Module 7 solutions

Part I: Timing `datasize1()`

```c
/* ex1.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_SIZE 16777216 // 128*1024*1024/8 elements
extern int datasize1(int);
double arr[MAX_SIZE]; // global array of length MAX_SIZE

#define mytimer clock
#define delta_t(a, b) ((double)((b) - (a)) / CLOCKS_PER_SEC)
#define MIN_RTIME 1.0 // minimum run time in sec.

int main(int argc, char *argv[])
{
    clock_t t1, t2;
    double tcpu = 0.0;
    size_t mem_acc;
    size_t reps;

    printf("# Testing function datasize1:\n");
    printf("%8s %8s\n", "Mem (KB)", "GFLOPS");
    for (size_t i = 2048; i <= MAX_SIZE; i *= 2)
    {
        tcpu = 0.0;
        /* incr. number of repetitions until run time > MIN_RTIME */
        for (reps = 4; tcpu < MIN_RTIME; reps *= 2)
        {
            tcpu = 0.0;
            mem_acc = 0;
            t1 = mytimer();
            for (size_t k = 0; k < reps; k++)
                mem_acc += datasize1(i);
            t2 = mytimer();
            tcpu = delta_t(t1, t2);
        }
        // Print memory (KB) and GFLOPS
        printf("%6.0f  %8.2f\n",
               sizeof(double) * i / 1024.0, 1e-9 * mem_acc / tcpu);
    }
    return 0;
}
```

Part II

1. Implement the functions `my_dgemv_v1()` and `my_dgemv_v2()`:

```c
#include "my_dgemv.h"

int my_dgemv_v1(double alpha, double beta, const array2d_t *A,
    const array_t *x,array_t *y)
{
  if (!A||!x||!y) return 1;
  if (A->shape[1] != x->len || A->shape[0] != y->len) return 1;

  size_t m=A->shape[0];
  size_t n=A->shape[1];
  double *px = x->val;
  double *py = y->val;

  for (size_t i=0;i<m;i++) py[i] *= beta;

  if (A->order == RowMajor) {
    for (size_t i=0;i<m;i++) {
      double *pA = A->val + i*n;
      double sum = 0.0;
      for (size_t j=0;j<n;j++) {
        sum += alpha*pA[j]*px[j];      // stride 1
      }
      py[i] += sum;
    }
  }
  else {
    for (size_t i=0;i<m;i++) {
      double *pA = A->val + i;
      double sum = 0.0;
      for (size_t j=0;j<n;j++) {
        sum += alpha*pA[m*j]*px[j];    // stride m
      }
      py[i] += sum;
    }
  }
  return 0;
}
```

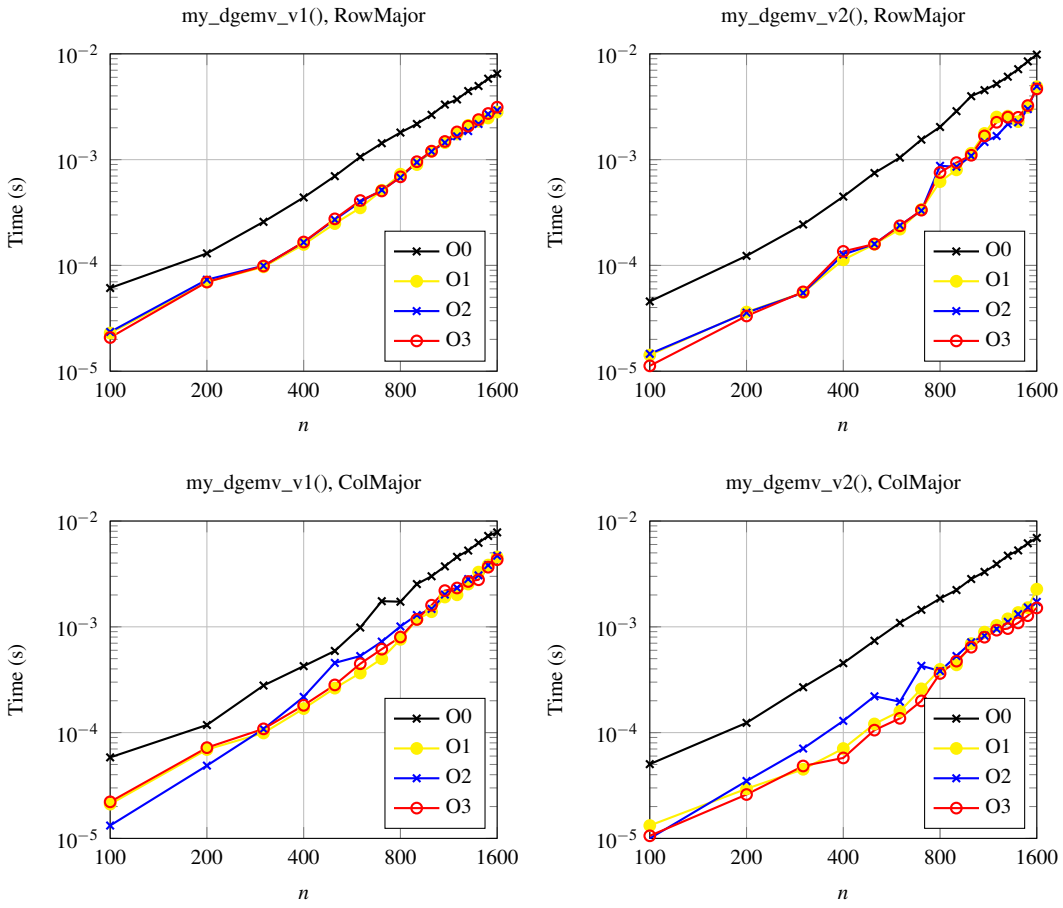
```c
#include "my_dgemv.h"

int my_dgemv_v2(double alpha, double beta, const array2d_t *A,
    const array_t *x,array_t *y)
{
```

```c
  if (!A||!x||!y) return 1;
  if (A->shape[1] != x->len || A->shape[0] != y->len) return 1;

  size_t m=A->shape[0];
  size_t n=A->shape[1];
  double *px = x->val;
  double *py = y->val;

  for (size_t i=0;i<m;i++) py[i] *= beta;

  if (A->order == RowMajor) {
    for (size_t j=0;j<n;j++) {
      double *pA = A->val + j;
      for (size_t i=0;i<m;i++) {
        py[i] += alpha*pA[i*n]*px[j];      // stride n
      }
    }
  }
  else {
    for (size_t j=0;j<n;j++) {
      double *pA = A->val + j*m;
      for (size_t i=0;i<m;i++) {
        py[i] += alpha*pA[i]*px[j];        // stride 1
      }
    }
  }
  return 0;
}
```

2. See solution to exercise 4 below.
3. See solution to exercise 4 below.
4. Repeat the two timing experiments with compiler optimizations:

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "msptools.h"
#include "testcase.h"

#define NREPEAT 200
#define mytimer clock
#define delta_t(a, b) ((double)((b) - (a)) / CLOCKS_PER_SEC)

#ifndef ORDER
#define ORDER RowMajor
#endif
```

```c
int my_dgemv_v1(double alpha, double beta, const array2d_t *A,
    const array_t *x,array_t *y);
int my_dgemv_v2(double alpha, double beta, const array2d_t *A,
    const array_t *x,array_t *y);

int main(int argc, char const *argv[])
{

    size_t i, m, n, N = NREPEAT;
    enum storage_order order = ORDER;
    double tcpu1, tcpu2;
    clock_t t1, t2;

    for (m = 100; m <= 1600; m += 100)
    {
        n = m;

        // Allocate test case
        array2d_t *a;
        array_t *x, *y;
        if (testcase(n, order, &a, &x, &y) != 0)
            return EXIT_FAILURE;

        /* CPU time for my_dgemv_v1 */
        t1 = mytimer();
        for (i = 0; i < N; i++)
            my_dgemv_v1(1.0, 0.0, a, x, y);
        t2 = mytimer();
        tcpu1 = delta_t(t1, t2) / N;

        /* CPU time for my_dgemv_v2 */
        t1 = mytimer();
        for (i = 0; i < N; i++)
            my_dgemv_v2(1.0, 0.0, a, x, y);
        t2 = mytimer();
        tcpu2 = delta_t(t1, t2) / N;

        /* Print n and results */
        printf("%4zu %8.3e %8.3e\n", n, tcpu1, tcpu2);

        // Deallocate test case
        array2d_dealloc(a);
        array_dealloc(x);
        array_dealloc(y);
    }

    return EXIT_SUCCESS;
}
```

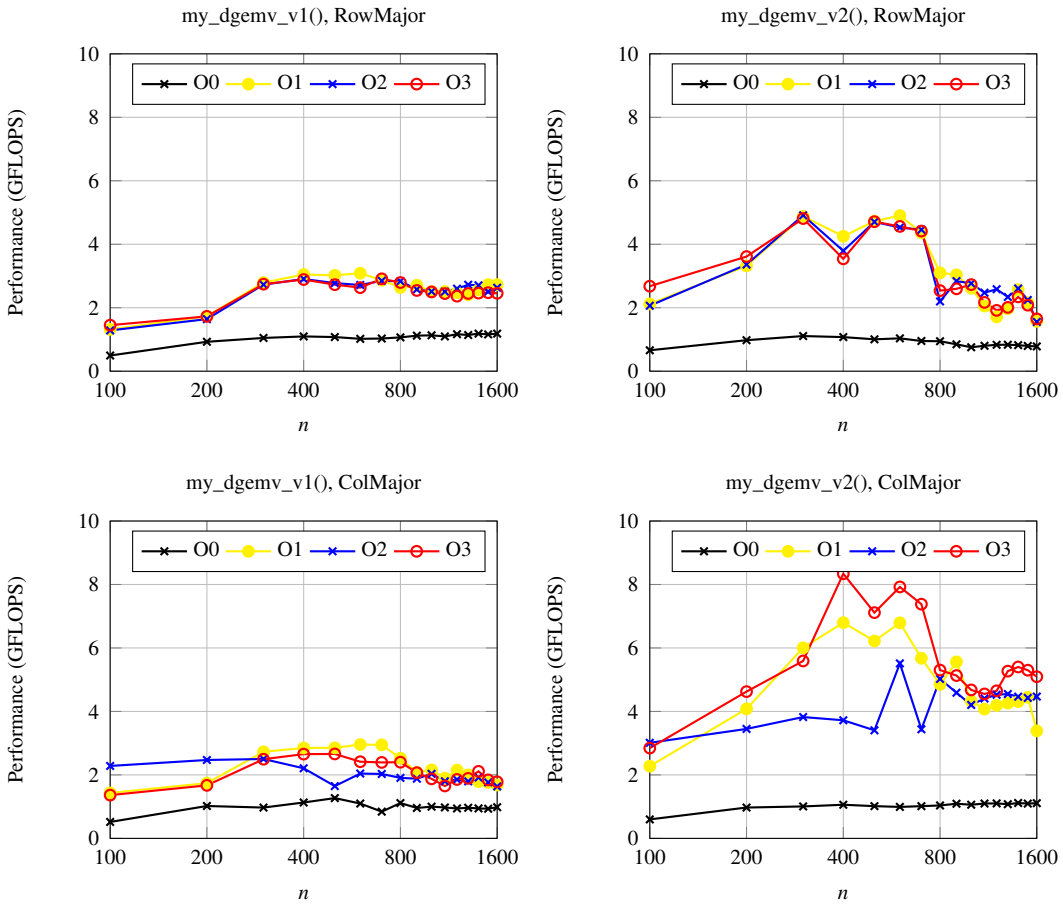


The results show that all versions benefit quite a bit from compiler optimization (for all n). Note that for large n, `my_dgemv_v1` is faster than `my_dgemv_v2` when the input array is in row-major storage order whereas `my_dgemv_v2` is faster than `my_dgemv_v1` when the input array is in column-major storage order. Indeed, the spacial locality is much better when the elements are a in accordance with the storage order.

Note that the CPU times may differ (significantly) on other systems.

5. The above implementation of `my_dgemv_v1` requires $3mn + 2m$ FLOPs whereas `my_dgemv_v2` requires $3mn + m$ FLOPs for a matrix A of size $m \times n$. Assuming that the time is T seconds and $m = n$, we can compute the performance in GFLOPS as

$$\frac{3n^2 + 2n}{T} \cdot 10^{-9}, \quad \text{or} \quad \frac{3n^2 + n}{T} \cdot 10^{-9}.$$


my_dgemv_v1(), RowMajor
my_dgemv_v2(), RowMajor
my_dgemv_v1(), ColMajor
my_dgemv_v2(), ColMajor
Performance (GFLOPS)
n
O0
O1
O2
O3
0
2
4
6
8
10
100
200
400
800
1600