Δημοκρίτειο Πανεπιστήμιο Θράκης
Τμήμα: Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Εργαστήριο: Εργαστήριο Αρχιτεκτονικής Υπολογιστών και Συστημάτων Υψηλών Επιδόσεων
Μάθημα: Τεχνολογία Παράλληλης Επεξεργασίας
Ακαδημαϊκό έτος: 2023

Θέμα: Επίλυση συστήματος γραμμικών εξισώσεων με τη μέθοδο απαλοιφής Gauss με χρήση κάρτας γραφικών και CUDA

Ομάδα 1
Αντωναράκος Δημήτριος
Μωραΐτη Παναγιώτα
Σούλη Ευθυμία

Εισαγωγή:

Στην εργασία αυτή δημιουργήσαμε ένα πρόγραμμα γραμμένο σε CUDA που εκτελεί την μέθοδο απαλοιφής Gauss. Ως βάση του κώδικα χρησιμοποιήσαμε το πρόγραμμα σε OpenMP της εργασίας 1.

Αρχικός αλγόριθμος:

C: > Users > Dimitris > Downloads > sol1.cu

```
188  main(int argc, char *argv[])
189  {
190      // Dimensions of the matrix
191      int n;
192      printf("Enter number of unknowns: ");
193      scanf("%d", &n);
194
195      int size_1D = (n+1) * sizeof(double);
196      int size_2D = (n+1) * (n+1) * sizeof(double);
197      double *x;
198      float *a;
199      double *x_temp;
200
201      // allocate memory on the CPU
202      x = (double*)malloc(size_1D);
203      a = (float*)malloc(size_2D);
204      x_temp = (double*)malloc(size_1D);
205
206      // initialize the matrices
207      for (int i=0; i<n; i++)
208      {
209          for (int j=0; j<=n; j++)
210          {
211              //a[i*n + j] = rand();
212              printf("a[%d][%d] = ", i, j);
213              scanf("%f", &a[i*(n+1)+j]);
214          }
215      }
216
217      GaussElimination(a, x, n,x_temp);
218
```

C: > Users > Dimitris > Downloads > sol1.cu

```
209             for (int j=0; j<=n; j++)
210             {
211                 //a[i*n + j] = rand();
212                 printf("a[%d][%d] = ", i, j);
213                 scanf("%f", &a[i*(n+1)+j]);
214             }
215         }
216 
217         GaussElimination(a, x, n,x_temp);
218 
219         // free the memory allocated on the CPU
220         free(a);
221         free(x);
222         free(x_temp);
223 
224         return 0;
225     }
226 
```

Παραπάνω φαίνεται η main συνάρτηση του κώδικά μας. Αρχικά για λόγους απλότητας, επιλέξαμε να χρησιμοποιήσουμε μονοδιάστατους πίνακες για την υλοποίηση του αλγορίθμου. Ο αλγόριθμος υλοποιείται στην συνάρτηση GaussElimination, η οποία έχει ορίσματα τον αριθμό των εξισώσεων n, τον επαυξημένο πίνακα a με μέγεθος (n+1)*(n+1), τον πίνακα x στον οποίο θα αποθηκευτούν όλες οι λύσεις και τον πίνακα x_temp που θα χρησιμοποιηθεί στην αντικατάσταση προς τα πίσω.

GaussElimination:

Αρχικά ορίζουμε τις μεταβλητές ad, xd, x_tempd και x_i. Οι ad, xd και x_tempd είναι οι αντίστοιχες των a, x και x_temp ενώ η x_i θα εξηγηθεί στο κομμάτι της αντικατάστασης προς τα πίσω. Επίσης ορίζουμε τις μεταβλητές blocksize και gridsize, οι οποίες θα έχουν μια σταθερή τιμή. Επιλέξαμε κάθε block να εμπεριέχει 1024 threads και κάθε grid 4096 blocks. Άρα, **κάθε φορά** χρησιμοποιούμε 1024*4096 = 4194304 threads.

C: > Users > Dimitris > Downloads > sol1.cu

```
30  }
31
32  void GaussElimination(float *a, double *x, int n, double *x_temp)
33  {
34      int size_1D = (n+1) * sizeof(double);
35      int size_2D = (n+1) * (n+1) * sizeof(double);
36      double *xd, *x_tempd, x_i;
37      float *ad;
38      float TotalTime, H2DTime1, KernelTime, D2HTime1, H2DTime2, D2HTime2, H2DTime3;
39      float KernelTime1 = 0;
40      float D2HTime2all = 0;
41      float H2DTime3all = 0;
42      float KernelTime2 = 0;
43
44      int blockSize, gridSize;
45      // Number of threads in each thread block
46      blockSize = 1024;
47      // Number of thread blocks in grid
48      gridSize = (int)ceil((float)N/blockSize);
```

Στη συνέχεια δημιουργούμε events που θα μας δώσουν την δυνατότητα να υπολογίζουμε τους χρόνους αντιγραφής μνήμης από και προς τον επεξεργαστή, τον χρόνο εκτέλεσης του kernel, τον χρόνο εκτέλεσης του σειριακού κώδικα και τον συνολικό χρόνο εκτέλεσης όλου του προγράμματος. Επίσης δεσμεύουμε μνήμη στη κάρτα γραφικών για τους πίνακες ad, xd και x_tempd και αντιγράφουμε τον a στον ad με χρήση της συνάρτησης cudaMemcpy μετρώντας παράλληλα το χρόνο εκτέλεσης της αντιγραφής.

sol1.cu

C: > Users > Dimitris > Downloads > sol1.cu

```
46      blockSize = 1024;
47      // Number of thread blocks in grid
48      gridSize = (int)ceil((float)N/blockSize);
49
50      // capture start time
51      cudaEvent_t  start, stop, start1, stop1, startall, stopall;
52      cudaEventCreate(&start);
53      cudaEventCreate(&stop);
54      cudaEventCreate(&start1);
55      cudaEventCreate(&stop1);
56      cudaEventCreate(&startall);
57      cudaEventCreate(&stopall);
58
59      // allocate memory on the GPU
60      cudaMalloc((void**)&ad, size_2D);
61      cudaMalloc((void**)&xd, size_1D);
62      cudaMalloc((void**)&x_tempd, size_1D);
63
64      //Start the timer for the whole program ***
65      cudaEventRecord(startall, 0);
66
67      // transfer a to device memory
68      cudaEventRecord(start, 0);
69      cudaMemcpy(ad, a, size_2D, cudaMemcpyHostToDevice);
70      cudaDeviceSynchronize();
71      cudaEventRecord(stop, 0);
72      cudaEventSynchronize(stop);
73      cudaEventElapsedTime(&H2DTime1, start, stop);
```

Στο σημείο αυτό ξεκινά το πρώτο βήμα του αλγορίθμου κατά το οποίο δημιουργούμε τον άνω τριγωνικό πίνακα. Μέσα σε μια επαναληπτική δομή for για κάθε στήλη, αρχικά ελέγχουμε αν το στοιχείο που χρησιμοποιούμε για να μηδενίσουμε τα υπόλοιπα στοιχεία είναι 0 και στη συνέχεια καλούμε το kernel μέσω της συνάρτησης upper_triangular, το οποίο θα εκτελέσει τον μηδενισμό των στοιχείων της στήλης i. Τα ορίσματα της συνάρτησης είναι ο αριθμός των εξισώσεων n, ο πίνακας ad και ο δείκτης της στήλης i.

```
75      // kernel1
76      for(int i=0; i<n-1; i++)
77      {
78          if(a[(n+1)*i+i] == 0.0)
79          {
80              printf("Error, Division by zero!");
81              exit(0);
82          }
83          cudaEventRecord(start, 0);
84          upper_triangular<<<gridSize, blockSize>>>(n, ad, i);
85          cudaDeviceSynchronize();
86          cudaEventRecord(stop, 0);
87          cudaEventSynchronize(stop);
88          cudaEventElapsedTime(&KernelTime, start, stop);
89          KernelTime1 += KernelTime;
90      }
91
```

C: > Users > Dimitris > Downloads > sol1.cu

```
1   #include <stdio.h>
2   #include <cuda.h>
3   #define N (2048 * 2048)
4
5   __global__ void upper_triangular(int n, float *a, int i)
6   {
7       int j = threadIdx.x + blockIdx.x * blockDim.x;
8       float ratio;
9
10      if (j<n && j>=i+1)
11      {
12          ratio = a[j*(n+1)+i]/a[i*(n+1)+i];
13
14          for (int k=0; k<n+1; k++)
15          {
16              a[j*(n+1)+k] = a[j*(n+1)+k] - ratio*a[i*(n+1)+k];
17          }
18      }
19  }
20
```

Το j χρησιμοποιείται σαν δείκτης ο οποίος δείχνει στην αρχή των γραμμών που εμπεριέχουν τα στοιχεία που πρέπει να μηδενιστούν σε κάθε επανάληψη. Η συνάρτηση εκτελείται για όσα threads έχουν ID μικρότερο του n και μεγαλύτερο ή ίσο του i+1 ώστε τα στοιχεία που μηδενίζονται να ανήκουν σε επόμενες γραμμές από το i. Ο αλγόριθμος της συνάρτησης ακολουθεί την λογική του αλγορίθμου που χρησιμοποιήσαμε στην OpenMP. Αφού ολοκληρωθεί η εκτέλεση του kernel αποθηκεύουμε τον χρόνο εκτέλεσης στο KernelTime και σε κάθε επανάληψη τον προσθέτουμε στο KernelTime1.

Στη συνέχεια αντιγράφουμε τον ad στον a του επεξεργαστή και μετράμε τον χρόνο αντιγραφής. Υπολογίζουμε τον άγνωστο x[n-1] και αντιγράφουμε τον x στον xd της κάρτας γραφικων.

```
92      // transfer a from device
93      cudaEventRecord(start, 0);
94      cudaMemcpy(a, ad, size_2D, cudaMemcpyDeviceToHost);
95      cudaDeviceSynchronize();
96      cudaEventRecord(stop, 0);
97      cudaEventSynchronize(stop);
98      cudaEventElapsedTime(&D2HTime1, start, stop);
99
100     x[n-1] = a[(n-1)*(n+1)+n]/a[(n-1)*(n+1)+n-1];
101
102     // transfer x to device
103     cudaEventRecord(start, 0);
104     cudaMemcpy(xd, x, size_1D, cudaMemcpyHostToDevice);
105     cudaDeviceSynchronize();
106     cudaEventRecord(stop, 0);
107     cudaEventSynchronize(stop);
108     cudaEventElapsedTime(&H2DTime2, start, stop);
```

Το επόμενο βήμα είναι να βρούμε τους υπόλοιπους αγνώστους του συστήματος χρησιμοποιώντας την αντικατάσταση προς τα πίσω. Για να το πετύχουμε αυτό χρησιμοποιούμε μια επαναληπτική δομή for η οποία ξεκινά από την προτελευταία εξίσωση και τελειώνει στην πρώτη. Αρχικά αποθηκεύουμε στο x_i την δεξιά πλευρά της εξίσωσης i. Στη συνέχεια καλούμε το kernel μέσω της συνάρτησης backwardSubstitution με ορίσματα τα n, ad, xd, I και x_tempd

C: > Users > Dimitris > Downloads > sol1.cu

```
110     // kernel2
111     for(int i=n-2; i>=0; i--)
112     {
113         x_i = a[i*(n+1)+n];
114
115         cudaEventRecord(start1, 0);
116         backwardSubstitution2<<<gridSize, blockSize>>>(n, ad, xd, i, x_tempd);
117         cudaDeviceSynchronize();
118         cudaEventRecord(stop1, 0);
119         cudaEventSynchronize(stop1);
120         cudaEventElapsedTime(&KernelTime, start1, stop1);
121         KernelTime2 += KernelTime;
122
123         // transfer x_temp from device
124         cudaEventRecord(start, 0);
125         cudaMemcpy(x_temp, x_tempd, size_1D, cudaMemcpyDeviceToHost);
126         cudaDeviceSynchronize();
127         cudaEventRecord(stop, 0);
128         cudaEventSynchronize(stop);
129         cudaEventElapsedTime(&D2HTime2, start, stop);
130         D2HTime2all += D2HTime2;
131
132         for (int k=i+1; k<n; k++)
133         {
134             x_i += - x_temp[k];
135         }
136
137         x[i] = x_i/a[i*(n+1)+i];
```

```
        for (int k=i+1; k<n; k++)
        {
            x_i += - x_temp[k];
        }

        x[i] = x_i/a[i*(n+1)+i];

        // transfer x to device
        cudaEventRecord(start, 0);
        cudaMemcpy(xd, x, size_1D, cudaMemcpyHostToDevice);
        cudaDeviceSynchronize();
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&H2DTime3, start, stop);
        H2DTime3all += H2DTime3;
    }
```

```
__global__ void backwardSubstitution2(int n, float *a, double *x, int i,double *x_temp)
{
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    if (j<n && j>=i+1)
    {
        x_temp[j] = a[i * (n+1) + j] * x[j];

    }
}
```

Στην backwardSubstitution πολλαπλασιάζουμε κάθε γνωστή λύση με τον συντελεστή της και την αποθηκεύουμε στη θέση j του x_tempd. Η πρόσθεση δεν μπορει να εκτελεστεί παράλληλα διότι θα υπάρχει ανταγωνισμός ανάμεσα στα threads (x[i] += a[i*(n+1)+j]*x[j]). Στη συνέχεια μετράμε τον χρόνο εκτέλεσης του kernel, τον προσθέτουμε στο KernelTime2 και αντιγράφουμε τον x_tempd στον x_temp του επεξεργαστή. Αφαιρούμε από το x_i κάθε στοιχείο που αποθηκεύτηκε στον x_temp στην εκάστοτε επανάληψη, διαιρούμε το τελικό x_i με τον συντελεστή του ώστε να βρούμε τον άγνωστο και τον αποθηκεύουμε στην θέση του στον πίνακα x. Στο τέλος κάθε επανάληψης αντιγράφουμε τον επικαιροποιημένο πίνακα x στον xd ώστε να χρησιμοποιηθεί στην επόμενη επανάληψη.

Στο τέλος της GaussElimination υπολογίζουμε τους συνολικούς χρόνους, εκτυπώνουμε αναλυτικά τους χρόνους κάθε βήματος μαζί με τα αποτελέσματα και αποδεσμεύουμε την μνήμη της κάρτας γραφικών.

C: > Users > Dimitris > Downloads > sol1.cu

```
    cudaDeviceSynchronize();
    cudaEventRecord(stopall, 0);
    cudaEventSynchronize(stopall);
    cudaEventElapsedTime(&TotalTime, startall, stopall);

    // display the timing results
    printf("Transfer a to device memory: %f\n", H2DTime1);
    printf("Time for kernel 1: %f\n",KernelTime1);
    printf("Transfer a from device: %f\n",D2HTime1);
    printf("Transfer x to device: %f\n",H2DTime2);
    printf("Transfer x to device all: %f\n", H2DTime3all);
    printf("Transfer x_temp from device all: %f\n", D2HTime2all);
    printf("Time for kernel 2: %f\n", KernelTime2);

    float Kernel_Time = KernelTime1 + KernelTime2;
    float MemCopy = H2DTime1 + H2DTime2 + H2DTime3all + D2HTime1 + D2HTime2all;
    float SerialTime = TotalTime - MemCopy - Kernel_Time;

    printf("\nTime for %dx%d array multiplication.\n", n, n);
    printf("MemCopy: %f sec, Kernel: %f sec, Serial: %f sec, Total: %f sec\n", MemCopy/1000, Kernel_Time/1000, SerialTime/1000, TotalTime/1000);

    // Print the solution
    printf("\nSolution:\n");
    for(int i=0; i<n; i++)
    {
        printf("x[%d] = %f\n", i, x[i]);
    }

    // free the memory allocated on the GPU
    cudaFree(ad);
    cudaFree(xd);
    cudaFree(x_tempd);
```

Αλγόριθμος 2:

Στον πρώτο αλγόριθμο, επειδή διατηρούσαμε σταθερό τον αριθμό των threads για όλους τους αριθμούς αγνώστων, δεν είχαμε τη βέλτιστη απόδοση. Συγκεκριμένα, για μικρά n, χρησιμοποιούσαμε παραπάνω νήματα από αυτά που χρειαζόμασταν και για μεγάλα n, (άνω των 2048) δεν χρησιμοποιούσαμε αρκετά. Έτσι, υλοποιήσαμε μια δεύτερη μέθοδο στην οποία χρησιμοποιούμε δισδιάστατα blocks και grid, με τον αριθμό των threads να εξαρτάται από τον αριθμό των αγνώστων. Παρακάτω φαίνονται οι αλλαγές:

C: > Users > Dimitris > Downloads > sol2D.cu

```
}

void GaussElimination(float *a, double *x, int n, double *x_temp)
{
    int size_1D = (n+1) * sizeof(double);
    int size_2D = (n+1) * (n+1) * sizeof(double);
    double *xd, *x_tempd, x_i;
    float *ad;
    float TotalTime, H2DTime1, KernelTime, D2HTime1, H2DTime2, D2HTime2, H2DTime3;
    float KernelTime1 = 0;
    float D2HTime2all = 0;
    float H2DTime3all = 0;
    float KernelTime2 = 0;


    dim3 blockSize(16, 16);
    dim3 gridSize((n + blockSize.x - 1) / blockSize.x, (n + blockSize.y - 1) / blockSize.y);

```

C: > Users > Dimitris > Downloads > sol2D.cu

```cpp
#include <stdio.h>
#include <cuda.h>
#define N (2048 * 2048)

__global__ void upper_triangular(int n, float *a, int i)
{
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int j = col + row * gridDim.x * blockDim.x;
    float ratio;

    if (j < n && j >= i + 1)
    {
        ratio = a[j * (n + 1) + i] / a[i * (n + 1) + i];

        for (int k = 0; k < n + 1; k++)
        {
            a[j * (n + 1) + k] = a[j * (n + 1) + k] - ratio * a[i * (n + 1) + k];
        }
    }
}

__global__ void backwardSubstitution2(int n, float *a, double *x, int i, double *x_temp)
{
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int j = col + row * gridDim.x * blockDim.x;

    if (j < n && j >= i + 1)
    {
        x_temp[j] = a[i * (n + 1) + j] * x[j];
    }
```

Έτσι, έχουμε πρόσβαση στις συντεταγμένες του grid με τις μεταβλητές col και row, που χρησιμοποιούνται για τον υπολογισμό της μεταβλητής j, η οποία όπως και πριν, αντιπροσωπεύει τις γραμμές των πινάκων.

Αποτελέσματα:

Με τη χρήση του παρακάτω συστήματος επικυρώσαμε την σωστή λειτουργία του προγράμματός μας.

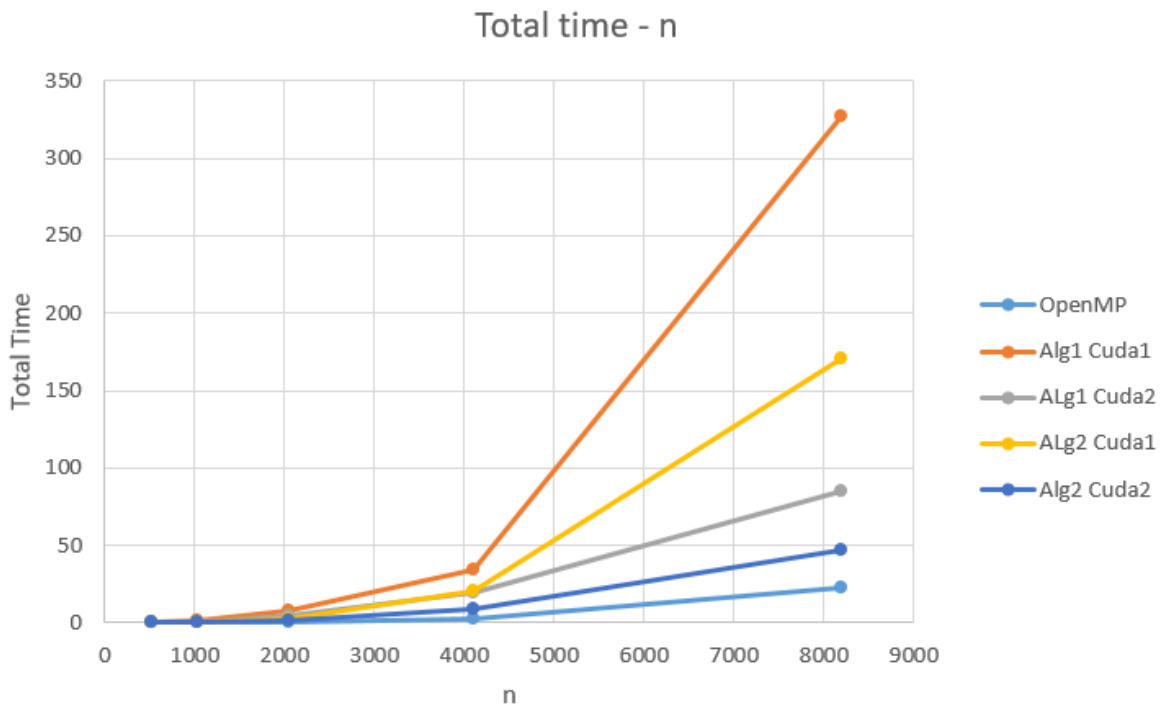
```
Enter number of unknowns: 4
a[0][0] = 1
a[0][1] = 9
a[0][2] = -3
a[0][3] = 6
a[0][4] = 7
a[1][0] = -5
a[1][1] = 2
a[1][2] = 0
a[1][3] = 1
a[1][4] = 3
a[2][0] = 1
a[2][1] = -2
a[2][2] = 8
a[2][3] = -4
a[2][4] = 5
a[3][0] = 2
a[3][1] = 6
a[3][2] = 0
a[3][3] = 1
a[3][4] = 8
Transfer a to device memory: 0.046496
Time for kernel 1: 0.385952
Transfer a from device: 0.023712
Transfer x to device: 0.011584
Transfer x to device all: 0.032800
Transfer x_temp from device all: 0.031968
Time for kernel 2: 0.356288

Time for 4x4 array multiplication.
MemCopy: 0.000147 sec, Kernel: 0.000742 sec, Serial: 0.000099 sec, Total: 0.000988 sec

Solution:
x[0] = -0.146402
x[1] = 1.506203
x[2] = 0.647643
x[3] = -0.744417
```

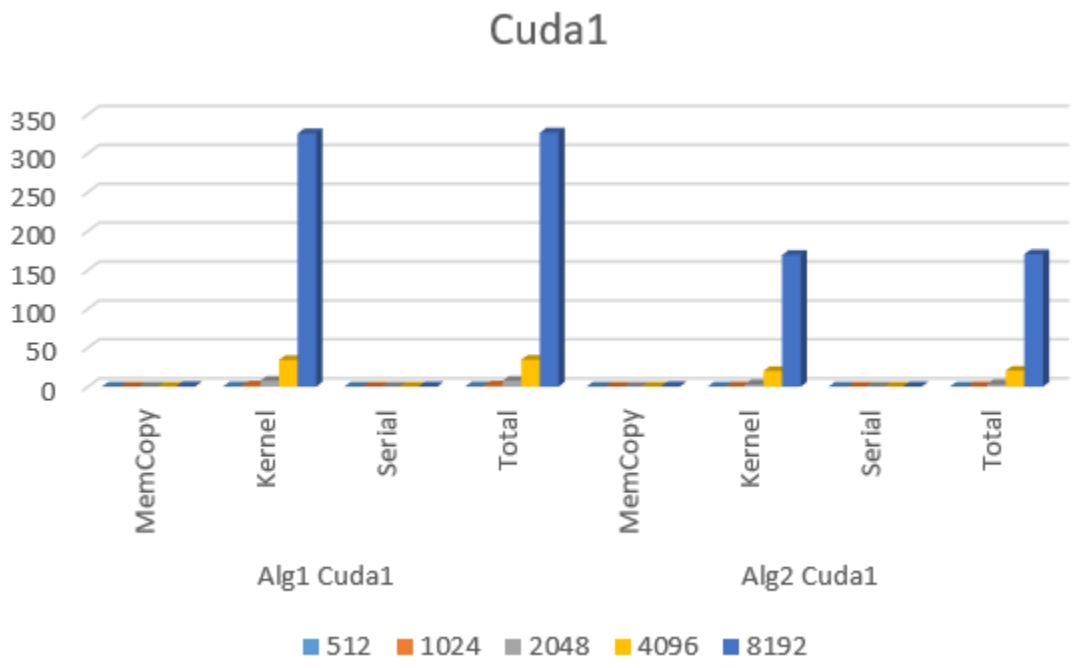
Παρακάτω φαίνονται τα αποτελέσματα των δύο αλγορίθμων για τα μηχανήματα cuda1 και cuda2. Επίσης φαίνονται και οι χρόνοι εκτέλεσης της OpenMP στο pxeon2 για 64 πυρήνες. Τα προγράμματα εκτελέστηκαν για πίνακες 512X512, 1024X1024, 2048X2048, 4096X4096 και 8192X8192. Η μνήμη των cuda1 και cuda2 δεν επέτρεπε την εκτέλεση του προγράμματος για πίνακες μεγέθους 16384X16384. Αυτό συμβαίνει διότι η global memory των καρτών γραφικών δεν επαρκεί για την αποθήκευση τόσων μεγάλων πινάκων, όμως αν αξιοποιούσαμε την shared memory η εκτέλεση του προγράμματος θα ήταν δυνατή ακόμα και για τέτοια μεγέθη προβλήματος (με χρήση τεχνικών όπως τα tiles που συζητήσαμε κατά τη διάρκεια του μαθήματος).

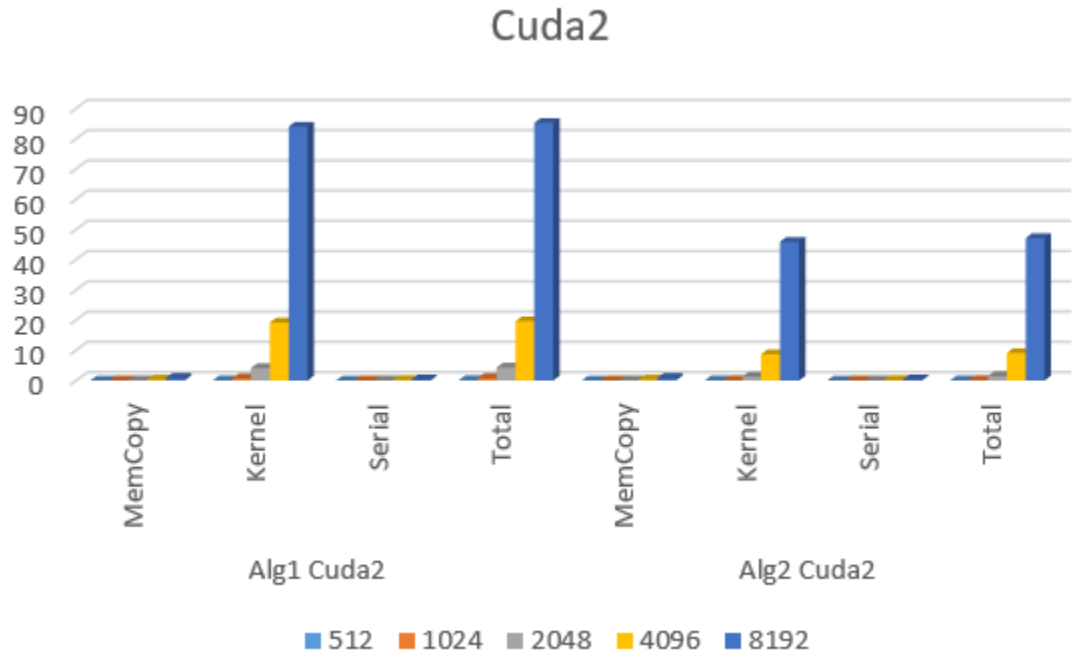
Πίνακες	Alg. 1 @cuda1				Alg. 2 @cuda1			
	MemCopy	Kernel	Serial	Total	MemCopy	Kernel	Serial	Total
0.5K x 0.5K	0.011183sec	0.251175sec	0.011011sec	0.273370sec	0.011213sec	0.077232sec	0.011113sec	0.099559sec
1K x 1K	0.025845sec	1.331230sec	0.020622sec	1.377698sec	0.025688sec	0.401610sec	0.021213sec	0.448511sec
2K x 2K	0.069119sec	7.475465sec	0.044444sec	7.589028sec	0.067612sec	2.868077sec	0.043979sec	2.979668sec
4K x 4K	0.208696sec	34.294033sec	0.100930sec	34.603657sec	0.209297sec	20.046535sec	0.100518sec	20.356350sec
8K x 8K	0.703526sec	325.618805sec	0.263844sec	326.58618sec	0.702259sec	169.272034sec	0.251844sec	170.226135sec

Πίνακες	Alg. 1 @cuda2				Alg. 2 @cuda2			
	MemCopy	Kernel	Serial	Total	MemCopy	Kernel	Serial	Total
0.5K x 0.5K	0.015961sec	0.097830sec	0.016278sec	0.130068sec	0.016012sec	0.028108sec	0.015998sec	0.060118sec
1K x 1K	0.036955sec	0.702172sec	0.032676sec	0.771803sec	0.036875sec	0.111568sec	0.032819sec	0.181261sec
2K x 2K	0.095122 sec	4.134621 sec	0.070448 sec	4.300191 sec	0.094363sec	1.297747sec	0.070773sec	1.462883sec
4K x 4K	0.271502sec	19.187576sec	0.155283sec	19.614361sec	0.271923sec	8.700479sec	0.152572sec	9.124973sec
8K x 8K	0.878752sec	83.976761sec	0.360758sec	85.216263sec	0.879063sec	45.878448sec	0.362707sec	47.120220sec

Πίνακες	Alg. 3 @pxeon2 OpenMP 64 cores
0.5K x 0.5K	0.093536sec
1K x 1K	0.144123sec
2K x 2K	0.557147sec
4K x 4K	2.803624sec
8K x 8K	22.821901sec
16K x 16K	258.595404sec







<u>Συμπεράσματα:</u>

Παρατηρούμε πως η αλλαγή των block και grid από μονοδιάστατα σε δισδιάστατα οδήγησε σε σημαντική βελτίωση του χρόνου εκτέλεσης και για τα δύο μηχανήματα ανεξαρτήτων αριθμού αγνώστων. Παρ' όλα αυτά η εκτέλεση με OpenMP στο pxeon2 με 64 πυρήνες είχε το μικρότερο χρόνο εκτέλεσης, ακόμα και σε μεγάλα μεγέθη προβλήματος (συστήματα 8192X8192). Επίσης, ο χρόνος εκτέλεσης στο cuda2 είναι πάντα μικρότερος από αυτόν του cuda1, καθώς έχει ισχυρότερη κάρτα γραφικών. Τέλος, το πιο χρονοβόρο κομμάτι της εκτέλεσης ήταν τα δύο Kernel με το Kernel1 να παίρνει τον περισσότερο χρόνο μέχρι να ολοκληρωθεί.