Τμημα Μηχανικων Η/Υ & Πληροφορικης,
Πανεπιστημιο Πατρων

Αναφορα Εξαμηνιαιας Εργασιας

Χειμερινο Εξαμηνο

Λογισμικό & Προγραμματισμός Συστημάτων Υψηλής Επίδοσης

Κωνσταντίνος Μπουραντάς 6145
Ομάδα 18
bourantas@ceid.upatras.gr

24 Ιουλίου 2021

Περιεχόμενα

Κατάλογος Σχημάτων

Ευρετήριο Κώδικα

1 Εισαγωγή

Στην συγκεκριμένη εργασία καλούμαστε να παραλληλοποίησουμε τον υπολογισμό του γινομένου του ανάστροφου ενός μητρώου με το αρχικό μητρώο

$$A^T * A$$

σε κάρτα γραφικών της NVIDIA με την χρήση του υπολογιστικού μοντέλου της CUDA. Στο πρώτο ερώτημα υλοποιούμε την συγκεκριμένη πράξη με την χρήση της βιβλιοθήκης cuBLAS. Έπειτα στο δεύτερο ερώτημα υλοποιούμε τον πιο απλό τρόπο πραγματοποίησης της πράξης στην κάρτα γραφικών δεσμεύοντας καθολική μνήμη για το μητρώο και κάνοντας τις πράξεις που απαιτούνται. Τέλος στο τρίτο ερώτημα , προσπαθούμε να βελτιστοποιήσουμε όσο καλύτερα γίνεται το δεύτερο ερώτημα ώστε να πάρουμε καλύτερο χρόνο εκτέλεσης.

Για τους σκοπούς της εργασίας δημιουργήθηκε makefile για την ταχύτερη μεταγλώττιση και εκτέλεση των προγραμμάτων. Για μεταγλώττιση όλων των ερωτημάτων χρησιμοποιούμε την εντολή **make compile**. Για την εκτέλεση του κάθε προγράμματος χρησιμοποιούμε την εντολή **make run** ως εξής: **make run PROG=erotima_1 ROWS=10 COLS=10 ITER=1**.

2 Ερωτήματα

2.1 Ερώτημα 1

Στο πρώτο ερώτημα ξεκινάμε δημιουργώντας το μητρώο A και το μητρώο στο οποίο θα αποθηκεύσουμε το αποτέλεσμα με βάση τις παραμέτρους που δίνει ο χρήστης στο πρόγραμμα και αναθέτοντας σε αυτό τυχαίες τιμές και το αποθηκεύουμε στο host. Επίσης δημιουργούμε τα μητρώα τα οποία θα χρησιμοποιήσουμε για την εκτέλεση πράξεων στο device(κάρτα γραφικών) , δεσμεύοντας κατάλληλα μνήμη. Επίσης χρησιμοποιούμε την συνάρτηση της CUDA , cudaEvent_t για να μπορέσουμε να μετρήσουμε σωστά τους χρόνους εκτέλεσης.

Απόσπασμα Κώδικα 1: Δημιουργία και αρχικοποίηση μητρώου A και δέσμευση μνήμης

```
1  if (argc < 4)
2       {
3               cout << "Usage: " << argv[0] << "[Rows]-[Cols]-[
                    ↪ Iterations]" << endl;
4               return 1;
5       }
6
7       unsigned int rows_A, cols_A, rows_C, cols_C, rows_A_T,
            ↪ cols_A_T;
8
9       rows_A = cols_A_T = atoi(argv[1]);
10      cols_A = rows_A_T = atoi(argv[2]);
```

```
11        rows_C = cols_C = cols_A;
12        int iterations = atoi(argv[3]);
13
14        cudaEvent_t start, stop;
15
16        // Allocate host_A and host_C on host
17        double *host_A = (double*) malloc(rows_A *cols_A* sizeof(
              ↪ double));
18        double *host_C = (double*) malloc(rows_C *cols_C* sizeof(
              ↪ double));
19
20        //fill array host_A with values in column major order
21
22        double random_data ;
23          for (int i = 0; i < rows_A * cols_A; i++) {
24               random_data = rand() % 1024;
25            host_A[i] = random_data;
26          }
27
28        // print_matrix(host_A, rows_A, cols_A);
29
30        // Allocate array device_A and deive_C on GPU memory
31        double *device_A;
32        cudaMalloc(&device_A, rows_A *cols_A* sizeof(double));
33        cudaCheckError();
34
35        double *device_C;
36        cudaMalloc(&device_C, rows_C *cols_C* sizeof(double));
37        cudaCheckError();
```

Έπειτα εκτελούμε την συνάρτηση cublasDgemm την οποία με την παρέχει η βιβλιοθήκη cuBLAS και την χρησιμοποιούμε για πράξεις πολλαπλασιασμού μητρώων. Για την εκτέλεση του πολλαπλασιασμού που θέλουμε να πραγματοποιήσουμε δηλώνουμε στην συνάρτηση ότι θέλουμε το ανάστροφο του μητρώου A με την μεταβλητή CUBLAS_OP_T. Επίσης δίνουμε ως είσοδο μόνο το μητρώο A και βάζουμε αντίστροφα ως είσοδο τον αριθμό των γραμμών και των στηλών του μητρώου A έτσι ώστε να πάρουμε το επιθυμητό αποτέλεσμα. Οι υπόλοιπες παράμετροι συμπληρώθηκαν με βάση το documantation της CUDA.

Απόσπασμα Κώδικα 2: Εκτέλεση του πολ/σμου με χρήση της βιβλ. cuBLAS

```
1  for (int i = 0; i < iterations; i++)
2        {
3                tempTime = 0;
4
5                double alpha = 1;
6                double beta = 0;
```

```
7
8               // Create cublas handle
9               cublasHandle_t handle;
10              cudaCheckError();
11              cublasCreate(&handle);
12              cudaCheckError();
13              cudaEventRecord(start);
14              cudaCheckError();
15
16              cublasDgemm(handle, CUBLAS_OP_T,
17                     CUBLAS_OP_N, cols_A,
18                     cols_A, rows_A, &alpha, device_A,
19                     rows_A, device_A,
20                     rows_A, &beta,
21                     device_C, cols_A);
22              cudaCheckError();
23
24              cudaEventRecord(stop);
25              cudaCheckError();
26
27              // Destroy the handle
28              cublasDestroy(handle);
29              cudaCheckError();
30
31              // Copy the resulted matrix device_C to host
32              cudaMemcpy(host_C, device_C, rows_C *cols_C* sizeof
                    ↪ (double), cudaMemcpyDeviceToHost);
33              cudaCheckError();
34
35              cudaEventSynchronize(stop);
36              cudaCheckError();
37
38              // Compute the elapsed time between the two events
                    ↪ start and stop
39              cudaEventElapsedTime(&tempTime, start, stop);
40              cudaCheckError();
41
42              cout <<"Elapsed␣Time:␣" << tempTime << "␣seconds"
                    ↪ << endl;
43              time_elapsed += tempTime;
44      }
```

2.2 Ερώτημα 2

Στο δεύτερο ερώτημα υλοποιούμε τον πιο απλό τρόπο πραγματοποίησης της πράξης στην κάρτα γραφικών δεσμεύοντας καθολική μνήμη για το μητρώο και κάνοντας τις πράξεις που απαιτούνται στην κάρτα γραφικών. Για την εκτέλεση του πυρήνα του πολλαπλασιάσουμε χρησιμοποιήθηκαν οι παρακάτω παράμετροι για τις μεταβλητές dimGrid & dimBlock ως εξής προκείμενου να εξασφαλίσουμε ότι θα έχουμε αρκετά μεγάλα μεγέθη ώστε να επεξεργαστούμε ολόκληρο το μητρώο μέσα στον πυρήνα:

Απόσπασμα Κώδικα 3: Αρχικοποίηση παραμέτρων υπ. πυρήνα

```
1  unsigned int grid_rows = (cols_A + BLOCK_SIZE - 1) / BLOCK_SIZE
       ↪ ;
2    unsigned int grid_cols = (rows_A + BLOCK_SIZE - 1) /
       ↪ BLOCK_SIZE;
3
4    dim3 dimGrid(grid_cols, grid_rows);
5    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
6
7    // Create cublas handle
8    cublasHandle_t handle;
9    cudaCheckError();
10   cublasCreate(&handle);
11   cudaCheckError();
12   cudaEventRecord(start);
13   cudaCheckError();
14
15   // run gpu kernel
16   multiplicationKernel<<<dimGrid, dimBlock>>>(device_A, device_C
       ↪ , rows_A,
17                                               cols_A);
```

Στην συνέχεια ακολουθεί ο υπολογιστικός πυρήνας που είναι υπεύθυνος για τον πολλαπλασιασμό που θέλουμε να υλοποιήσουμε. Χρησιμοποιούμε τις μεταβλητές row & col προκειμένου το κάθε νήμα να επεξεργαστεί το κατάλληλο στοιχείο προκειμένου να γίνει σωστά ο πολλαπλασιασμός. Έπειτα μέσα στον βρόγχο πραγματοποιούμε τον πολλαπλασιασμό στοιχείων όπου για να κάνουμε τον ζητούμε πολλαπλασιασμό , πολλαπλασιάζουμε μια γραμμή την φορά με όλες τις γραμμές. Τέλος αποθηκεύουμε σε κάθε στοιχείου του παραγόμενου μητρώου την τιμή που προέκυψε.

Απόσπασμα Κώδικα 4: Υπολογιστικός πυρήνας - ερώτημα 2

```
1  __global__ void multiplicationKernel(double const *const
       ↪ input_matrix,
2                                        double *const output_matrix,
3                                        const int rows_A, const int
                                             ↪ cols_A) {
```

```
4
5   const int row = blockIdx.y * blockDim.y + threadIdx.y;
6   const int col = blockIdx.x * blockDim.x + threadIdx.x;
7
8   double tempSum = 0.0;
9
10  if (row < cols_A && col < cols_A) {
11
12    for (int i = 0; i < rows_A; ++i) {
13
14      tempSum +=
15          input_matrix[i * cols_A + row] * input_matrix[i * cols_A
                ↪  + col];
16
17    }
18    output_matrix[row * cols_A + col] = tempSum;
19  }
20 }
```

2.3 Ερώτημα 3

Στο τελευταίο ερώτημα , καλούμαστε να βελτιστοποιήσουμε τη προηγούμενη υλοποίηση. Για να καταφέρουμε να πάρουμε χαμηλότερους χρόνους εκτέλεσης , έγινε χρήση της κοινής μνήμης καθώς και της τεχνικής του tiling. Το μόνο που άλλαξε σε σχέση με το προηγούμενο ερώτημα είναι ο υπολογιστικός πυρήνας. Αρχικά ξεκινάμε αντιγράφοντας κάθε φορά ένα tile στην κοινή μνήμη προκειμένου να επεξεργαστή από τα threads. Έπειτα πραγματοποιούμε τους πολλαπλασιασμούς και αποθηκεύουμε την τιμή σε έναν register στον οποίο συγκεντρώνουμε το αποτέλεσμα και στην συνέχεια αποθηκεύουμε το αποτέλεσμα στην κατάλληλη θέση του μητρώου που θα γυρίσουμε στο host. Επίσης χρησιμοποιούμε τις εντολές pragma unroll προκειμένου να ξεδιπλώσουμε τους βρόγχους κάτι που θα μας δώσει λίγο καλύτερους χρόνους. Χρησιμοποιούμε τις εντολές syncthreads προκειμένου να εξασφαλίσουμε ότι όλα τα νήματα θα έχουν ολοκληρώσει τις πράξεις που κάνουν για να μπορέσουν να συνεχίσουν με την εκτέλεση. Ακολουθεί ο κώδικας του υπολογιστικού πυρήνα:

Απόσπασμα Κώδικα 5: Υπολογιστικός πυρήνας - ερώτημα 2

```
1  __global__ void matrixMultKernel(double *A, double *
       ↪ resultedMatrix, int rows_A, int cols_A, int rows_C,
2                      int cols_C) {
3
4  __shared__ double A_T_shared[BLOCK_SIZE][BLOCK_SIZE+1];
5  __shared__ double A_shared[BLOCK_SIZE][BLOCK_SIZE+1];
6
7  int by = blockIdx.y;
```

```
8   int ty = threadIdx.y;
9
10  int bx = blockIdx.x;
11  int tx = threadIdx.x;
12
13  int width = cols_A;
14  int height = rows_A;
15
16  int row = by * BLOCK_SIZE + ty;
17  int col = bx * BLOCK_SIZE + tx;
18
19  double temp_sum = 0;
20  int helper;
21
22  #pragma unroll
23  for (int i = 0; i < (BLOCK_SIZE + cols_A - 1) / BLOCK_SIZE+1;
        ↪ ++i) {
24
25    helper = i*BLOCK_SIZE;
26
27        A_shared[tx][ty] = A[(helper + ty) * width + col];
28
29        A_T_shared[tx][ty] = A[(helper + tx) * width + row];
30
31    __syncthreads();
32
33    #pragma unroll
34    for (int j = 0; j < BLOCK_SIZE; ++j) {
35
36      temp_sum += A_T_shared[j][tx] * A_shared[ty][j];
37    }
38
39    __syncthreads();
40
41  }
42
43  if (row < rows_C && col < cols_C)
44    resultedMat
```

3 Πειραματική Αξιολόγηση

Παρακάτω παρουσιάζονται τα αποτελέσματα των μετρήσεων για τα τρία ερωτήματα σε δευτερόλεπτα. Τα προγράμματα εκτελέστηκαν για τετραγωνικά μητρώα μεγέθους 1000, 2000, 3000 , 4000, 5000, 6000, 7000, 8000, 9000 και 10000 γραμμών και στηλών. Για την κάθε περίπτωση , τα προγράμματα εκτελέστηκαν 5 φορές και ως αποτέλεσμα θεωρούμε τον μέσο όρο των 5 αυτών μετρήσεων. Η εκτέλεση των προγραμμάτων έγινε στο μηχάνημα του εργαστηρίου scgroup17.ceid.upatras.gr. Όπως μπορούμε να παρατηρήσουμε από τα αποτελέσματα των μετρήσεων, η πιο γρήγορη υλοποίηση από τα 3 ερώτημα ήταν η υλοποίηση του 1ου ερωτήματος όπου χρησιμοποιήθηκε η βιβλιοθήκη cuBLAS όπως ήταν αναμενόμενο. Έπειτα ακολουθεί η βελτιστοποιημένη υλοποίηση του πολλαπλασιασμού μητρώων με την χρήση tiling και shared memory. Τέλος η πιο αργή υλοποίηση ήταν η naive παραλληλοποίηση του ερωτήματος 2. Και στις τρεις περιπτώσεις παρατηρούμε ότι ο χρόνος εκτέλεσης αυξάνεται ανάλογα με το μέγεθος του μητρώου , επομένως μπορούμε να πούμε ότι παρατηρούμε μια εκθετική συμπεριφορά.

Πίνακας 1: Στοιχεία για τα πειράματα

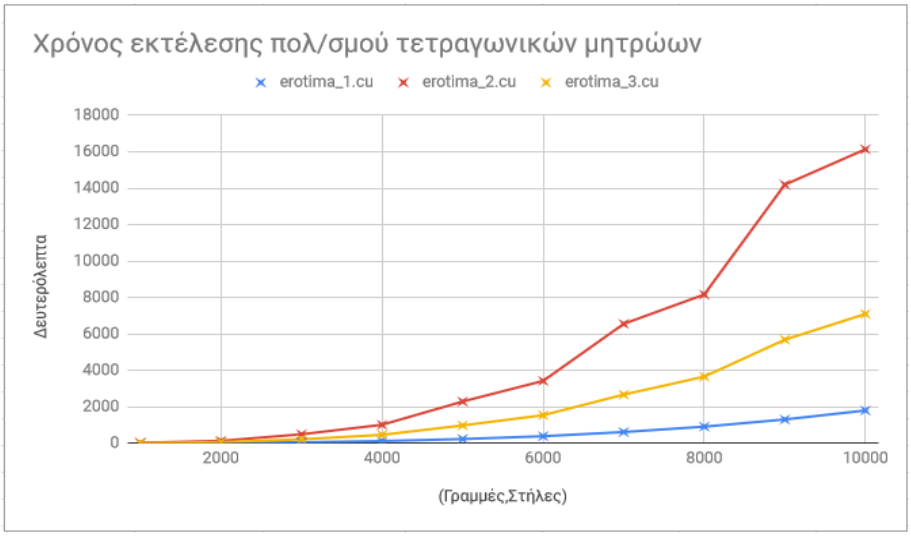
Χαρακτηριστικό	Ενδεικτική Απάντηση
Device name	Tesla K40c
Memory Clock Rate (KHz)	3004000
Memory Bus Width (bits)	384
Peak Memory Bandwidth (GB/s)	288.384000

erotima_2.cu	Rows=1000, Columns=1000	Rows=2000, Columns=2000	Rows=3000, Columns=3000	Rows=4000, Columns=4000	Rows=5000, Columns=5000	Rows=6000, Columns=6000	Rows=7000, Columns=7000	Rows=8000, Columns=8000	Rows=9000, Columns=9000	Rows=10000, Columns=10000
Iteration 1:	17.9496	125.682	486.117	1007.77	2283.87	3420.06	6533.14	8162.82	14052.8	16127.2
Iteration 2:	17.8647	125.6	485.754	1006.73	2282.53	3417.57	6568.68	8155.62	14195.7	16126.2
Iteration 3:	17.814	125.578	485.718	1006.04	2282.68	3417.48	6538.02	8157.89	14234.1	16136.2
Iteration 4:	17.8327	125.614	485.717	1006	2282.87	3416.94	6551.25	8162.35	14212.9	16137.3
Iteration 5:	17.8258	125.53	485.601	1005.49	2283.72	3416.38	6555.74	8161.77	14264.7	16149.7
Μέσος Όρος:	17.85736	125.6008	485.7814	1006.406	2283.134	3417.686	6549.366	8160.09	14192.04	16135.32

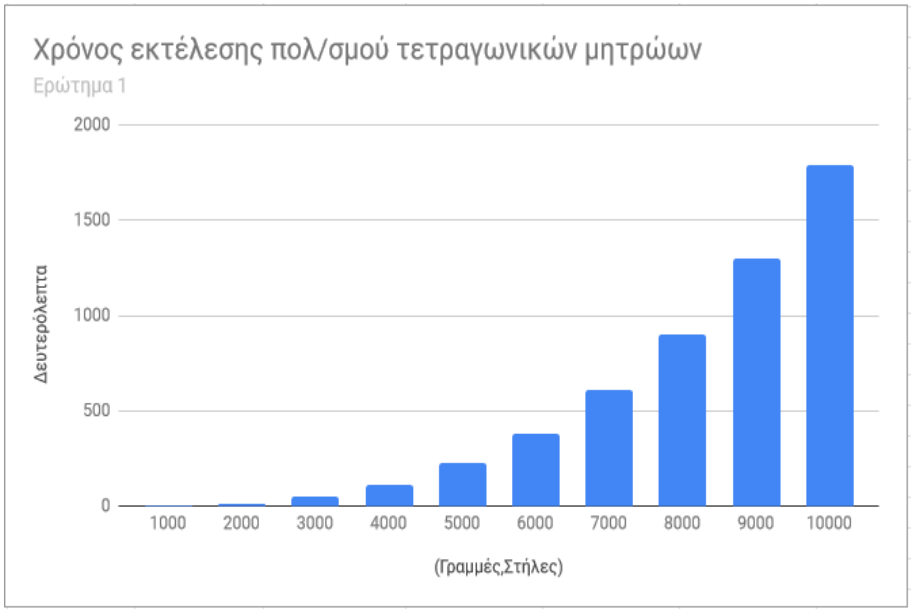
Σχήμα 1: Μετρήσεις χρόνου εκτέλεσης πολλαπλασιασμού τετραγωνικών μητρώων - ερώτημα 2.

erotima_3.cu	Rows=1000, Columns=1000	Rows=2000, Columns=2000	Rows=3000, Columns=3000	Rows=4000, Columns=4000	Rows=5000, Columns=5000	Rows=6000, Columns=6000	Rows=7000, Columns=7000	Rows=8000, Columns=8000	Rows=9000, Columns=9000	Rows=10000, Columns=10000
Iteration 1:	8.00813	57.0991	210.386	454.847	971.046	1533.47	2664.51	3630.96	5670.55	7091.08
Iteration 2:	7.924	57.0199	210.308	454.755	970.779	1533.47	2664.29	3630.56	5669	7091.52
Iteration 3:	7.9471	57.0475	210.223	454.72	970.854	1533.37	2664.24	3731.64	5710.39	7091.81
Iteration 4:	7.9183	57.0169	210.255	454.709	970.742	1533.33	2664.51	3630.96	5670.63	7092.55
Iteration 5:	7.92538	56.9919	210.262	454.74	970.689	1533.66	2664.73	3631.81	5670.89	7093.85
Μέσος Όρος:	7.944582	57.03506	210.2868	454.7542	970.822	1533.46	2664.456	3651.186	5678.292	7092.162

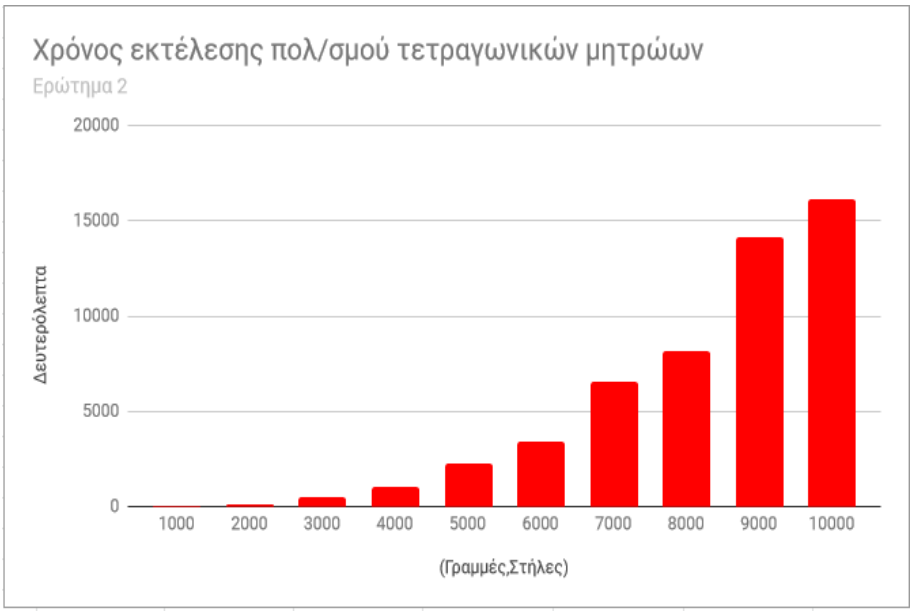
Σχήμα 2: Μετρήσεις χρόνου εκτέλεσης πολλαπλασιασμού τετραγωνικών μητρώων - ερώτημα 3.



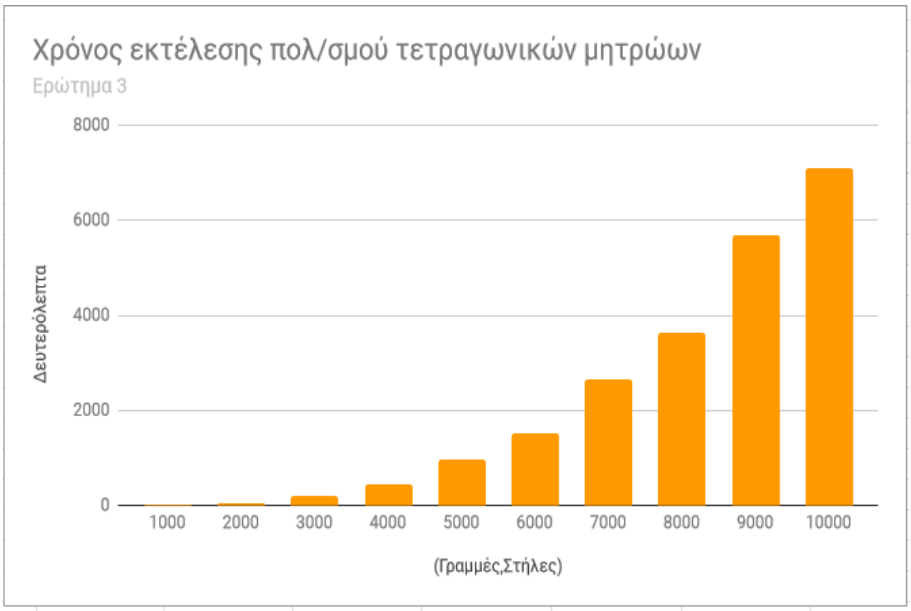
Σχήμα 3: Συγκεντρωτικό γράφημα μετρήσεων χρόνου εκτέλεσης πολ/σμου μητρώων.



Σχήμα 4: Γράφημα μετρήσεων χρόνου εκτέλεσης πολ/σμου μητρώων - ερώτημα 1.



Σχήμα 5: Γράφημα μετρήσεων χρόνου εκτέλεσης πολ/σμου μητρώων - ερώτημα 2.



Σχήμα 6: Γράφημα μετρήσεων χρόνου εκτέλεσης πολ/σμου μητρώων - ερώτημα 3.