Week 7 solutions

Exercises

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) (1.0e3 * ((b) - (a)) / CLOCKS_PER_SEC)
#define MIN_RTIME 2000  // run iterations for at least MIN_RTIME msecs

int main(int argc, char *argv[]) {
    clock_t t1, t2;
    double tcpu;
    int mem_acc;
    int iter;

    printf("# Testing function datasize1:\n");
    printf("%10s %9s\n","Mem (kB)","Mflop/s");
    for(size_t i = 2048; i <= MAX_SIZE; i *= 2) {
        tcpu = 0.0; iter = 0;
        t1 = mytimer();
        do {
            mem_acc = datasize1(i);
            t2 = mytimer();
            tcpu = delta_t(t1, t2);
            iter++;
        } while (tcpu < MIN_RTIME);

        // Print memory (kB) and Mflop/s
        printf("%10.0f %9.2e\n", 8.0*i/1024, (1e-3*iter)*mem_acc/tcpu);
    }

    return(0);
}
```

```c
/* datasize1.c */
extern double arr[];
```

```
int datasize1(int elem) {

    for (int i=0; i<elem; i++)
        arr[i] *= 3;

    return(elem);
}
```

Part II

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

```
void my_dgemv_v1(
  int m,          /* number of rows                    */
  int n,          /* number of columns                 */
  double alpha,   /* scalar                            */
  double ** A,    /* two-dim. array A of size m-by-n   */
  double * x,     /* one-dim. array x of length n      */
  double beta,    /* scalar                            */
  double * y      /* one-dim. array x of length m      */
) {
  int i,j;
  for (i=0;i<m;i++) {
    y[i] *= beta;
    for (j=0;j<n;j++) {
      y[i] += alpha*A[i][j]*x[j];
    }
  }
  return;
}
```

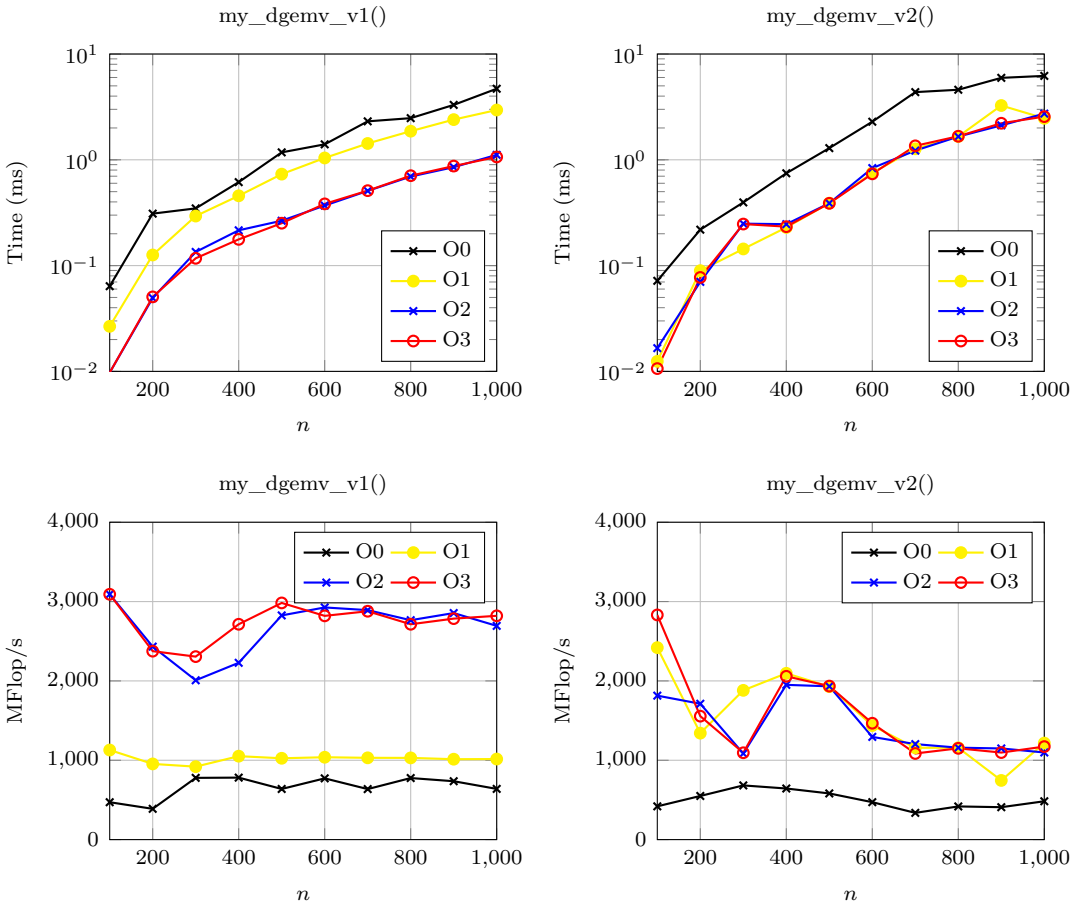
```
void my_dgemv_v2(
  int m,          /* number of rows                    */
  int n,          /* number of columns                 */
  double alpha,   /* scalar                            */
  double ** A,    /* two-dim. array A of size m-by-n   */
  double * x,     /* one-dim. array x of length n      */
  double beta,    /* scalar                            */
  double * y      /* one-dim. array x of length m      */
) {
  int i,j;
  for (i=0;i<m;i++)
    y[i] *= beta;
  for (j=0;j<n;j++) {
    for (i=0;i<m;i++) {
      y[i] += alpha*A[i][j]*x[j];
```

```
    }
  }
  return;
}
```

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

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "my_dgemv_v1.h"
#include "my_dgemv_v2.h"
#include "array_2d.h"

#define NREPEAT 200
#define mytimer clock
#define delta_t(a,b) (1.0e3 * ((b) - (a)) / CLOCKS_PER_SEC)

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

  int i, m, n, N = NREPEAT;
  double *x, *y, **A, tcpu1, tcpu2;
  clock_t t1, t2;

  printf("%4s %8s %8s\n", "n", "v1", "v2");
  for (m = 100; m <= 1000; m += 100) {
    n = m;

    /* Allocate memory */
    A = malloc_2d(m, n);
    x = malloc(n*sizeof(*x));
    y = malloc(m*sizeof(*y));
    if (A == NULL || x == NULL || y == NULL) {
      fprintf(stderr, "Memory allocation error..\n");
      free_2d(A); free(x); free(y);
      exit(EXIT_FAILURE);
    }

    /* CPU time for my_dgemv_v1 */
    t1 = mytimer();
    for (i = 0; i < N; i++)
      my_dgemv_v1(m, n, 1.0, A, x, 0.0, 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(m, n, 1.0, A, x, 0.0, y);
    t2 = mytimer();
    tcpu2 = delta_t(t1,t2)/N;

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

    /* Free memory */
    free_2d(A);
    free(x);
    free(y);
  }

  return EXIT_SUCCESS;
}
```



The results show that both versions benefit quite a bit from compiler optimization (for all n). Moreover, `my_dgemv_v1` is significantly faster than `my_dgemv_v2` when n is large. Indeed, the spacial locality is much better in the first variant of the method since it accesses the elements of A row-by-row in accordance with the row-major storage.

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

5. The above implementations of `my_dgemv_v1` and `my_dgemv_v2` both require $3mn+m$ floating-point operations 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 MFlop/s as

$$\frac{3n^2 + n}{10^6 \cdot T}.$$

Finally, note that nested loops in `my_dgemv_v1` can also be implemented as follows:

```
for (i=0;i<m;i++) {
  double dotx = 0.0;
  for (j=0;j<n;j++) {
    dotx += A[i][j]*x[j];
  }
  y[i] = alpha*dotx + beta*y[i];
}
```

This implementation requires $2mn + 3m$ Flops instead of $3mn + m$ Flops. Similarly, `my_dgemv_v2` can also be implemented with a lower Flop count than that of the implementation provided above.