N,3)/time_taken1;

      clock_t t2;
      t2 = clock();
      square_dgemm_block(N,A,B,C2);
      t2 = clock() - t2;
      double time_taken2 = ((double)t2)/CLOCKS_PER_SEC;
      double perf2 = 2*pow((double)N,3)/time_taken2;

      printf("%f,%f\n",perf1,perf2);

      return 0;
    }

    ############ R code for ploting #########
    library(ggplot2)
    library(reshape2)
    #1
    flops = read.csv('basic_block_compare.csv',header=F)/10^6
    sizes = seq(100,1500,100)
    perf.dt = data.frame(matrix_size=sizes,basic_mul=flops[,1],block_mul=flops[,2])
    nperf.dt = melt(perf.dt,id='matrix_size')
    ggplot(data=nperf.dt,aes(x=matrix_size,y=value,colour=variable))+
      geom_line()+scale_y_continuous('Performance (Mflops/s)')+
      ggtitle("Basic V.S Block Matrix-Matrix Multiplication")

    #2
    blocks = read.csv('block_size_compare.csv',header=F)
    blocks[,3] = blocks[,3]/10^6
    names(blocks) = c('BlockSize','MatrixSize','Performance')
    ggplot(blocks, aes(x = MatrixSize, y = Performance)) +
      geom_line(aes(color = factor(BlockSize)))+scale_y_continuous('Performance (Mflops/s)')+
      ggtitle("Different Block Matrix-Matrix Multiplications")


    ############## Shell Scripts #########
    for n in $(seq 100 100 1500);do ./test_main $n ; done > basic_block_compare.csv
    for s in $(seq 5 5 30); do for n in $(seq 100 100 2000);do ./blocked_dgemm $n $s; done; dor
    > block_size_compare.csv
```