Jonathan Frame
Spring 2019
Pantarhei usr: sp19hpc_jmframe

CS 491/591: High Performance Computing Project 2
Performance Optimization via Cache Reuse Due date: 11:59 pm, February 24th, 2019

Note: You need to upload a pdf report for the project into Blackboard Learn system. Please also upload all your source codes and a makefile as a tar file into Blackboard Learn system so that our grader can verify what you achieved in your report. Please send an email to Jiannan Tian (jtian10@crimson.ua.edu) if you have any issues with your Pantarhei account.

Suppose your data cache has 60 lines and each line can hold 10 doubles. You are performing a matrix- matrix multiplication (**C=C+A*B**) with square matrices of size **10000X10000** and **10X10** respectively. Assume data caches are only used to cache matrix elements which are doubles. The cache replacement rule is ***least recently used first***. Assume no registers can be used to cache intermediate computing results. One-dimensional arrays are used to represent matrices with the row major order.

Summary
Below are my answers to the second high performance computing project. Part one shows that for the unblocked algorithm the cache misses are pretty low for the 10X10 matrix, but pretty high for the 10,000X10,000 matrix. Part two shows that using blocking algorithms significantly reduces the cache misses, and thus can be expected to significantly improve the performance of the matrix multiplication algorithms. Part three shows the results of varying the block size in those matrix multiplication algorithms tested on a 2048X2048 matrix of random floating point numbers. The performance results are dependent on the algorithm, so no single best block size value can be determined, but block size 64 has the lowest minimum run time (for algorithm kij). The results for Part 4 shows that the overall best performance was achieved with algorithm ikj, a block size of 64 and compiled with gcc-5.4.0 and the -O3 optimization flag.

Part 1. (25 points) When matrix-matrix multiplication is performed using the *simple triple-loop* algorithm with single register reuse, there are 6 versions of the algorithm (ijk, ikj, jik, jki, kij, kji). Calculate the **number** of read cache misses for **each** element in **each** matrix for **each** version of the algorithm when the sizes of the matrices are **10000X10000** and **10X10** respectively. What is the percentage of read cache miss for each algorithm?

When the matrix multiplication algorithm begins cycling through the a matrix column-wise there will be a cache miss each time the first column is called and then any time the cache exceeds its size (in this case every ten columns). Any place with a column index in the set {0,9,19…, n-10 ,n} will have a miss each time it is called. As the program cycles through a row-wise matrix, there will always be a cache miss, because it will effectively be like calling a first column. From what I can tell a fixed matrix will act like a row-wise matrix, so it will miss when it is first called, but hit until the cache runs out. Any matrix element that is called in the second loop with be called a total of n^2 times. Any matrix element called in the third loop will be called a total of n^3 times. These generalizations make calculating the cache misses for **each** element in **each** matrix for **each** version of the algorithm fairly straight forward. The calculations are shown below. Note that some of the algorithms have similarities in terms of where the matrices are called (i.e., in the second or third loop), and thus have similar calculations.

Table 1.1 Number of element cache misses in Matrices A, B & C depending on column index.

Algorithm	Matrix	Row or Column wise	Called in loop	Element column index set	Misses 10X10	Misses 10k X 10k
ijk & jik	A	Row	3	{0, 9, 19, …, n-10, n}	1000	10^{12}
ijk & jik	A	Row	3	~{0, 9, 19, …, n-10, n}	0	0
ijk & jik	B	Column	3	{All}	1000	10^{12}
ijk & jik	C	Fixed	2	{0, 9, 19, …, n-10, n}	100	10^{9}
ijk & jik	C	Fixed	2	~{0, 9, 19, …, n-10, n}	0	0
kij & ikj	A	Fixed	2	{0, 9, 19, …, n-10, n}	100	10^{9}
kij & ikj	A	Fixed	2	~{0, 9, 19, …, n-10, n}	0	0
kij & ikj	B	Row	3	{0, 9, 19, …, n-10, n}	1000	10^{12}
kij & ikj	B	Row	3	~{0, 9, 19, …, n-10, n}	0	0
kij & ikj	C	Row	3	{0, 9, 19, …, n-10, n}	1000	10^{12}
kij & ikj	C	Row	3	~{0, 9, 19, …, n-10, n}	0	0
jki & kji	A	Column	3	{All}	1000	10^{12}
jki & kji	B	Fixed	2	{0, 9, 19, …, n-10, n}	100	10^{9}
jki & kji	B	Fixed	2	~{0, 9, 19, …, n-10, n}	0	0
jki & kji	C	Column	3	{All}	1000	10^{12}

The total algorithm miss percentage is the total number of cache misses over the total calls to any element. For instance, if a total of 100 calls to memory are made, regardless of matrix or element, and the total cache misses are 10, 5, and 1 for A, B and C, respectively, then the miss

percentage would be (10 + 5 + 1)/100 = 16%. Using that basis, the calculations are shown below.

Table 1.2 Cache miss percentage based on algorithm.

Algorithm	Matrix A miss rate	Memory calls to A	Matrix B miss rate	Memory calls to B	Matrix C miss rate	Memory calls to C	Total algorithm miss percentage
ijk & jik	1/n	n^3	1	n^3	1/n	n^2	53% = $(n^2 + n^3 + n) / (2n^3 + n^2)$
kij & ikj	1/n	n^2	1/n	n^3	1/n	n^3	10% = $(n + n^2 + n^2) / (2n^3 + n^2)$
jki & kji	1	n^3	1/n	n^2	1	n^3	96% = $(n^3 + n + n^3) / (2n^3 + n^2)$

Part 2. (25 points) If matrices are partitioned into block matrices with each block being a 10 by 10 matrix, then the matrix-matrix multiplication can be performed using one of the 6 ***blocked version algorithms*** (ijk, ikj, jik, jki, kij, kji). Assume the multiplication of two blocks in the inner three loops uses the same loop order as the three outer loops in the blocked version algorithms. Calculate the **number** of read cache misses for **each** element in **each** matrix for **each** version of the blocked algorithm when the size of the matrices is **10000**. What is the percentage of read cache miss for each algorithm?

This problem is very similar to Part 1, and I will use the same generalization to defend my calculations, but I will add on more. Any element that gets called in the inner most loop will have cache misses divided by the block size (i.e., Block size = 10).

Table 2.1 Number of element misses in Matrices A, B & C depending on column index, with 10 X 10 blocks for inner loop.

Algorithm	Matrix	Row or Column wise	Called in loop	Element column index set	Misses 10k X 10k
ijk & jik	A	Row	3	{0, 9, 19, …, n-10, n}	10^{11}
ijk & jik	A	Row	3	~{0, 9, 19, …, n-10, n}	0
ijk & jik	B	Column	3	{All}	10^{11}
ijk & jik	C	Fixed	2	{0, 9, 19, …, n-10, n}	10^{9}
ijk & jik	C	Fixed	2	~{0, 9, 19, …, n-10, n}	0
kij & ikj	A	Fixed	2	{0, 9, 19, …, n-10, n}	10^{9}
kij & ikj	A	Fixed	2	~{0, 9, 19, …, n-10, n}	0
kij & ikj	B	Row	3	{0, 9, 19, …, n-10, n}	10^{11}
kij & ikj	B	Row	3	~{0, 9, 19, …, n-10, n}	0
kij & ikj	C	Row	3	{0, 9, 19, …, n-10, n}	10^{11}
kij & ikj	C	Row	3	~{0, 9, 19, …, n-10, n}	0
jki & kji	A	Column	3	{All}	10^{11}
jki & kji	B	Fixed	2	{0, 9, 19, …, n-10, n}	10^{9}
jki & kji	B	Fixed	2	~{0, 9, 19, …, n-10, n}	0
jki & kji	C	Column	3	{All}	10^{11}

Table 2.2 Cache miss percentage based on algorithm with blocks used for inner loop.

Algorithm	Matrix A miss rate	Memory calls to A	Matrix B miss rate	Memory calls to B	Matrix C miss rate	Memory calls to C	Total algorithm miss percentage
ijk & jik	1/(nB)	n^3	1/B	n^3	1/n	n^2	6% = $(n^2/B + n^3/B + n) / (2n^3 + n^2)$
kij & ikj	1/n	n^2	1/(nB)	n^3	1/(nB)	n^3	1% = $(n + n^2/B + n^2/B) / (2n^3 + n^2)$
jki & kji	1/B	n^3	1/n	n^2	1/B	n^3	1% = $(n^3/B + n + n^3/B) / (2n^3 + n^2)$

Part 3. (25 points) Implement the algorithms in part (1) and (2). Report your execution time on our Pantarhei cluster. Adjust the block size from 10 to other numbers to see what the optimal block size is. Compile your code using the default compiler (gcc-7.3.0) on Pantarhei without optimization tag. Compare and analyze the performance of your codes for n=2048. Please always verify the correctness of your code.

There is no single objective optimal block size. For algorithm ijk the optimal size is 32, but by less than half a second over block size 16. Block size 8 has the lowest average run time over all the algorithms, and also has the lowest maximum run time by far. Block size 64 has the lowest minimum run time (for algorithm kij). Block size 128 is overall the worst performing size with the highest average and maximum run times. Of course these values will change slightly each time the program is run.

Compile using gcc-7.3.0: “gcc part3.c -o part3”
Run using batch script (slurm_part3.job):

```
#!/bin/bash
#SBATCH -q defq
#SBATCH -N 1
#SBATCH -n 16
#SBATCH -t 02:00:00
#SBATCH --exclude=gpu01
#SBATCH -J part3
#SBATCH -o results/part3.%j.out
#SBATCH -e results/part3.%j.error
./part3
```

Code for this part of the project is shown in Appendix, and has been uploaded to Blackboard (part3.c). Each matrix multiplication was tested against the dgemm0 algorithm, and the results printout file shows that each algorithm implementation had zero difference from the result.

Table 3.1 Results (reported in run time seconds) from blocked matrix multiplication algorithms.

block size	ijk	ikj	jik	jki	kij	kji	Average	Minimum	Maximum	Range
8	36.9	27.9	39.6	41.1	27.5	40.8	34.3	27.5	41.1	13.6
16	34.3	26.0	34.8	66.3	26.1	54.5	38.8	26.0	66.3	40.4
32	33.9	25.0	34.4	68.4	25.0	67.1	43.3	25.0	68.4	43.4
64	34.8	24.6	35.5	68.5	24.7	67.8	57.6	24.6	68.5	43.9
128	83.3	25.9	62.5	81.2	26.6	79.2	69.5	25.9	83.3	57.4

Part 4. (25 points) Improve your implementation by using both cache blocking and register blocking at the same time. Optimize your block sizes. Compile your code using both the default compiler and gcc-5.4.0 with different optimization flags (-O0, -O1, -02, and -O3) respectively. Compare and analyze the performance of your codes for n=2048. Highlight the best performance you achieved. Please always verify the correctness of your code. Note that you can use "module swap gnu7/7.3.0 gnu/5.4.0" to replace the default compiler by gcc-5.4.0.

This part of the project was very similar to Part 3. Almost all of the code to get the results shown below was the same, except for the actual matrix multiplication functions, shown in Appendix. The code was also uploaded to blackboard, or can be found on Pantarhei here: /home/sp19hpc_jmframe/hw/2/part4.c. The results of this part include numerous permutations for the algorithms. There is no elegant way to present these results, so below is a plethora of tables. The best performances for minimum, average and maximum time are highlighted in the tables. **The overall best performance was achieved with algorithm ikj, a block size of 64 and compiled with gcc-5.4.0 and the -O3 optimization flag.**

Code for this part of the project is shown in Appendix, and has been uploaded to Blackboard. Each matrix multiplication was tested against the dgemm0 algorithm, and the results printout file shows that each algorithm implementation had zero difference from the result.

Tables 4.1 - 4.8 Results of the algorithms implemented for n=2048 with both cache blocking and register blocking. Titled with their compiling options:

Table 4.1 gcc part4.c -o part4_gnu7_O0 -O0

block size	ijk	ikj	jik	jki	kij	kji	Average	Minimum	Maximum	Range
8	17.4	13.5	17.1	21.4	13.5	21.8	17.5	13.5	21.8	8.3
16	12.9	12.7	13.3	15.2	12.8	15.1	13.7	12.7	15.2	2.5
32	12.0	12.3	12.0	14.1	12.4	14.1	12.8	12.0	14.1	2.1
64	12.1	12.2	12.3	14.0	12.4	13.9	12.8	12.1	14.0	1.9
128	23.4	12.4	19.7	31.4	12.5	26.9	21.1	12.4	31.4	19.0

Table 4.2 gcc part4.c -o part4_gnu7_O1 -O1

block size	ijk	ikj	jik	jki	kij	kji	Average	Minimum	Maximum	Range
8	6.9	4.6	7.1	10.4	4.7	9.8	7.3	4.6	10.4	5.8
16	4.8	4.1	5.3	10.6	4.1	9.6	6.4	4.1	10.6	6.5
32	3.9	4.0	4.4	10.1	3.9	9.5	6.0	3.9	10.1	6.1
64	4.0	3.9	4.6	9.8	4.1	9.4	6.0	3.9	9.8	5.9
128	10.8	4.2	8.2	15.4	4.3	13.6	9.4	4.2	15.4	11.2

Table 4.3 gcc part4.c -o part4_gnu7_O2 -O2

block size	ijk	ikj	jik	jki	kij	kji	Average	Minimum	Maximum	Range

8	5.8	4.0	6.2	10.4	3.7	9.1	6.5	3.7	10.4	6.7
16	4.0	3.0	4.8	10.6	2.8	9.5	5.8	2.8	10.6	7.9
32	3.4	2.7	3.8	10.0	2.6	9.5	5.3	2.6	10.0	7.4
64	3.5	2.8	3.9	9.7	2.8	9.4	5.3	2.8	9.7	7.0
128	9.6	3.2	8.2	14.0	3.1	13.4	8.6	3.1	14.0	10.9

Table 4.4 gcc part4.c -o part4_gnu7_O3 -O3

block size	ijk	ikj	jik	jki	kij	kji	Average	Minimum	Maximum	Range
8	5.7	3.5	6.1	9.8	3.7	8.4	6.2	3.5	9.8	6.3
16	4.0	2.8	4.8	10.4	2.7	9.3	5.7	2.7	10.4	7.7
32	3.4	2.6	3.8	10.0	2.6	9.4	5.3	2.6	10.0	7.3
64	3.4	2.6	3.9	9.6	2.8	9.4	5.3	2.6	9.6	7.0
128	9.7	3.1	7.8	13.5	3.1	13.3	8.4	3.1	13.5	10.4

Changed compilers: module swap gnu7/7.3.0 gnu/5.4.0

Table 4.5 gcc part4.c -o part4_gnu5_O0 -O0

block size	ijk	ikj	jik	jki	kij	kji	Average	Minimum	Maximum	Range
8	16.6	13.6	16.7	21.2	13.7	21.6	17.2	13.6	21.6	8.0
16	12.4	12.7	12.6	15.2	12.8	15.2	13.5	12.4	15.2	2.9
32	11.3	12.4	11.4	14.2	12.5	14.3	12.7	11.3	14.3	3.0
64	11.4	12.5	11.6	14.1	12.4	13.8	12.6	11.4	14.1	2.7
128	25.9	12.6	19.9	28.6	12.8	26.9	21.1	12.6	28.6	16.0

Table 4.6 gcc part4.c -o part4_gnu5_O1 -O1

block size	ijk	ikj	jik	jki	kij	kji	Average	Minimum	Maximum	Range
8	7.0	4.7	7.3	10.4	4.5	9.7	7.2	4.5	10.4	5.9
16	4.9	4.1	5.4	10.6	4.1	9.6	6.4	4.1	10.6	6.6
32	4.1	3.9	4.5	10.0	3.9	9.5	6.0	3.9	10.0	6.1
64	4.1	4.0	4.5	9.7	4.0	9.5	6.0	4.0	9.7	5.7
128	11.5	4.1	8.3	15.8	4.2	13.3	9.5	4.1	15.8	11.7

Table 4.7 gcc part4.c -o part4_gnu5_O2 -O2

block size	ijk	ikj	jik	jki	kij	kji	Average	Minimum	Maximum	Range
8	5.8	3.8	6.3	9.9	3.8	8.4	6.3	3.8	9.9	6.1
16	4.1	2.9	4.8	10.5	2.9	9.3	5.7	2.9	10.5	7.6

32	3.4	2.7	3.9	9.9	2.7	9.4	5.3	2.7	9.9	7.3
64	3.5	2.7	3.9	9.7	2.8	9.4	5.3	2.7	9.7	7.0
128	11.9	3.2	8.0	13.5	3.1	13.2	8.8	3.1	13.5	10.3

Table 4.8 gcc part4.c -o part4_gnu5_O3 -O3

block size	ijk	ikj	jik	jki	kij	kji	Average	Minimum	Maximum	Range
8	6.4	4.4	6.9	10.3	3.8	9.2	6.8	3.8	10.3	6.5
16	4.3	2.3	5.1	10.5	2.8	9.4	5.7	2.3	10.5	8.3
32	3.5	1.9	3.9	10.0	2.7	9.4	5.2	1.9	10.0	8.1
64	3.5	1.7	4.0	9.9	2.8	9.5	5.2	1.7	9.9	8.2
128	10.6	2.5	7.8	14.0	3.2	13.5	8.6	2.5	14.0	11.5

Appendix: Code for Part 3

```
//////////////////////     INCLUDE LIBRARIES    ////////////////////
#include <stdio.h>

//Only use one of these
#include <stdlib.h>
//#include <string.h>

#include <stdbool.h>
#include <time.h>
//////////////////

////////////////////////////////////   VOID FUNCTION   ////   VOID FUNCTION   ////////////////////////////////
void dgemm0(double *A, double *B, double *C0, int n){
  int i, j, k;
  for ( i = 0; i < n; i++ ) {
    for ( j = 0; j < n; j++ ) {
      for ( k = 0; k < n; k++ ) {
        C0[i * n + j] += A[i * n + k] * B[k * n + j];
      }
    }
  }
}

////////////////////////////////////   VOID FUNCTION   ////   VOID FUNCTION   ////////////////////////////////
void ijk_blocked(double *A, double *B, double *C1, int n, int Blc){
  int i, j, k, i1, j1, k1;
  for ( i = 0; i < n; i += Blc ) {
    for ( j = 0; j < n; j += Blc ) {
      for ( k = 0; k < n; k += Blc ) {
        // B x B mini matrix multiplications
        for ( i1 = i; i1 < (i + Blc); i1 ++ ) {
          for ( j1 = j; j1 < (j + Blc); j1++ ) {
            register double r = C1[i1 * n + j1];
            for ( k1 = k; k1 < (k + Blc); k1++ ) {
              r += A[i1 * n + k1] * B[k1 * n + j1];
              C1[i1 * n + j1] = r;
            }
          }
        }
      }
    }
  }
}
////////////////////////////////////   VOID FUNCTION   ////   VOID FUNCTION   ////////////////////////////////
void jik_blocked(double *A, double *B, double *C1, int n, int Blc){
  int i, j, k, i1, j1, k1;
  for ( j = 0; j < n; j += Blc ) {
    for ( i = 0; i < n; i += Blc ) {
      for ( k = 0; k < n; k += Blc ) {
        // B x B mini matrix multiplications
        for ( j1 = j; j1 < (j + Blc); j1 ++ ) {
          for ( i1 = i; i1 < (i + Blc); i1++ ) {
            register double r = C1[i1 * n + j1];
            for ( k1 = k; k1 < (k + Blc); k1++ ) {
```

```
                r += A[i1 * n + k1] * B[k1 * n + j1];
                C1[i1 * n + j1] = r;
              }
            }
          }
        }
      }
    }
}
////////////////////////////////////  VOID FUNCTION  ////  VOID FUNCTION  ////////////////////////////////
void kij_blocked(double *A, double *B, double *C1, int n, int Blc){
  int i, j, k, i1, j1, k1;
  for ( k = 0; k < n; k += Blc ) {
    for ( i = 0; i < n; i += Blc ) {
      for ( j = 0; j < n; j += Blc ) {
        // B x B mini matrix multiplications
        for ( k1 = k; k1 < (k + Blc); k1 ++ ) {
          for ( i1 = i; i1 < (i + Blc); i1++ ) {
            register double r = A[i1 * n + k1];
            for ( j1 = j; j1 < (j + Blc); j1++ ) {
              C1[i1 * n + j1] += r * B[k1 * n + j1];
            }
          }
        }
      }
    }
  }
}
////////////////////////////////////  VOID FUNCTION  ////  VOID FUNCTION  ////////////////////////////////
void ikj_blocked(double *A, double *B, double *C1, int n, int Blc){
  int i, j, k, i1, j1, k1;
  for ( i = 0; i < n; i += Blc ) {
    for ( k = 0; k < n; k += Blc ) {
      for ( j = 0; j < n; j += Blc ) {
        // B x B mini matrix multiplications
        for ( i1 = i; i1 < (i + Blc); i1 ++ ) {
          for ( k1 = k; k1 < (k + Blc); k1++ ) {
            register double r = A[i1 * n + k1];
            for ( j1 = j; j1 < (j + Blc); j1++ ) {
              C1[i1 * n + j1] += r * B[k1 * n + j1];
            }
          }
        }
      }
    }
  }
}
////////////////////////////////////  VOID FUNCTION  ////  VOID FUNCTION  ////////////////////////////////
void jki_blocked(double *A, double *B, double *C1, int n, int Blc){
  int i, j, k, i1, j1, k1;
  for ( j = 0; j < n; j += Blc ) {
    for ( k = 0; k < n; k += Blc ) {
      for ( i = 0; i < n; i += Blc ) {
        // B x B mini matrix multiplications
        for ( j1 = j; j1 < (j + Blc); j1 ++ ) {
          for ( k1 = k; k1 < (k + Blc); k1++ ) {
            register double r = B[k1 * n + j1];
```

```
            for ( i1 = i; i1 < (i + Blc); i1++ ) {
              C1[i1 * n + j1] += r * A[i1 * n + k1];
            }
          }
        }
      }
    }
  }
}
////////////////////////////////////  VOID FUNCTION  ////  VOID FUNCTION  ////////////////////////////////
void kji_blocked(double *A, double *B, double *C1, int n, int Blc){
  int i, j, k, i1, j1, k1;
  for ( k = 0; k < n; k += Blc ) {
    for ( j = 0; j < n; j += Blc ) {
      for ( i = 0; i < n; i += Blc ) {
        // B x B mini matrix multiplications
        for ( k1 = k; k1 < (k + Blc); k1 ++ ) {
          for ( j1 = j; j1 < (j + Blc); j1++ ) {
            register double r = B[k1 * n + j1];
            for ( i1 = i; i1 < (i + Blc); i1++ ) {
              C1[i1 * n + j1] += r * A[i1 * n + k1];
            }
          }
        }
      }
    }
  }
}

////////////////////////////////////  VOID FUNCTION  ////  VOID FUNCTION  ////////////////////////////////
void checkresults(double *C0, double *C1, int m, clock_t start_t, clock_t end_t){
  int i;
  // Check that the matrix multiplactions are the same for each
  // NOTE: the matrix multiplications were tested in dgemm0 & Blocked
  double maxDiff = 0;
  double diff;
  for ( i = 0; i < m; i++ ) {
    diff = C1[i] - C0[i];
    if ( abs(diff) > maxDiff ){
      maxDiff = diff;
      printf("ELEMENTS IN MATRIX C0 & C1 ARE NOT THE SAME\t C0[%d] = %f\t C1[%d] = %f\n", i,C0[i], i, C1[i]);
    }
  }
  double total_t = ((double)(end_t - start_t)) / CLOCKS_PER_SEC;
  printf("Total matrix multiplication run time: %f seconds\n", total_t);
  printf("The maximum difference between blocked and unblocked is: %lf\n",maxDiff);
  printf("-----------------------------------------------------------------\n");
}
////////////////////////////////////  VOID FUNCTION  ////  VOID FUNCTION  ////////////////////////////////
void clearmatrixC(double *C1, int n){
  int i, j;
  // Clear the C matrices, to make sure we dont cross over results.
  for ( i = 0; i < n; i++ ) {
    for ( j = 0; j < n; j++ ){
      C1[i * n + j] = 0;
    }
  }
```

```
}


////////////////////////////////////////////////////////////////////////////////////////////////
// MAIN Program, which sets up and executes ////////////////////////////////////////////////////
// matrix multibplications for blocked and ////////////////////////////////////////////////////
// unblocked algorithms.///////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////
int main () {
  int m, n, i, j, k, Blc;
  clock_t start_t, end_t;

  // MATRIX SIZE
  n=2048;
  // Number of total elements in the matrices, also the length of the
  // one dimensional vectors to do the matix multiplications.
  m = n*n;
  printf("-----------------------------------------\n");
  printf("The matrix size is %d by %d \n", n,n);

  //setting up an array with x rows and y columns
  double *A, *B, *C0, *C1;
  A = malloc(m * sizeof *A);
  B = malloc(m * sizeof *B);
  C0 = malloc(m * sizeof *C0);
  C1 = malloc(m * sizeof *C1);

  ////////////////////////////////////////////
  // SETTING UP THE MATRICES FOR MULTIPLICATION
  for ( i = 0; i < n; i++ ) {
    for ( j = 0; j < n; j++ ){
      A[i * n + j] = (double)rand()/RAND_MAX*2.0-1.0;
      B[i * n + j] = (double)rand()/RAND_MAX*2.0-1.0;
      C0[i * n + j] = 0;
      C1[i * n + j] = 0;
    }
  }

  ////////////////////////////////////////////
  // RUN THE BASIC MULTIPLICATION SCRIPT
  // TO ENSURE RESULTS OF BLOCK MULTIPLICATION
  // Time this program, to use for efficiency tests
  dgemm0(A, B, C0, n);

  printf("-------------------------------------------------\n");
  printf("Looping through block sizes.\n");
  printf("Running each algorithm for each block size\n");
  printf("-------------------------------------------------\n\n\n\n\n");

  ////////////////////////////////////////////////////////////////
  // LOOP THROUGH THE BLOCK SIZE TO FIND OPTIMIZE
  ////////////////////////////////////////////////////////////////
  for ( Blc = 8; Blc <=128; Blc += Blc ) {
    ////////////////////////////////////////////////////////////
    // One of these for each algorithm
    printf("BLOCK SIZE: %d, ALGORITHM: ijk\n", Blc);
    start_t = clock();
```

```
      ijk_blocked(A, B, C1, n, Blc);
      end_t = clock();
      checkresults(C0, C1, m, start_t, end_t);
      clearmatrixC(C1, n);
      printf("BLOCK SIZE: %d, ALGORITHM: ikj\n", Blc);
      start_t = clock();
      ikj_blocked(A, B, C1, n, Blc);
      end_t = clock();
      checkresults(C0, C1, m, start_t, end_t);
      clearmatrixC(C1, n);
      printf("BLOCK SIZE: %d, ALGORITHM: jik\n", Blc);
      start_t = clock();
      jik_blocked(A, B, C1, n, Blc);
      end_t = clock();
      checkresults(C0, C1, m, start_t, end_t);
      clearmatrixC(C1, n);
      printf("BLOCK SIZE: %d, ALGORITHM: jki\n", Blc);
      start_t = clock();
      jki_blocked(A, B, C1, n, Blc);
      end_t = clock();
      checkresults(C0, C1, m, start_t, end_t);
      clearmatrixC(C1, n);
      printf("BLOCK SIZE: %d, ALGORITHM: kij\n", Blc);
      start_t = clock();
      kij_blocked(A, B, C1, n, Blc);
      end_t = clock();
      checkresults(C0, C1, m, start_t, end_t);
      clearmatrixC(C1, n);
      printf("BLOCK SIZE: %d, ALGORITHM: kji\n", Blc);
      start_t = clock();
      kji_blocked(A, B, C1, n, Blc);
      end_t = clock();
      checkresults(C0, C1, m, start_t, end_t);
      clearmatrixC(C1, n);
      printf("------------------------------------------------------------------------\n\n");
    }
    printf("--- END PROGRAM ----------------------------------------------------\n");
    printf("-----------------END PROGRAM ----------------------------------\n");
    printf("--------------------------------END PROGRAM----------------------");
    return 0;
}
//End of program
```

Appendix: Blocked functions for Part 4:

```
////////////////////////////////////   VOID FUNCTION   ////   VOID FUNCTION   ////////////////////////////////
void ijk_blocked(double *A, double *B, double *C1, int n, int Blc){
  int i, j, k, i1, j1, k1;
  for ( i = 0; i < n; i += Blc ) {
    for ( j = 0; j < n; j += Blc ) {
      for ( k = 0; k < n; k += Blc ) {
        // B x B mini matrix multiplications
        for ( i1 = i; i1 < (i + Blc); i1 += 2 ) {
          for ( j1 = j; j1 < (j + Blc); j1 += 2 ) {
            register double C00 = C1[(i1 + 0) * n + (j1 + 0)];
            register double C01 = C1[(i1 + 0) * n + (j1 + 1)];
            register double C10 = C1[(i1 + 1) * n + (j1 + 0)];
            register double C11 = C1[(i1 + 1) * n + (j1 + 1)];
            for ( k1 = k; k1 < (k + Blc); k1 += 2 ) {
              register double A00 = A[(i1 + 0) * n + (k1 + 0)];
              register double A01 = A[(i1 + 0) * n + (k1 + 1)];
              register double A10 = A[(i1 + 1) * n + (k1 + 0)];
              register double A11 = A[(i1 + 1) * n + (k1 + 1)];
              register double B00 = B[(k1 + 0) * n + (j1 + 0)];
              register double B01 = B[(k1 + 0) * n + (j1 + 1)];
              register double B10 = B[(k1 + 1) * n + (j1 + 0)];
              register double B11 = B[(k1 + 1) * n + (j1 + 1)];
              C00 += A00 * B00 + A01 * B10;
              C01 += A00 * B01 + A01 * B11;
              C10 += A10 * B00 + A11 * B10;
              C11 += A10 * B01 + A11 * B11;
            }
            C1[(i1 + 0) * n + (j1 + 0)] = C00;
            C1[(i1 + 0) * n + (j1 + 1)] = C01;
            C1[(i1 + 1) * n + (j1 + 0)] = C10;
            C1[(i1 + 1) * n + (j1 + 1)] = C11;
          }
        }
      }
    }
  }
}
////////////////////////////////////   VOID FUNCTION   ////   VOID FUNCTION   ////////////////////////////////
void jik_blocked(double *A, double *B, double *C1, int n, int Blc){
  int i, j, k, i1, j1, k1;
  for ( j = 0; j < n; j += Blc ) {
    for ( i = 0; i < n; i += Blc ) {
      for ( k = 0; k < n; k += Blc ) {
        // B x B mini matrix multiplications
        for ( j1 = j; j1 < (j + Blc); j1 += 2 ) {
          for ( i1 = i; i1 < (i + Blc); i1 += 2 ) {
            register double C00 = C1[(i1 + 0) * n + (j1 + 0)];
            register double C01 = C1[(i1 + 0) * n + (j1 + 1)];
            register double C10 = C1[(i1 + 1) * n + (j1 + 0)];
            register double C11 = C1[(i1 + 1) * n + (j1 + 1)];
            for ( k1 = k; k1 < (k + Blc); k1 += 2 ) {
              register double A00 = A[(i1 + 0) * n + (k1 + 0)];
              register double A01 = A[(i1 + 0) * n + (k1 + 1)];
```

```
                register double A10 = A[(i1 + 1) * n + (k1 + 0)];
                register double A11 = A[(i1 + 1) * n + (k1 + 1)];
                register double B00 = B[(k1 + 0) * n + (j1 + 0)];
                register double B01 = B[(k1 + 0) * n + (j1 + 1)];
                register double B10 = B[(k1 + 1) * n + (j1 + 0)];
                register double B11 = B[(k1 + 1) * n + (j1 + 1)];
                C00 += A00 * B00 + A01 * B10;
                C01 += A00 * B01 + A01 * B11;
                C10 += A10 * B00 + A11 * B10;
                C11 += A10 * B01 + A11 * B11;
              }
              C1[(i1 + 0) * n + (j1 + 0)] = C00;
              C1[(i1 + 0) * n + (j1 + 1)] = C01;
              C1[(i1 + 1) * n + (j1 + 0)] = C10;
              C1[(i1 + 1) * n + (j1 + 1)] = C11;
            }
          }
        }
      }
    }
  }
////////////////////////////////////  VOID FUNCTION  ////  VOID FUNCTION  ////////////////////////////////
void kij_blocked(double *A, double *B, double *C1, int n, int Blc){
  int i, j, k, i1, j1, k1;
  for ( k = 0; k < n; k += Blc ) {
    for ( i = 0; i < n; i += Blc ) {
      for ( j = 0; j < n; j += Blc ) {
        // B x B mini matrix multiplications
        for ( k1 = k; k1 < (k + Blc); k1 += 2 ) {
          for ( i1 = i; i1 < (i + Blc); i1 += 2 ) {
            register double A00 = A[(i1 + 0) * n + (k1 + 0)];
            register double A01 = A[(i1 + 0) * n + (k1 + 1)];
            register double A10 = A[(i1 + 1) * n + (k1 + 0)];
            register double A11 = A[(i1 + 1) * n + (k1 + 1)];
            for ( j1 = j; j1 < (j + Blc); j1 += 2 ) {
              register double B00 = B[(k1 + 0) * n + (j1 + 0)];
              register double B01 = B[(k1 + 0) * n + (j1 + 1)];
              register double B10 = B[(k1 + 1) * n + (j1 + 0)];
              register double B11 = B[(k1 + 1) * n + (j1 + 1)];
              C1[(i1 + 0) * n + (j1 + 0)] += A00 * B00 + A01 * B10;
              C1[(i1 + 0) * n + (j1 + 1)] += A00 * B01 + A01 * B11;
              C1[(i1 + 1) * n + (j1 + 0)] += A10 * B00 + A11 * B10;
              C1[(i1 + 1) * n + (j1 + 1)] += A10 * B01 + A11 * B11;
            }
          }
        }
      }
    }
  }
}
////////////////////////////////////  VOID FUNCTION  ////  VOID FUNCTION  ////////////////////////////////
void ikj_blocked(double *A, double *B, double *C1, int n, int Blc){
  int i, j, k, i1, j1, k1;
  for ( i = 0; i < n; i += Blc ) {
    for ( k = 0; k < n; k += Blc ) {
      for ( j = 0; j < n; j += Blc ) {
        // B x B mini matrix multiplications
```

```
          for ( i1 = i; i1 < (i + Blc); i1 +=2 ) {
            for ( k1 = k; k1 < (k + Blc); k1 += 2 ) {
              register double A00 = A[(i1 + 0) * n + (k1 + 0)];
              register double A01 = A[(i1 + 0) * n + (k1 + 1)];
              register double A10 = A[(i1 + 1) * n + (k1 + 0)];
              register double A11 = A[(i1 + 1) * n + (k1 + 1)];
              for ( j1 = j; j1 < (j + Blc); j1 +=2 ) {
                register double B00 = B[(k1 + 0) * n + (j1 + 0)];
                register double B01 = B[(k1 + 0) * n + (j1 + 1)];
                register double B10 = B[(k1 + 1) * n + (j1 + 0)];
                register double B11 = B[(k1 + 1) * n + (j1 + 1)];
                C1[(i1 + 0) * n + (j1 + 0)] += A00 * B00 + A01 * B10;
                C1[(i1 + 0) * n + (j1 + 1)] += A00 * B01 + A01 * B11;
                C1[(i1 + 1) * n + (j1 + 0)] += A10 * B00 + A11 * B10;
                C1[(i1 + 1) * n + (j1 + 1)] += A10 * B01 + A11 * B11;
              }
            }
          }
        }
      }
    }
  }
}
////////////////////////////////////  VOID FUNCTION  ////  VOID FUNCTION  ////////////////////////////////
void jki_blocked(double *A, double *B, double *C1, int n, int Blc){
  int i, j, k, i1, j1, k1;
  for ( j = 0; j < n; j += Blc ) {
    for ( k = 0; k < n; k += Blc ) {
      for ( i = 0; i < n; i += Blc ) {
        // B x B mini matrix multiplications
        for ( j1 = j; j1 < (j + Blc); j1 += 2 ) {
          for ( k1 = k; k1 < (k + Blc); k1 += 2 ) {
            register double B00 = B[(k1 + 0) * n + (j1 + 0)];
            register double B01 = B[(k1 + 0) * n + (j1 + 1)];
            register double B10 = B[(k1 + 1) * n + (j1 + 0)];
            register double B11 = B[(k1 + 1) * n + (j1 + 1)];
            for ( i1 = i; i1 < (i + Blc); i1 += 2 ) {
              register double A00 = A[(i1 + 0) * n + (k1 + 0)];
              register double A01 = A[(i1 + 0) * n + (k1 + 1)];
              register double A10 = A[(i1 + 1) * n + (k1 + 0)];
              register double A11 = A[(i1 + 1) * n + (k1 + 1)];
              C1[(i1 + 0) * n + (j1 + 0)] += A00 * B00 + A01 * B10;
              C1[(i1 + 0) * n + (j1 + 1)] += A00 * B01 + A01 * B11;
              C1[(i1 + 1) * n + (j1 + 0)] += A10 * B00 + A11 * B10;
              C1[(i1 + 1) * n + (j1 + 1)] += A10 * B01 + A11 * B11;
            }
          }
        }
      }
    }
  }
}
////////////////////////////////////  VOID FUNCTION  ////  VOID FUNCTION  ////////////////////////////////
void kji_blocked(double *A, double *B, double *C1, int n, int Blc){
  int i, j, k, i1, j1, k1;
  for ( k = 0; k < n; k += Blc ) {
    for ( j = 0; j < n; j += Blc ) {
      for ( i = 0; i < n; i += Blc ) {
```

```
        // B x B mini matrix multiplications
        for ( k1 = k; k1 < (k + Blc); k1 += 2 ) {
          for ( j1 = j; j1 < (j + Blc); j1 += 2 ) {
            register double B00 = B[(k1 + 0) * n + (j1 + 0)];
            register double B01 = B[(k1 + 0) * n + (j1 + 1)];
            register double B10 = B[(k1 + 1) * n + (j1 + 0)];
            register double B11 = B[(k1 + 1) * n + (j1 + 1)];
            for ( i1 = i; i1 < (i + Blc); i1 += 2 ) {
              register double A00 = A[(i1 + 0) * n + (k1 + 0)];
              register double A01 = A[(i1 + 0) * n + (k1 + 1)];
              register double A10 = A[(i1 + 1) * n + (k1 + 0)];
              register double A11 = A[(i1 + 1) * n + (k1 + 1)];
              C1[(i1 + 0) * n + (j1 + 0)] += A00 * B00 + A01 * B10;
              C1[(i1 + 0) * n + (j1 + 1)] += A00 * B01 + A01 * B11;
              C1[(i1 + 1) * n + (j1 + 0)] += A10 * B00 + A11 * B10;
              C1[(i1 + 1) * n + (j1 + 1)] += A10 * B01 + A11 * B11;
            }
          }
        }
      }
    }
  }
}
```