



---

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ  
ΥΠΟΛΟΓΙΣΤΩΝ

Δ.Π.Μ.Σ ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ & ΜΗΧΑΝΙΚΗ ΜΑΘΗΣΗ

---

Εξαμηνιαία Εργασία

**Παράλληλες Αρχιτεκτονικές για Μηχανική Μάθηση**

parml04

Νικήτας Νικόλαος (03400043)  
Ζωγραφάκης Δημήτριος (03400050)  
Μαρκολέφας Φίλιππος (03400061)  
Σαμψάκης-Μπακόπουλος Μύρων (03400074)

Αθήνα

30/06/2020

# Περιεχόμενα

Σημειώσεις . . . . .	2
<b>1 Παραλληλοποίηση σε CPU</b>	<b>3</b>
1.1 Παραλληλοποίηση του με χρήση προγραμματιστικού μοντέλου OpenMP . . . . .	3
1.1.1 Αποτελέσματα . . . . .	4
1.2 Υλοποίηση τους με χρήση της βιβλιοθήκης OpenBLAS . . . . .	4
1.2.1 dgemm . . . . .	5
1.2.2 dgemm_ta . . . . .	5
1.2.3 dgemm_tb . . . . .	5
1.2.4 Αποτελέσματα . . . . .	6
<b>2 Παραλληλοποίηση σε GPU</b>	<b>7</b>
2.1 Παράλληλη naïve έκδοση της GEMM για GPUs με CUDA . . . . .	7
2.1.1 dgemm_naive . . . . .	8
2.1.2 dgemm_ta_naive . . . . .	8
2.1.3 dgemm_tb_naive . . . . .	8
2.1.4 Αποτελέσματα . . . . .	9
2.2 Παράλληλη shmem έκδοση της GEMM για GPUs με CUDA . . . . .	9
2.2.1 dgemm_optimized . . . . .	11
2.2.2 dgemm_ta_optimized . . . . .	12
2.2.3 dgemm_tb_optimized . . . . .	13
2.2.4 Αποτελέσματα . . . . .	14
2.2.5 Διαφοροποίηση (Static Shared Memory) . . . . .	14
2.3 Παράλληλη έκδοση της GEMM για GPUs με cuBLAS . . . . .	15
2.3.1 dgemm . . . . .	16
2.3.2 dgemm_ta . . . . .	16
2.3.3 dgemm_tb . . . . .	17
2.3.4 Αποτελέσματα . . . . .	17
<b>3 Πειράματα και μετρήσεις επιδόσεων</b>	<b>18</b>
3.1 Κλιμακωσιμότητα σε CPU . . . . .	19
3.2 Σύγκριση επιδόσεων σε CPU και GPU . . . . .	20

---

## Σημειώσεις

Η δομή της εργασίας έχει ως ακολουθεί: Αρχικά, θα παρουσιαστούν οι υλοποιήσεις των ζητούμενων συναρτήσεων, μαζί με τις εγγραφές των αρχείων (.out/.err) που δημιουργούν (Ενότητες 1, 2). Σκοπός των παραπάνω ενότητων αποτελεί η αποτύπωση της τεχνικής που χρησιμοποιήθηκε, ενώ παράλληλα να σχολιαστεί ο κώδικας. Στην συνέχεια, θα δημιουργηθούν τα ζητούμενα διαγράμματα και θα γίνουν σχολιασμοί όσον αφορά τους χρόνους εκτελέσεις των παραπάνω συναρτήσεων καθώς και συγκρίσεις μεταξύ των υλοποιήσεων (Ενότητα 3).

Στο παραδοτέο zip, θα συμπεριληφθούν οι κώδικες των αρχείων linalg.c και linalg.cu, μαζί με την παρούσα αναφορά.

Όσον αφορά το αρχείο linalg.cu, θα δοθεί σε 2 μορφές. Μία που κάνει χρήση δυναμικής κοινής μνήμης (linalg\_dynamic.cu) και μία που χρησιμοποιεί στατική κοινή μνήμη (linalg\_static.cu). **Ως λύση παραδίδεται το (linalg\_dynamic.cu).** Το (linalg\_static.cu), αποτελεί κυρίως πειραματισμό της ομάδας και εφόσον υπήρξε κάποιο καλύτερο αποτέλεσμα από το (linalg\_dynamic.cu), προστίθεται στον φάκελο εργασίας.

### Κύριες πηγές αναφοράς της παρούσας εργασίας αποτελούν

- OpenMP: [Εγχειρίδια](#) του OpenMP.
- BLAS: [Math Kernel Library](#) της Intel.
- CUDA: [Εγχειρίδια](#) της Nvidia καθώς και το εν λόγω [φόρουμ](#).
- cuBLAS: [Εγχειρίδια](#) του cuBLAS της Nvidia.

### Παραδοτέο

```
~/parm04_cpu_gpu/  
├── src  
│   ├── linalg.c          # ==> Cpu Parallelism  
│   ├── linalg_dynamic.cu #  
│   └── linalg_static.cu  # => GPU Parallelism  
└── Documentation.pdf
```

Σχήμα 1: Δομή του Πρότζεκτ.

καθώς και το [Repository](#) του μαθήματος.

# Παραλληλοποίηση σε CPU

## 1.1 Παραλληλοποίηση του με χρήση προγραμματιστικού μοντέλου OpenMP

Στην περίπτωση αυτή, καλούμαστε να δομήσουμε την εντολή που θα παραλληλοποιήσει μέσω OpenMP τρεις σχετικά όμοιες function. Αυτές περιέχουν ένα τριπλό for loop, το οποίο “περνάει” τις θέσεις του τελικού πίνακα (συντεταγμένες  $i, j$ , οι οποίες έχουν μετατραπεί όμως σε μία ενιαία διάσταση, με την πράξη  $i * N + j$ ). Ο δείκτης  $k$ , περνάει τις θέσεις των πινάκων A,B, και πολλαπλασιάζει element-by-element τα στοιχεία τους, αθροίζοντας τα στην μεταβλητή sum. Η sum όμως σε κάθε θέση του πίνακα C (δηλαδή για κάθε διαφορετικό  $i, j$ ), θα είναι διαφορετική. Επίσης, δεδομένου ότι τα loops των  $i, j$  είναι ανεξάρτητα, μπορούμε να τα “ενοποιήσουμε” στην παραλληλοποίηση, δηλαδή αντί να σπάσουμε το  $i$ -loop, και κάθε πυρήνας να αναλαμβάνει ένα εύρος τιμών του  $i$ , να σπάσουμε το σύνολο του  $i, j$  loop, ώστε κάθε πυρήνας να παίρνει έναν αριθμό από  $(i, j)$  tuples.

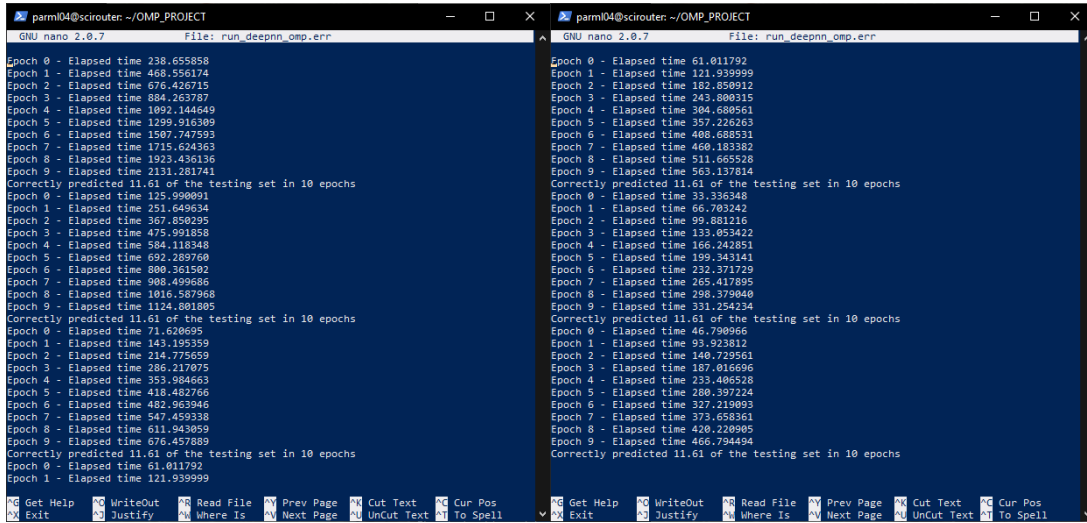
Συνεπώς, και στις τρεις περιπτώσεις, το header θα γίνει:

```
1 int OMP_NUM_THREADS;  
2 #pragma omp parallel for collapse(2) private(k, sum) num_threads(OMP_NUM_THREADS)
```

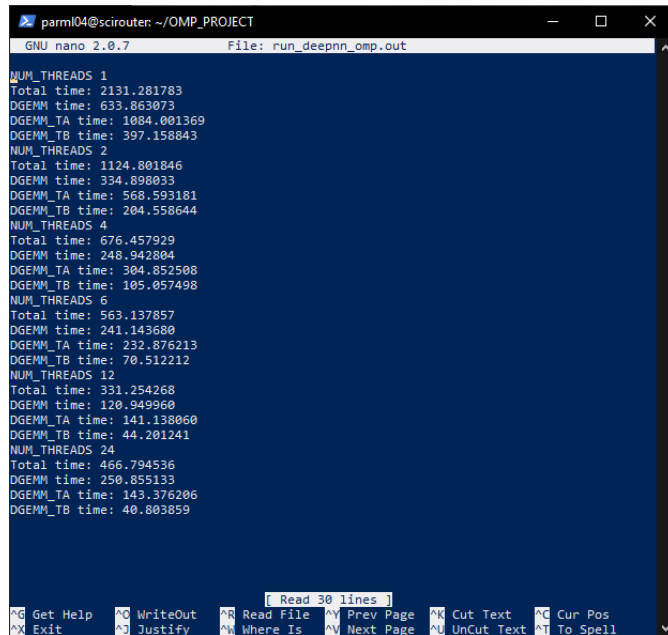
Όπου

- Pragma omp: δηλώνει ότι θα χρησιμοποιήσουμε την OpenMP
- For: θα γίνει παράλληλο for loop
- collapse(2): τα δύο παρακάτω loops θα ενοποιηθούν (collapse(1) είναι ισοδύναμο με το να μην το βάλουμε καθόλου)
- private(k,sum): οι δύο αυτές μεταβλητές θα είναι διαφορετικές σε κάθε  $(i,j)$  loop, άρα θα πρέπει να αναφέρονται σε διαφορετικές θέσεις μνήμης εντός του κάθε διαφορετικού loop, άρα πρέπει να γίνουν private. Οι  $i,j$  δεν απαιτείται να γίνουν private, καθώς δηλώνεται αυτόματα όταν του ορίζουμε να κάνει collapse(2), δηλαδή να παραλληλοποιήσει τα 2 παρακάτω loops. Επίσης, οι πίνακες A,B,C δεν χρειάζεται να δηλωθούν ως Public (το Public είναι default). Σε αυτούς τους πίνακες μάλιστα δεν υπάρχουν race conditions, καθώς το μόνο ταυτόχρονο που γίνεται, είναι η ανάγνωση. Η εγγραφή γίνεται σε διαφορετική θέση ανά ζεύγος  $i,j$ . Τέλος, ο αριθμός των threads (OMP\_NUM\_THREADS) είναι μεταβλητή συστήματος, η οποία ορίζεται όταν τρέχουμε με command το αρχείο. Στην περίπτωση αυτή, θα πάρει τιμές [1, 2, 4, 6, 12, 24].

### 1.1.1 Αποτελέσματα



Σχήμα 1.1: OpenMP (err).



Σχήμα 1.2: OpenMP (out).

## 1.2 Υλοποίηση τους με χρήση της βιβλιοθήκης OpenBLAS

Η OpenBlas είναι μια βιβλιοθήκη μαθηματικών, γραμμένη και compiled σε FORTRAN, η οποία περιέχει optimized συναρτήσεις, συμπεριλαμβανομένου και αυτής για πολλαπλασιασμό πινάκων.

Θα χρησιμοποιήσουμε την συνάρτηση **cblas\_dgemm**(Order, TransA, TransB, M, N, K, A, lda, b, ldb, beta, ldc) προκειμένου να υπολογιστούν τα ζητούμενα εσωτερικά γινόμενα. Η εν λόγω συνάρτηση, υλοποιεί τις παρακάτω πράξεις:

$$C = \alpha \cdot op(A) \cdot op(B) + \beta C$$

Το *op*, απευθύνεται στα πεδία *transA*, *transB*. Συγκεκριμένα, τα ορίσματα της συνάρτησης ορίζονται ως εξής:

Πίνακας 1.1: Συνάρτηση `cblas_dgemm`.

Παράμετρος	Επεξήγηση
Order	Διάταξη κατά Γραμμή ή Στήλη (CblasRowMajor ή CblasColMajor)
TransA	$A$ ή $A^T$ (CblasNoTrans ή CblasTrans)
TransB	$B$ ή $B^T$ (CblasNoTrans ή CblasTrans)
M	Αριθμός Σειρών των πινάκων A, C
N	Αριθμός Στηλών των πινάκων B, C
K	Αριθμός Στηλών του πίνακα A, και Σειρών του πίνακα B
A	Ο πίνακας A
lda	Δεύτερη διάσταση του πίνακα A
alpha	Ο συντελεστής $\alpha$
B	Ο πίνακας B
ldb	Δεύτερη διάσταση του πίνακα B
beta	Ο συντελεστής $\beta$
ldc	Δεύτερη διάσταση του πίνακα C

### 1.2.1 `dgemm`

$$C = 1.0 \cdot (A \cdot B) + 0.0 \cdot C$$

```

1 cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
2             M, N, K,
3             1.0,
4             A, K,
5             B, N,
6             0.0,
7             C, N);
```

### 1.2.2 `dgemm_ta`

$$C = 1.0 \cdot (A^T \cdot B) + 0.0 \cdot C$$

```

1 cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans,
2             M, N, K,
3             1.0,
4             A, M,
5             B, N,
6             0.0,
7             C, N);
```

### 1.2.3 `dgemm_tb`

Υλοποιείται η πράξη και στην συνέχεια αντιγράφονται τα αποτελέσματα από τον πίνακα  $C$  στον  $D$ .

$$C = 1.0 \cdot (A \cdot B^T) + 1.0 \cdot C$$

$$D = C$$

```

1  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans,
2      M, N, K,
3      1.0,
4      A, K,
5      B, K,
6      1.0,
7      C, N);
8
9  for (int i=0; i<N*M; i++){ D[i] = C[i]; }

```

## 1.2.4 Αποτελέσματα

```

GNU nano 2.0.7 File: run_deepnn_bias.err
Epoch 0 - Elapsed time 24.585377
Epoch 1 - Elapsed time 48.914822
Epoch 2 - Elapsed time 73.204987
Epoch 3 - Elapsed time 97.529965
Epoch 4 - Elapsed time 121.880148
Epoch 5 - Elapsed time 146.441358
Epoch 6 - Elapsed time 171.087045
Epoch 7 - Elapsed time 195.479835
Epoch 8 - Elapsed time 220.215889
Epoch 9 - Elapsed time 244.790881
Correctly predicted 1.72 of the testing set in 10 epochs
Epoch 0 - Elapsed time 14.514588
Epoch 1 - Elapsed time 29.007241
Epoch 2 - Elapsed time 43.586391
Epoch 3 - Elapsed time 57.796361
Epoch 4 - Elapsed time 72.136377
Epoch 5 - Elapsed time 86.349217
Epoch 6 - Elapsed time 100.474843
Epoch 7 - Elapsed time 114.615417
Epoch 8 - Elapsed time 128.778854
Epoch 9 - Elapsed time 142.913483
Correctly predicted 1.72 of the testing set in 10 epochs
Epoch 0 - Elapsed time 9.011672
Epoch 1 - Elapsed time 17.922741
Epoch 2 - Elapsed time 26.762887
Epoch 3 - Elapsed time 35.579221
Epoch 4 - Elapsed time 44.402947
Epoch 5 - Elapsed time 53.181106
Epoch 6 - Elapsed time 61.952583
Epoch 7 - Elapsed time 70.745241
Epoch 8 - Elapsed time 79.562199
Epoch 9 - Elapsed time 88.426926
Correctly predicted 1.72 of the testing set in 10 epochs
Epoch 0 - Elapsed time 8.331182
Epoch 1 - Elapsed time 16.474637

GNU nano 2.0.7 File: run_deepnn_bias.err
Epoch 0 - Elapsed time 8.331182
Epoch 1 - Elapsed time 16.474637
Epoch 2 - Elapsed time 24.584251
Epoch 3 - Elapsed time 32.714875
Epoch 4 - Elapsed time 40.786614
Epoch 5 - Elapsed time 48.865636
Epoch 6 - Elapsed time 56.925128
Epoch 7 - Elapsed time 64.960728
Epoch 8 - Elapsed time 72.986378
Epoch 9 - Elapsed time 81.037952
Correctly predicted 1.72 of the testing set in 10 epochs
Epoch 0 - Elapsed time 6.581130
Epoch 1 - Elapsed time 13.015751
Epoch 2 - Elapsed time 19.701129
Epoch 3 - Elapsed time 26.179209
Epoch 4 - Elapsed time 32.610830
Epoch 5 - Elapsed time 39.036493
Epoch 6 - Elapsed time 45.407393
Epoch 7 - Elapsed time 51.941270
Epoch 8 - Elapsed time 58.372875
Epoch 9 - Elapsed time 64.800946
Correctly predicted 1.72 of the testing set in 10 epochs
Epoch 0 - Elapsed time 7.509185
Epoch 1 - Elapsed time 14.988675
Epoch 2 - Elapsed time 22.723766
Epoch 3 - Elapsed time 30.834631
Epoch 4 - Elapsed time 38.700633
Epoch 5 - Elapsed time 46.250745
Epoch 6 - Elapsed time 53.449411
Epoch 7 - Elapsed time 60.651793
Epoch 8 - Elapsed time 67.914581
Epoch 9 - Elapsed time 75.131930
Correctly predicted 1.72 of the testing set in 10 epochs

```

Σχήμα 1.3: cBlas (err).

```

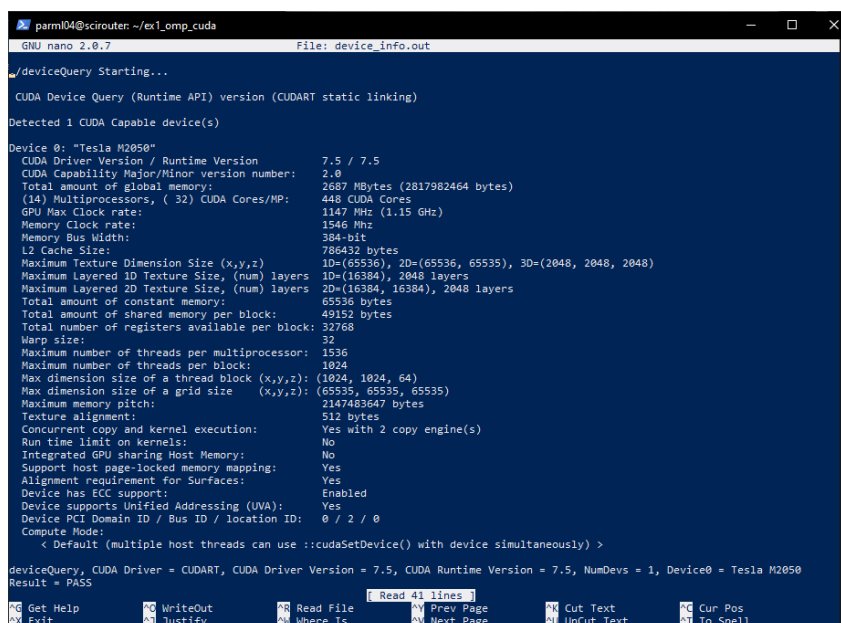
GNU nano 2.0.7 File: run_deepnn_bias.out
NUM_THREADS 1
Total time: 244.790734
DGEMM time: 67.124818
DGEMM_TA time: 83.441351
DGEMM_TB time: 79.155355
NUM_THREADS 2
Total time: 142.913517
DGEMM time: 35.914273
DGEMM_TA time: 45.970665
DGEMM_TB time: 43.215657
NUM_THREADS 4
Total time: 88.426972
DGEMM time: 19.522483
DGEMM_TA time: 26.294149
DGEMM_TB time: 23.431148
NUM_THREADS 6
Total time: 81.037994
DGEMM time: 17.623101
DGEMM_TA time: 21.042938
DGEMM_TB time: 21.645427
NUM_THREADS 12
Total time: 64.800990
DGEMM time: 12.219146
DGEMM_TA time: 15.923911
DGEMM_TB time: 15.110114
NUM_THREADS 24
Total time: 75.131975
DGEMM time: 11.418573
DGEMM_TA time: 17.491970
DGEMM_TB time: 14.459852

```

Σχήμα 1.4: cBlas (out).

# Παραλληλοποίηση σε GPU

Αρχικά εκτελείται το πρόγραμμα deviceQuery στο μηχανήμα termi. Μετά το πέρας της εκτέλεσης λαμβάνονται τα ακόλουθα



```
parmi04@scirouten: ~/ex1_omp_cuda
GNU nano 2.0.7 File: device_info.out
./deviceQuery Starting...
CUDA Device Query (Runtime API) version (CUDART static linking)
Detected 1 CUDA Capable device(s)
Device 0: "Tesla M2050"
  CUDA Driver Version / Runtime Version      7.5 / 7.5
  CUDA Capability Major/Minor version number: 2.0
  Total amount of global memory:             2687 MBytes (2817982464 bytes)
  (14) Multiprocessors, ( 32) CUDA Cores/MP: 448 CUDA Cores
  GPU Max Clock rate:                       1147 MHz (1.15 GHz)
  Memory Clock rate:                        1546 Mhz
  Memory Bus Width:                         384-bit
  L2 Cache Size:                            786432 bytes
  Maximum Texture Dimension Size (x,y,z)    1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
  Maximum Layered ID Texture Size, (num) layers 1D=(16384), 2D=(16384, 2048) layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
  Total amount of constant memory:           65536 bytes
  Total amount of shared memory per block:   49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                32
  Maximum number of threads per multiprocessor: 1536
  Maximum number of threads per block:       1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z): (65535, 65535, 65535)
  Maximum memory pitch:                     2147483647 bytes
  Texture alignment:                        512 bytes
  Concurrent copy and kernel execution:      Yes with 2 copy engine(s)
  Run time limit on kernels:                 No
  Integrated GPU sharing Host Memory:         No
  Support host page-locked memory mapping:   Yes
  Alignment requirement for Surfaces:         Yes
  Device has ECC support:                    Enabled
  Device supports Unified Addressing (UVA):   Yes
  Device PCI Domain ID / Bus ID / location ID: 0 / 2 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 7.5, CUDA Runtime Version = 7.5, NumDevs = 1, Device0 = Tesla M2050
Result = PASS
```

Σχήμα 2.1: Πληροφορίες Κάρτας Γραφικών.

Ο κώδικας που χρησιμοποιήθηκε είναι ως ακολουθεί

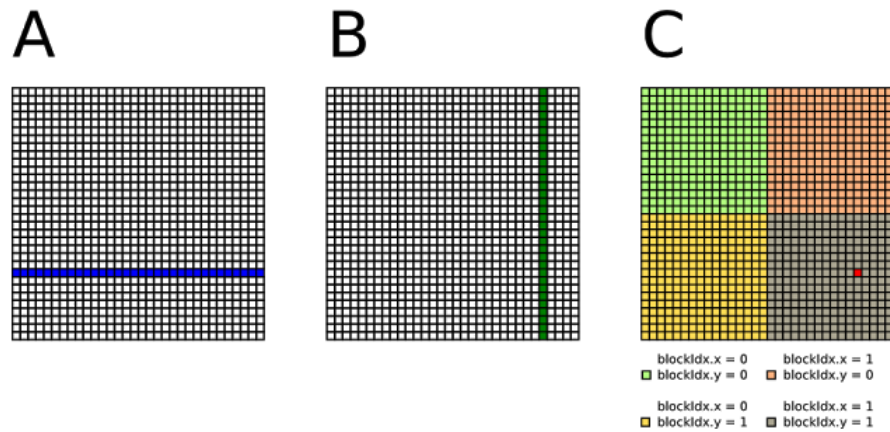
```
1 #!/bin/bash
2
3 #PBS -l nodes=termi1:ppn=1:cuda
4 #PBS -o device_info.out
5 #PBS -e device_info.err
6
7 cd /usr/local/cuda/samples/1_Uutilities/deviceQuery
8 ./deviceQuery
```

## 2.1 Παράλληλη naïve έκδοση της GEMM για GPUs με CUDA

Για την **Naive υλοποίηση**, κάθε νήμα υπολογίζει ένα στοιχείο του  $C$  κάθε φορά. Συγκεκριμένα, κάθε νήμα φορτώνει 1 γραμμή της μήτρας  $A$  και 1 στήλη της  $B$  από την καθολική μνήμη, υπολογίζει το εσωτερικό γινόμενο και το αποθηκεύει στην μήτρα  $C$  της καθολικής μνήμης. Η διαδικασία αυτή επαναλαμβάνεται έως ότου να υπολογιστούν όλα τα στοιχεία της μήτρας  $C$ . Η υλοποίηση αυτή χαρακτηρίζεται ως Naive, δηλαδή ως απλοϊκή λύση, αφού οι γραμμές και οι στήλες των μητρώων  $A, B$  αντίστοιχα φορτώνονται διαρκώς, χωρίς να εκμεταλλεύονται στο έπακρον. Η λύση αυτή, μπορεί να



παράγει το ζητούμενο αποτέλεσμα, όμως μπορεί να βελτιωθεί αρκετά όσον αφορά την “επαναχρησιμοποίηση” της μνήμης.



Σχήμα 2.2: Στιγμιότυπο Πολλαπλασιασμού Naive.

Παρακάτω παρατίθεται ο ζητούμενος κώδικας των τριών συναρτήσεων.

### 2.1.1 dgemm\_naive

```

1  __global__ void dgemm_naive(const double *A, const double *B, double *C,
2                               const int M, const int N, const int K) {
3      int row = blockIdx.y * blockDim.y + threadIdx.y;
4      int col = blockIdx.x * blockDim.x + threadIdx.x;
5
6      if (row < M && col < N) {
7          double sum = 0.;
8          for (int k = 0; k < K; k++)
9              sum += A[row * K + k] * B[k * N + col];
10         C[row * N + col] = sum;
11     }
12 }

```

### 2.1.2 dgemm\_ta\_naive

```

1  __global__ void dgemm_ta_naive(const double *A, const double *B, double *C,
2                                  const int M, const int N, const int K) {
3      int row = blockIdx.y * blockDim.y + threadIdx.y;
4      int col = blockIdx.x * blockDim.x + threadIdx.x;
5
6      if (row < M && col < N) {
7          double sum = 0;
8          for (int k = 0; k < K; k++)
9              sum += A[k * M + row] * B[k * N + col];
10         C[row * N + col] = sum;
11     }
12 }

```

### 2.1.3 dgemm\_tb\_naive

```

1  __global__ void dgemm_tb_naive(const double *A, const double *B, const double *C,
2                                  double *D,

```

```

3         const int M, const int N, const int K) {
4     int row = blockIdx.y * blockDim.y + threadIdx.y;
5     int col = blockIdx.x * blockDim.x + threadIdx.x;
6
7     if (row < M && col < N) {
8         double sum = 0;
9         for (int k = 0; k < K; k++)
10             sum += A[row * K + k] * B[col * K + k];
11         D[row * N + col] = sum + C[row * N + col];
12     }
13 }

```

## 2.1.4 Αποτελέσματα

```

GNU nano 2.0.7 File: run_deepnn_cuda.out
Total time: 180.490957
DGEMM time: 27.626453
DGEMLTA time: 27.581275
DGEMLTB time: 104.937957

GNU nano 2.0.7 File: run_deepnn_cuda.err
Epoch 0 - Elapsed time 18.036801
Epoch 1 - Elapsed time 36.025696
Epoch 2 - Elapsed time 54.028305
Epoch 3 - Elapsed time 71.970773
Epoch 4 - Elapsed time 89.952114
Epoch 5 - Elapsed time 107.889066
Epoch 6 - Elapsed time 125.892890
Epoch 7 - Elapsed time 143.924600
Epoch 8 - Elapsed time 162.340196
Epoch 9 - Elapsed time 180.490920
Correctly predicted 12.17 of the testing set in 10 epochs

```

Σχήμα 2.3: Naive Run (out) - Naive Run (error).

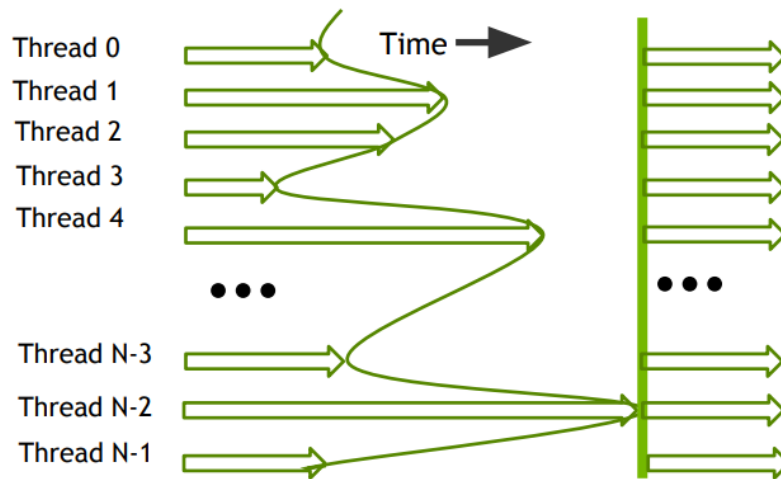
## 2.2 Παράλληλη shmem έκδοση της GEMM για GPUs με CUDA

Προκειμένου να χρησιμοποιηθούν οι συναρτήσεις `*_optimized`, προστίθεται η γραμμή “make GEMM\_OPTIMIZED=1” στο τέλος του αρχείου `compile_on_termis.sh`.

Η παρακάτω λύση κάνει χρήση της τεχνικής **tiling με κοινή μνήμη**. Η κοινή μνήμη αποτελεί μία “ειδική μνήμη”, της οποίας τα δεδομένα προσπελάζονται γρηγορότερα από την καθολική μνήμη και μόνο από τον κώδικα του kernel. Η διάρκεια ζωής της είναι ένα μπλοκ.

Βασική ιδέα πίσω από την υλοποίηση, αποτελεί η ενσωμάτωση ενός επιπλέον επιπέδου μνήμης μεταξύ της καθολικής και των νημάτων ενός μπλοκ. Το επιπλέον επίπεδο είναι ξεχωριστό για κάθε μπλοκ και φορτώνει προσωρινά (όση διάρκεια έχει το μπλοκ) ένα μέρος των πινάκων  $A, B$  από την καθολική μνήμη, τα νήματα του συγκεκριμένου μπλοκ προσπελάζουν μόνο στοιχεία που έχουν φορτωθεί στο νέο επίπεδο, με γρηγορότερη ταχύτητα δίχως να χρειάζεται να “ζητήσουν” κάτι από την καθολική μνήμη. Με αυτόν τον τρόπο, κάθε μπλοκ δουλεύει στην “δικιά” του μνήμη και στο τέλος ενώνει τα αποτελέσματα του με τα υπόλοιπα μπλοκ, ώστε να δημιουργήσει το ζητούμενο αποτέλεσμα.

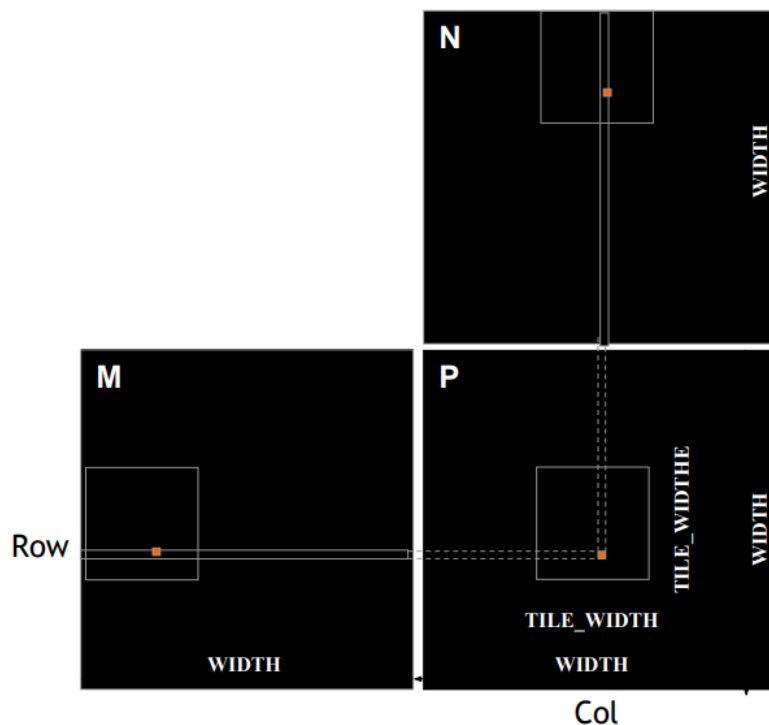
Η παραπάνω πρακτική αποτελεί καλή ιδέα, όταν το πρόγραμμα των νημάτων είναι παρόμοιο. Αντιθέτως, αποτελεί δυσμενές λύση όταν το πρόγραμμα είναι σε μεγάλο βαθμό διαφορετικό, αφού απαιτεί τον συγχρονισμό μεταξύ των νημάτων.



Σχήμα 2.4: Στιγμιότυπο Συγχρονισμού Νημάτων.

Ο υπολογισμός των στοιχείων του πίνακα  $C$ , παραμένει ο ίδιος. Δηλαδή σε κάθε υπο-μπλοκ (tile), υπολογίζεται ο πολλαπλασιασμός μεταξύ της σειράς του πίνακα  $A$  και της στήλης του  $B$ . Με λίγα λόγια, το κάθε υπο-μπλοκ του  $A$ ,  $B$  υπολογίζει ένα υπο-μπλοκ του  $C$ . Εδώ, προσθέτουμε έναν επιπλέον έλεγχο, σε περίπτωση που το υπο-μπλοκ δεν είναι ακέραιο πολλαπλάσιο της διάστασης του Πίνακα που εξετάζει, πρέπει να προστεθούν κελιά με μηδενικά στην αντίστοιχη διάσταση. Τα μηδενικά προστίθενται στο τελευταίο μπλοκ της αντίστοιχης διάστασης της κοινής μνήμης και όχι στον αρχικό πίνακα.

Απαραίτητη προϋπόθεση, να τελειώσουν τους επιμέρους πολλαπλασιασμούς, προτού τοποθετηθεί το τελικό στοιχείο στον πίνακα  $C$ . Συνεπώς, χρειάζεται να ενσωματωθούν 2 έλεγχοι συγχρονισμού των νημάτων (που αποτελούν τους κύριους λόγους επιβράδυνσης της υλοποίησης), έναν έλεγχο κατά την δημιουργία των υπο-μπλοκ της κοινής μνήμης και έναν δεύτερο, ώστε να σιγουρευτεί ότι έχουν φτάσει στο πέρας όλοι οι επιμέρους πολλαπλασιασμοί.



Σχήμα 2.5: Στιγμιότυπο Πολλαπλασιασμού Tiling.

Η παρακάτω υλοποίηση κάνει χρήση δυναμικής κοινής μνήμης, για μέγεθος υπο-μπλοκ  $16 \times 16$ , καθώς παρατηρήσαμε ότι δίνει τους καλύτερους χρόνους. Ωστόσο, επειδή κάναμε κάποιες παραπάνω

μελέτες με χρήση στατικής μνήμης, δηλαδή να γνωρίζει ο compiler εξαρχής το μέγεθος κοινής μνήμης κάθε kernel, παρατηρήσαμε καλύτερους χρόνους. Θα δοθεί ο σχετικός κώδικας σαν παράρτημα του συγκεκριμένου κεφαλαίου (Ενότητα 2.2.5), ενώ ακόμα θα προστεθεί σαν έξτρα παρατήρηση στα τελικά διαγράμματα χρόνων.

Το όφελος του δυναμικό allocate της κοινής μνήμης είναι ότι μπορεί να αλλάξει κατά το runtime. Προκειμένου να πραγματοποιηθεί δυναμική χρήση κοινής μνήμης, δηλώνουμε την τιμή της μεταβλητής **shmem\_size** ως ακολουθεί και την “περνάμε” ως τρίτο όρισμα στον kernel ([docs](#)), στην περίπτωση μας θεωρούμε πάντα.

$$shmem\_size = 2 \times TILE\_DIM \times TILE\_DIM \times sizeof(double)$$

Αφού, στην κοινή μνήμη κάθε μπλοκ υπάρχουν μόνο 2 tiles, τα οποία είναι ουσιαστικά 2 τετράγωνοι υπο-πίνακες μεγέθους  $TILE\_DIM \times TILE\_DIM$  και εμπεριέχουν μεταβλητές τύπου double, λόγω ότι αποθηκεύουν στοιχεία των double πινάκων  $A, B$  της καθολικής μνήμης.

Ενώ από την πλευρά του kernel, πρέπει να χρησιμοποιήσουμε Pointers, προκειμένου να χωρίσουμε στα 2 τους κοινούς υπο-πίνακες, γνωρίζουμε ότι έχουν ίδιο μέγεθος, επομένως συγκεντρωτικά ο κώδικας είναι όπως παρουσιάζεται παρακάτω

## 2.2.1 dgemm\_optimized

### Main

```
1 int TILE_DIM = 16;
2
3 dim3 dimBlock(TILE_DIM, TILE_DIM, 1);
4 dim3 dimGrid((N + dimBlock.x - 1)/dimBlock.x, (M + dimBlock.y - 1)/dimBlock.y);
5 size_t shmem_size = 2 * TILE_DIM * TILE_DIM * sizeof(double);
6
7 dgemm_optimized<<<dimGrid, dimBlock, shmem_size>>>(A, B, C, M, N, K);
8 checkCudaErrors(cudaPeekAtLastError());
9 checkCudaErrors(cudaDeviceSynchronize());
```

### Kernel

```
1 __global__ void dgemm_optimized(const double *A, const double *B, double *C,
2     const int M, const int N, const int K) {
3
4     double CValue = 0;
5
6     extern __shared__ double shared[];
7
8     int tx = threadIdx.x;
9     int ty = threadIdx.y;
10    int dim = blockDim.x;
11
12    int Row = blockIdx.y * dim + ty;
13    int Col = blockIdx.x * dim + tx;
14
15    double* As = (double*)shared;
16    double* Bs = (double*)&shared[dim*dim];
17
18    int ARows = M; int ACols = K;
19    int BRows = K; int BCols = N;
20    int CRows = M; int CCols = N;
21
22    for (int k = 0; k < (dim + ACols - 1)/dim; k++) {
23
24        if (k*dim + tx < ACols && Row < ARows)
```

```

25     As[ty * dim + tx] = A[Row * ACols + k * dim + tx];
26     else
27         As[ty * dim + tx] = 0.0;
28
29     if (k * dim + ty < BRows && Col < BCols)
30         Bs[ty * dim + tx] = B[(k * dim * BCols) + (ty * BCols) + Col];
31     else
32         Bs[ty * dim + tx] = 0.0;
33
34     __syncthreads();
35
36     for (int n = 0; n < dim; n++) {
37         CValue += As[ty * dim + n] * Bs[n * dim + tx];
38     }
39     __syncthreads();
40 }
41
42 if (Row < CRows && Col < CCols) {
43     C[Row*CCols + Col] = CValue;
44 }
45 }

```

## 2.2.2 dgemm\_ta\_optimized

### Main

```

1  int TILE_DIM = 16;
2  dim3 dimBlock(TILE_DIM, TILE_DIM, 1);
3  dim3 dimGrid((N + dimBlock.x - 1)/dimBlock.x, (M + dimBlock.y - 1)/dimBlock.y);
4  size_t shmem_size = 2 * TILE_DIM * TILE_DIM * sizeof(double);
5
6  dgemm_ta_optimized<<<dimGrid, dimBlock, shmem_size>>>(A, B, C, M, N, K);
7  checkCudaErrors(cudaPeekAtLastError());
8  checkCudaErrors(cudaDeviceSynchronize());

```

### Kernel

```

1  __global__ void dgemm_ta_optimized(const double *A, const double *B, double *C,
2      const int M, const int N, const int K) {
3
4      double CValue = 0;
5
6      int tx = threadIdx.x;
7      int ty = threadIdx.y;
8      int dim = blockDim.x;
9
10     int Row = blockIdx.y * dim + ty;
11     int Col = blockIdx.x * dim + tx;
12
13     extern __shared__ double shared[];
14
15     double* As = (double*)shared;
16     double* Bs = (double*)&shared[dim*dim];
17
18     int ARows = M; int ACols = K;
19     int BRows = K; int BCols = N;
20     int CRows = M; int CCols = N;
21
22     for (int k = 0; k < (dim + ACols - 1)/dim; k++) {
23         if (k * dim + tx < ACols && Row < ARows)

```

```

24     As[ty * dim + tx] = A[(k * dim + tx)*ARows + Row];
25     else
26         As[ty * dim + tx] = 0.0;
27
28     if (k * dim + ty < BRows && Col < BCols)
29         Bs[ty * dim + tx] = B[(k * dim + ty)*BCols + Col];
30     else
31         Bs[ty * dim + tx] = 0.0;
32
33     __syncthreads();
34
35     for (int n = 0; n < dim; n++)
36         CValue += As[ty * dim + n] * Bs[n * dim + tx];
37
38     __syncthreads();
39 }
40
41 if (Row < CRows && Col < CCols) {
42     C[Row*CCols + Col] = CValue;
43 }
44 }

```

### 2.2.3 dgemm\_tb\_optimized

#### Main

```

1  int TILE_DIM = 16;
2  dim3 dimBlock(TILE_DIM, TILE_DIM, 1);
3  dim3 dimGrid((N + dimBlock.x - 1)/dimBlock.x, (M + dimBlock.y - 1)/dimBlock.y);
4  size_t shmem_size = 2 * TILE_DIM * TILE_DIM * sizeof(double);
5
6  dgemm_tb_optimized<<<dimGrid, dimBlock, shmem_size>>>(A, B, C, D, M, K, N);
7  checkCudaErrors(cudaPeekAtLastError());
8  checkCudaErrors(cudaDeviceSynchronize());

```

#### Kernel

```

1  __global__ void dgemm_tb_optimized(const double *A, const double *B, const double *C,
2                                     double *D, const size_t M,
3                                     const size_t K, const size_t N) {
4
5     double DValue = 0;
6
7     int tx = threadIdx.x;
8     int ty = threadIdx.y;
9     int dim = blockDim.x;
10
11     int Row = blockIdx.y * dim + ty;
12     int Col = blockIdx.x * dim + tx;
13
14     extern __shared__ double shared[];
15     double* As = (double*)shared;
16     double* Bs = (double*)&shared[dim*dim];
17
18     int ARows = M; int ACols = K;
19     int BRows = K; int BCols = N;
20     int CRows = M; int CCols = N;
21
22     for (int k = 0; k < (dim + ACols - 1)/dim; k++) {

```

```

23     if (k*dim + tx < ACols && Row < ARows)
24         As[ty * dim + tx] = A[Row*ACols + k*dim + tx];
25     else
26         As[ty * dim + tx] = 0.0;
27
28     if (k*dim + ty < BRows && Col < BCols)
29         Bs[ty * dim + tx] = B[k*dim + Col*BRows + ty];
30     else
31         Bs[ty * dim + tx] = 0.0;
32
33     __syncthreads();
34
35     for (int n = 0; n < dim; n++)
36         DValue += As[ty * dim + n] * Bs[n * dim + tx];
37
38     __syncthreads();
39 }
40
41 if (Row < CRows && Col < CCols) {
42     D[Row*CCols + Col] = DValue + C[Row*CCols + Col];
43 }
44 }

```

## 2.2.4 Αποτελέσματα

Σχήμα 2.6: CUDA Dynamic Tiling (out) - CUDA Dynamic Tiling (err).

## 2.2.5 Διαφοροποίηση (Static Shared Memory)

Εδώ παρουσιάζεται συνοπτικά ο κώδικας για την υλοποίηση με στατική μνήμη, το όφελος της οποίας είναι ότι ο compiler γνωρίζει εξ'αρχής το μέγεθος της μνήμης που πρέπει να δεσμεύσει σε κάθε kernel, κάνοντας έτσι αποδοτικότερη κατανομή πόρων.

Αρχικά, ορίζεται η global μεταβλητή TILE\_DIM.

```
1 const int TILE_DIM = 16;
```

Οι πίνακες δηλώνονται ως εξής

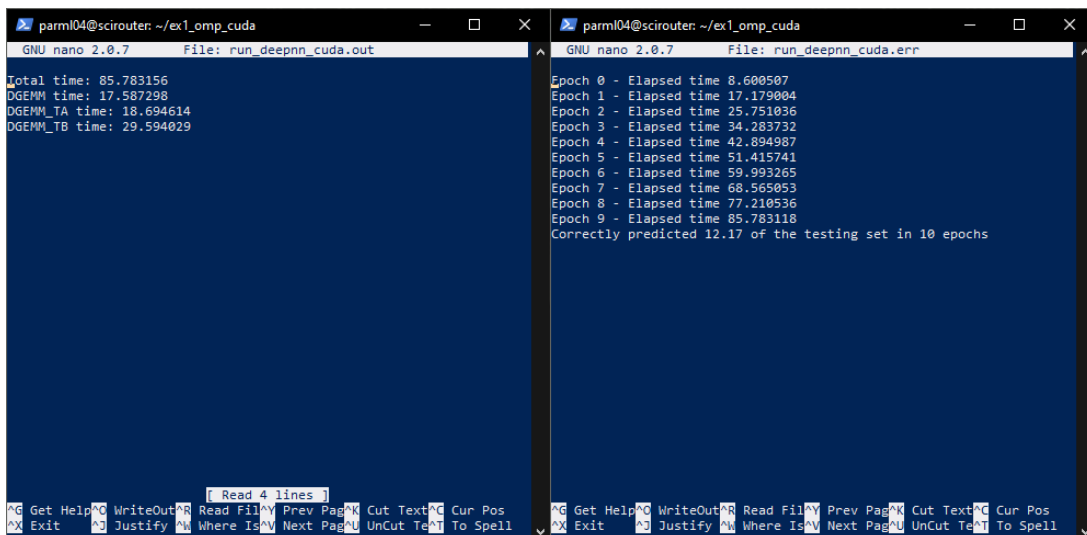
```
1 __shared__ double As[TILE_DIM][TILE_DIM];
2 __shared__ double Bs[TILE_DIM][TILE_DIM];
```

Ενώ, η μόνη διαφοροποίηση είναι κατά την προσπέλαση των στοιχείων (και αυτό διότι τους ορίσαμε ως πίνακες 2 διαστάσεων, σε αντίθεση με 1 διάστασης στην δυναμική υλοποίηση). Προφανώς,

για κάθε τα  $t_a/t_b$ , πρέπει να γίνουν οι αντίστοιχες αλλαγές. Επισυνάπτεται, στιγμιότυπο του κώδικα για το **dgemm\_optimized**.

```
1 // Define Cols, Rows..
2 ...
3
4 // Loop over tiles
5 for (int k = 0; k < (TILE_DIM + ACols - 1)/TILE_DIM; k++) {
6     if (k*TILE_DIM + threadIdx.x < ACols && Row < ARows)
7         As[ty][tx] = A[Row*ACols + k*TILE_DIM + tx];
8     else
9         As[ty][tx] = 0.0;
10    if (k*TILE_DIM + threadIdx.y < BRows && Col < BCols)
11        Bs[ty][tx] = B[(k*TILE_DIM + ty)*BCols + Col];
12    else
13        Bs[ty][tx] = 0.0;
14
15    __syncthreads();
16
17    for (int n = 0; n < TILE_DIM; ++n)
18        CValue += As[ty][n] * Bs[n][tx];
19    __syncthreads();
20 }
21 int index_c = 0;
22 if (Row < CRows && Col < CCols) {
23     index_c = (blockIdx.y * blockDim.y + ty) * CCols + (blockIdx.x * blockDim.x + tx);
24     C[index_c] = CValue;
25 }
26 // End of function
```

Οι σχετικοί χρόνοι παρακάτω



```
parml04@scirouter: ~/ex1_omp_cuda
GNU nano 2.0.7 File: run_deepnn_cuda.out
Total time: 85.783156
DGEWM time: 17.587298
DGEWM_TA time: 18.694614
DGEWM_TB time: 29.594029

parml04@scirouter: ~/ex1_omp_cuda
GNU nano 2.0.7 File: run_deepnn_cuda.err
Epoch 0 - Elapsed time 8.600507
Epoch 1 - Elapsed time 17.179004
Epoch 2 - Elapsed time 25.751036
Epoch 3 - Elapsed time 34.283732
Epoch 4 - Elapsed time 42.894987
Epoch 5 - Elapsed time 51.415741
Epoch 6 - Elapsed time 59.993265
Epoch 7 - Elapsed time 68.565053
Epoch 8 - Elapsed time 77.210536
Epoch 9 - Elapsed time 85.783118
Correctly predicted 12.17 of the testing set in 10 epochs
```

Σχήμα 2.7: CUDA Static Tiling (out) - CUDA Static Tiling (err).

## 2.3 Παράλληλη έκδοση της GEMM για GPUs με cuBLAS

Το cuBLAS γυρνάει column-major μήτρες, που σημαίνει ότι προκειμένου να λάβουμε τα ζητούμενα αποτελέσματα πρέπει πάντα να υπάρχει ένα τελικό transpose. Μπορούμε να χρησιμοποιήσουμε το cuBLAS ώστε να γυρίσει το αποτέλεσμα που επιδιώκουμε, υπολογίζοντας εξ αρχής την σχέση που αν γίνει transposed, θα επιστρέφει το ζητούμενο αποτέλεσμα. Η συνάρτηση dgemm υπολογίζει



$$C = \alpha \cdot op(A) \cdot op(B) + \beta C$$

Το  $op$ , απευθύνεται στα πεδία `transA`, `transB`. Συγκεκριμένα, τα ορίσματα της συνάρτησης ορίζονται ως εξής:

Πίνακας 2.1: Συνάρτηση `cublasDgemm()`.

Παράμετρος	Επεξήγηση
<code>handle</code>	<code>handle</code> για το context του cuBLAS
<code>transa</code>	$A$ ή $A^T$ ή $A^H$ (CUBLAS_OP_N ή CUBLAS_OP_T ή CUBLAS_OP_H)
<code>transb</code>	$B$ ή $B^T$ ή $B^H$ (CUBLAS_OP_N ή CUBLAS_OP_T ή CUBLAS_OP_H)
<code>m</code>	Αριθμός Σειρών των πινάκων A, C
<code>n</code>	Αριθμός Στηλών των πινάκων B, C
<code>k</code>	Αριθμός Στηλών του πίνακα A, και Σειρών του πίνακα B
<code>alpha</code>	Ο συντελεστής $\alpha$
<code>A</code>	Ο πίνακας A
<code>lda</code>	Δεύτερη διάσταση του πίνακα A
<code>B</code>	Ο πίνακας B
<code>ldb</code>	Δεύτερη διάσταση του πίνακα B
<code>beta</code>	Ο συντελεστής $\beta$
<code>ldc</code>	Δεύτερη διάσταση του πίνακα C

Κοινές μεταβλητές για τις DGEMM συναρτήσεις

```
1 const double alpha(1.0);
2 const double beta(0.0);
```

### 2.3.1 dgemm

Ζητείται για  $A_{M \times K}$ ,  $B_{K \times N}$ ,  $C_{M \times N}$  να υπολογιστεί  $C = A \cdot B$ . Ο κώδικας παρακάτω υλοποιεί τον πολλαπλασιασμό

$$C = 1.0 \cdot (B^T \cdot A^T)^T + 0.0 \cdot C$$

```
1 cublasDgemm(dublas_handle, CUBLAS_OP_N, CUBLAS_OP_N
2             N, M, K,
3             &alpha,
4             B, N,
5             A, K,
6             &beta,
7             C, N);
```

Αφού  $(B^T \cdot A^T)^T = (A^T)^T \cdot (B^T)^T = A \cdot B$ .

### 2.3.2 dgemm\_ta

Ζητείται για  $A_{K \times M}$ ,  $B_{K \times N}$ ,  $C_{M \times N}$  να υπολογιστεί  $C = A^T \cdot B$ . Ο κώδικας παρακάτω υλοποιεί τον πολλαπλασιασμό

$$C = 1.0 \cdot (B^T \cdot (A^T)^T)^T + 0.0 \cdot C$$

```

1 cublasDgemm(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_T,
2             N, M, K,
3             &alpha,
4             B, N,
5             A, M,
6             &beta,
7             C, N);

```

Αφού  $(B^T \cdot (A^T)^T)^T = A^T \cdot (B^T)^T = A^T \cdot (B^T)^T = A^T \cdot B$ .

### 2.3.3 dgemm\_tb

Εδώ ζητείται για μήτρες  $A_{M \times K}$ ,  $B_{N \times K}$ ,  $C_{M \times N}$ ,  $D_{M \times N}$ , να υλοποιηθεί ο πολλαπλασιασμός  $D = A \cdot B^T$  και στην συνέχεια η πρόσθεση  $D = C + D$ . Αρχικά θα υλοποιήσουμε τον πολλαπλασιασμό

$$D = 1.0 \cdot ((B^T)^T \cdot A^T)^T + 0.0 \cdot D$$

```

1 cublasDgemm(cublas_handle, CUBLAS_OP_T, CUBLAS_OP_N,
2             N, M, K,
3             &alpha,
4             B, K,
5             A, K,
6             &beta,
7             D, N);

```

Αφού  $((B^T)^T \cdot A^T)^T = (A^T)^T \cdot B^T = A \cdot B^T$ . Ενώ χρησιμοποιώντας την συνάρτηση `dgeam` που λειτουργεί με την ίδια λογική της `dgemm`, προσθέτουμε τον πίνακα  $C$  για το τελικό αποτέλεσμα.

$$D = 1.0 \cdot D + 1.0 \cdot C$$

```

1 cublasDgeam(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_N,
2             N, M,
3             &alpha,
4             D, N,
5             &alpha,
6             C, N,
7             D, N);

```

### 2.3.4 Αποτελέσματα

```

GNU nano 2.0.7 File: run_deepnn_cublas.out
Total time: 65.966698
DGEAM time: 14.182593
DGEAM_TA time: 10.127328
DGEAM_TB time: 21.393733

GNU nano 2.0.7 File: run_deepnn_cublas.err
Epoch 0 - Elapsed time 6.642789
Epoch 1 - Elapsed time 13.254928
Epoch 2 - Elapsed time 19.874082
Epoch 3 - Elapsed time 26.448240
Epoch 4 - Elapsed time 33.046571
Epoch 5 - Elapsed time 39.627402
Epoch 6 - Elapsed time 46.218438
Epoch 7 - Elapsed time 52.792543
Epoch 8 - Elapsed time 59.371701
Epoch 9 - Elapsed time 65.966653
Correctly predicted 12.17 of the testing set in 10 epochs

```

Σχήμα 2.8: cuBlas (out) - cuBLAS (err).

# Πειράματα και μετρήσεις επιδόσεων

Συγκεντρωτικά οι χρόνοι κάθε πειράματος περιγράφονται στους παρακάτω πίνακες.

Πίνακας 3.1: Χρόνοι cBLAS.

Αρ. Νήμ.	1	2	4	6	12	24
Συνάρτ.						
DGEMM	67.124818	35.914273	19.522483	17.623101	12.219146	11.418573
DGEMM_TA	83.441351	45.970665	26.294149	21.042938	15.923911	17.491970
DGEMM_TB	79.155355	43.215657	23.431148	21.645427	15.110114	14.459852
Συνολικά	244.790734	142.913517	88.426972	81.037994	64.800990	75.131975

Πίνακας 3.2: Χρόνοι OpenMP.

Αρ. Νήμ.	1	2	4	6	12	24
Συνάρτ.						
DGEMM	633.863073	334.898033	248.942804	241.143680	120.94996	250.855133
DGEMM_TA	1084.001369	568.593181	304.852508	232.876213	141.13806	143.376206
DGEMM_TB	397.158843	204.558644	105.057498	70.512212	44.201241	40.803859
Συνολικά	2131.281783	1124.801846	676.457929	563.137857	331.254268	466.794536

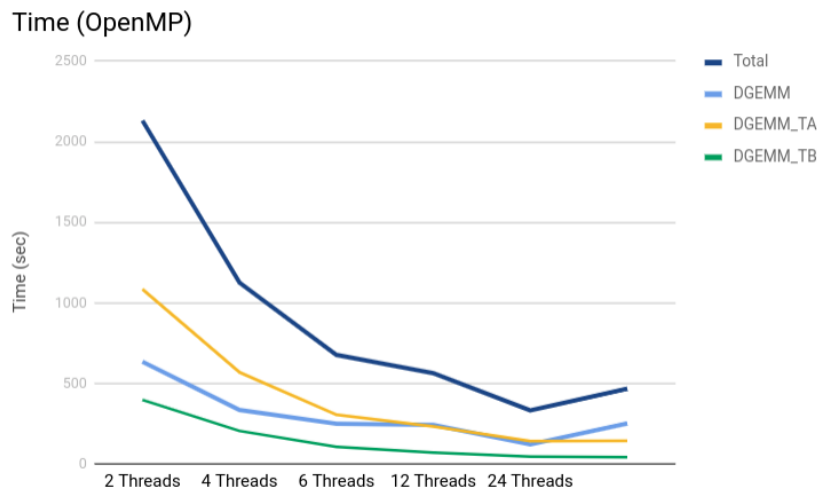
Ενώ, οι χρόνοι για naive, optimized (dynamic/static) και cuBLAS απεικονίζονται στον παρακάτω πίνακα

Πίνακας 3.3: Χρόνοι CUDA.

Υλοποίηση	Naive	Optimized Dynamic	Optimized Static	cuBLAS
Συνάρτ.				
DGEMM	27.626453	19.950643	17.587298	14.182593
DGEMM_TA	27.501275	21.222518	18.694614	10.127328
DGEMM_TB	104.937957	32.318991	29.594029	21.393733
Συνολικά	180.490957	93.210933	85.783156	65.966698

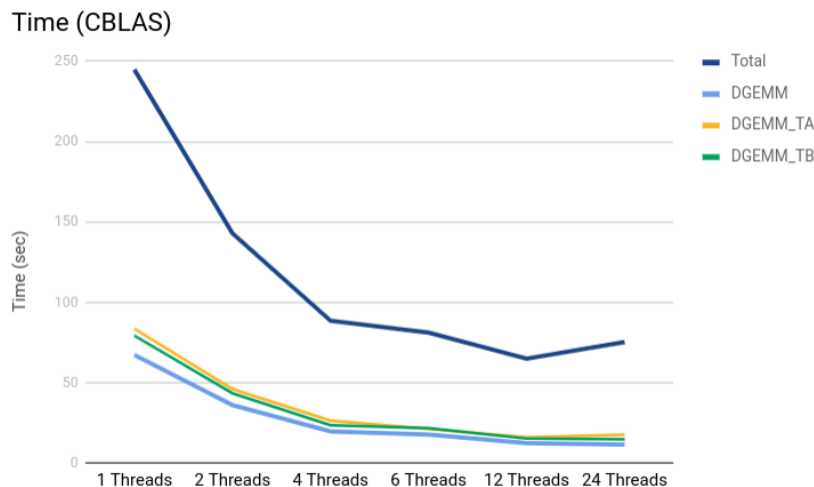
### 3.1 Κλιμακωσιμότητα σε CPU

Στο διάγραμμα που ακολουθεί παρατηρούμε τους συνολικούς χρόνους εκτέλεσης καθώς μεταβάλλεται ο αριθμός των threads με την χρήση του OpenMP. Όπως είναι λογικό όσο αυξάνεται ο αριθμός των threads τόσο μειώνεται και ο χρόνος εκτέλεσης μέχρι τον αριθμό των 24 threads. Αυτό μπορεί να συμβαίνει για τους λόγους ότι: 1) δεν υπάρχει ικανή παραλληλοποίηση στον κώδικα, με αποτέλεσμα το κομμάτι που είναι σειριακό να μη επιδέχεται βελτιστοποίηση ή 2) εξαιτίας των πολλών threads και η ανάγκη για συγχρονισμό αυτών σε πολλά σημεία του κώδικα να δημιουργεί επιπλέον overhead.



Σχήμα 3.1: OpenMP time.

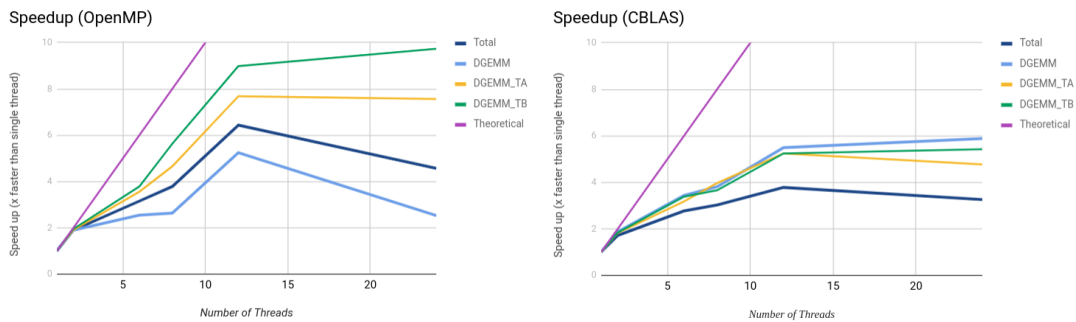
Στο παρακάτω διάγραμμα παρατηρούμε την συγκριτικά καλύτερη υλοποίηση της έτοιμης συνάρτησης cblas του OpenBlas σε σχέση με την απλή υλοποίηση του OpenMP. Αυτό συμβαίνει διότι στο CBLAS, χρησιμοποιείται μια υπορουτίνα της FORTRAN, η οποία είναι βελτιστοποιημένη από πολλές απόψεις (compiled με optimization flags, αξιοποιεί κοντινές θέσεις μνήμης για ταχύτερη προσπέλαση δεδομένων -χρήση CblasRowMajor-). Γίνεται φανερό λοιπόν, ότι όσους πόρους και να έχουμε, εάν ο κώδικας δεν είναι σωστά δομημένος, τα κέρδη θα είναι λίγα, και αυτό φαίνεται κατά κανόνα όταν το 24-thread του OpenMP, πάει πιο αργά από το single-thread του OpenBLAS.



Σχήμα 3.2: Cblas time.

Εάν δούμε τα διαγράμματα του speedup, δηλαδή του χρόνου της σειριακή εκτέλεση προς τον χρόνο της παραλληλοποιημένης εκτέλεσης του κώδικα, βλέπουμε ότι απέχει από το θεωρητικό πράγμα που σημαίνει ότι διπλάσιοι πυρήνες δεν μειώνουν τον χρόνο εκτέλεσης στο μισό, ο λόγος όπως είπαμε και παραπάνω είναι ότι οι κώδικες έχουν overhead αλλά και μη παράλληλα κομμάτια τα οποία τους καθυστερούν.

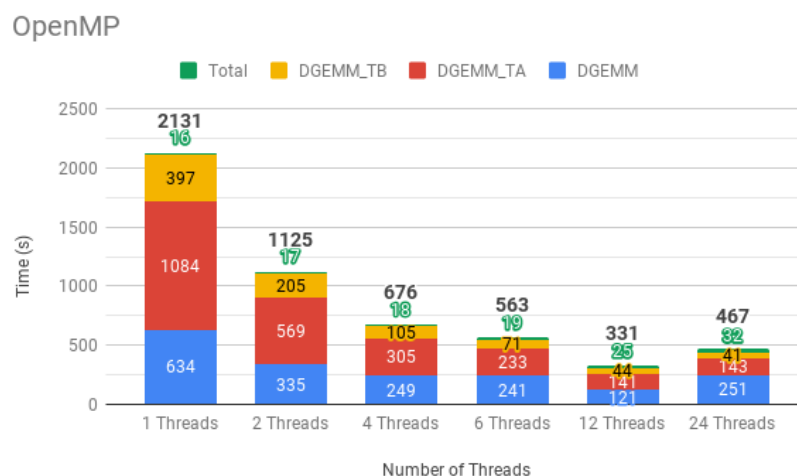
Ενά ακόμα ενδιαφέρον στοιχείο είναι στην χρήση συνάρτηση του cblas όπου το speedup είναι μικρότερο από ότι στην υλοποίησή την απλή του OpenMP αυτό συμβαίνει διότι οι βελτιστοποιημένες υλοποιήσεις του cblas δεν είναι τόσο ευαίσθητες στην αύξηση των threads. Από την άλλη η μη βελτιστη υλοποίηση στο OpenMP είναι λογικό να δίνει καλύτερο speedup αφού τα περιθώρια κέρδους σε χρόνο είναι μεγαλύτερα.



Σχήμα 3.3: OpenMP-cBLAS Speedup

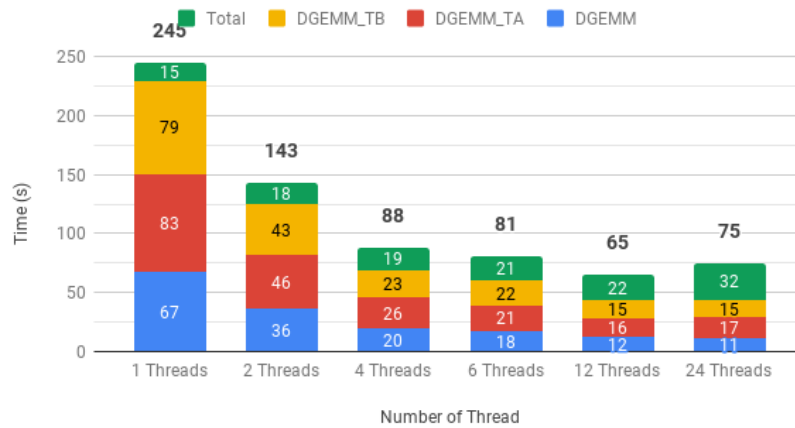
## 3.2 Σύγκριση επιδόσεων σε CPU και GPU

Παρακάτω παρουσιάζονται barplots για τις περιπτώσεις CPU (OpenMP, cBLAS), GPU (Naive, Optimized, cuBLAS). Σε κάθε διάγραμμα απεικονίζεται με πράσινο χρώμα ο συνολικός χρόνος, με μπλε ο χρόνος εκτέλεσης της `dgemm`, με κόκκινο χρώμα ο χρόνος εκτέλεσης της `dgemm_ta` και τέλος με κίτρινο χρώμα ο χρόνος εκτέλεσης της `dgemm_tb`. Προσπαθήσαμε να συμπεριλάβουμε τον τελικό χρόνο (πάνω από κάθε μπάρα) καθώς και τους επιμέρους χρόνους μέσα στα γραφήματα. Σαν τιμή της διακριτής πράσινης μπάρας ορίζεται ο συνολικός χρόνος αφαιρώντας τους 3 επιμέρους. Αυτό μας βοηθάει να κατανοήσουμε ακριβώς, πόσος χρόνος χρησιμοποιήθηκε από τις συναρτήσεις και πόσος χρόνος “χάθηκε” από διεργασίες του συστήματος.



Σχήμα 3.4: OpenMP barplot.

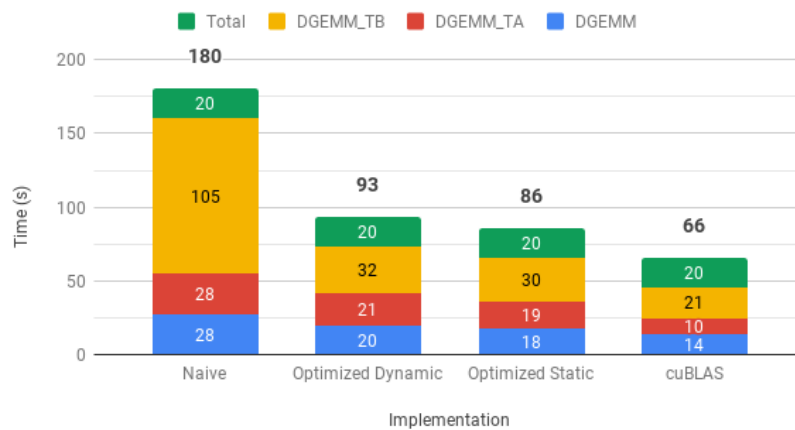
cBLAS



Σχήμα 3.5: cBlas Barplot.

Στο κομμάτι του Επεξεργαστή, παρατηρούμε διαρκή μείωση του απαιτούμενου χρόνου, μέχρι τα 12 νήματα. Μετά, για την 24 νημάτα φαίνεται να αυξάνεται ο χρόνος και στις δύο περιπτώσεις (OpenMP, cBLAS). Ωστόσο, για το OpenMP, κύριος παράγοντας αύξησης του συνολικού χρόνου είναι ο χρόνος περάτωσης την συνάρτησης `dgemm`, η οποία αυξάνεται κατά 130 δευτερόλεπτα. Εν αντιθέσει, στην περίπτωση του cBLAS, οι χρόνοι περάτωσης των συναρτήσεων παραμένουν σταθεροί, ενώ παρατηρούμε ότι αυξάνεται ο συνολικός χρόνος. Γεγονός που οδηγεί στο συμπέρασμα που αναφέρθηκε παραπάνω, δηλαδή ότι το σύστημα απαιτεί παραπάνω χρόνο προκειμένου να κατανείμει, συγχρονίσει, συλλέξει τις δουλειές των νημάτων.

Naive, Optimized and cuBLAS



Σχήμα 3.6: GPU Barplot.

Όπως παρατηρείται από τα διαγράμματα της κάρτας γραφικών, η Naive υλοποίηση πετυχαίνει τους χειρότερους χρόνους, ακολουθούμενη από την optimized και την καλύτερη θέση κατέχει η υλοποίηση της cuBLAS. Κάνοντας χρήση της κοινής μνήμης (optimized), πετυχαίνουμε περίπου 50% μικρότερο χρόνο εκπαίδευσης. Ενώ, χρησιμοποιώντας τις συναρτήσεις της βιβλιοθήκης cuBLAS που είναι περαιτέρω βελτιστοποιημένες, ο χρόνος βελτιώνεται περίπου 25% ακόμα.

Σε όλες τις περιπτώσεις για GPU παρατηρούμε ότι η συνάρτηση `DGEMM_TB` απαιτεί τον περισσότερο χρόνο για να τρέξει, λογικό αφού υλοποιεί τους περισσότερους υπολογισμούς (εσωτερικό γινόμενο και πρόσθεση). Τέλος, οι χρόνοι των συναρτήσεων `DGEMM` και `DGEMM_TA` είναι σχεδόν παντού παρόμοιοι, αφού υλοποιούν την ίδια πράξη (εσωτερικό γινόμενο), με την διαφορά ότι η `DGEMM_TA` φορτώνει τον ανάστροφο του  $A$ . Ο υπολογισμός του ανάστροφου, δεν εμπεριέχει κάποιο

---

penalty αφού, φορτώνεται κατευθείαν από την μνήμη όπως η κανονική έκδοση του πίνακα.

**Συμπερασματικά**, μπορούμε να πούμε ότι πετυχαίνουμε βέλτιστους χρόνους με την χρήση της βιβλιοθήκης BLAS, είτε για CPU, είτε για GPU. Στην πλευρά των CPU, απαιτούνται 12 νήματα για να φτάσουν τις επιδόσεις των GPU. Βέβαια, τα αποτελέσματα μας ενδέχεται να αλλάξουν για πράξεις μεταξύ μικρότερων ή μεγαλύτερων πινάκων. Όταν οι πίνακες έχουν μικρό μέγεθος, συνιστάται οι πράξεις να γίνονται στον επεξεργαστή καθώς τα κόστη αντιγραφής, συγχρονισμού είναι σημαντικά σε σχέση με τον συνολικό χρόνο περάτωσης. Εν αντιθέσει, όταν βρισκόμαστε αντιμέτωποι με μεγάλους πίνακες προτιμάται η χρήση της κάρτας γραφικών για τον υπολογισμό αυτών.