

# Παράλληλα και Διανεμημένα Συστήματα Εργασία 3 .

## CUDA Ising Model

Μιχάλης Καρατζάς

AEM:9137

email:[mikalaki@ece.auth.gr](mailto:mikalaki@ece.auth.gr)

Ομάδα στο elearning: Ομάδα M

Github link για το project :[https://github.com/mikalaki/Ising\\_model\\_CUDA](https://github.com/mikalaki/Ising_model_CUDA)

### 1. Σύντομη Περιγραφή των υλοποιήσεων.

- **V0-Sequential:** Η Ακολουθιακή υλοποίηση του προγράμματος στην C, υλοποιεί την εξέλιξη του Ising model , με τις τροποποιήσεις που αναφέρθηκαν και με το template της συνάρτησης `ising()` που μας δόθηκε στον online tester. Συγκεκριμένα , η συνάρτηση `ising()`, καλεί την `nextStateCalculation()` , τόσες φορές όσες και τα βήματα στα οποία θέλουμε να εξελίσσουμε το ising μοντέλο. Αυτή η συνάρτηση , για κάθε ένα σημείο του πλέγματος(μέσω for επαναλήψεων) καλεί την `getTheSpin()`, η οποία δίνει την ροπή (spin) του σημείου στην αμέσως επόμενη κατάσταση του φερομαγνητικού υλικού , με βάση την κατάσταση των γειτόνων του, στο δεδομένο πλέγμα και το εκχωρεί στον δεύτερο πίνακα G (secondG). Αφού ολοκληρωθεί αυτή η διαδικασία, η συνάρτηση `ising()` κάνει την αλλαγή των `pointer` όπως μας ζητήθηκε, μέσω της `pointer_swap()` ενώ σε περίπτωση περιττού αριθμού σημείων , γίνεται αντιγραφή ( `memcpy()`) των σημείων του δευτέρου πίνακα G στον πρώτο πίνακα, ώστε ο πρώτος(δοθέν) πίνακας να είναι ενημερωμένος με τα τελευταία σημεία της εξέλιξης του μοντέλου. Για τις οριακές συνθήκες του πλέγματος, λόγω περιοδικών ιδιοτήτων, χρησιμοποιείται αριθμητική με `modulus`. (προστίθεται +n , πριν υπολογιστεί το `modulo`, καθώς το `modulo` στην C ταυτίζεται με το υπόλοιπο της διαίρεσης και δίνει και αρνητικές τιμές).
- **V1:** Στην πρώτη υλοποίηση με CUDA, μας ζητήθηκε ένα `thread` να βρίσκει την ροπή (spin) ενός σημείου(1 `thread` για 1 σημείο). Έτσι τροποποιήθηκε ο σειριακός κώδικας , ώστε να γίνουν οι απαραίτητες δεσμεύσεις και "αντιγραφές- μεταφορές" στην μνήμη της GPU, ώστε να μπορεί να υλοποιηθεί το πρόγραμμα μας αξιοποιώντας την κάρτα γραφικών .Επίσης η μέθοδος `nextStateCalculation()` , καλείται σαν `kernel` με ένα `grid` από `threads`, συνολικών διαστάσεων  $(gridDim.x*blockDim.x \geq n \ \& \ gridDim.y*blockDim.y \geq n)$  όπου η μια πλευρά του δοθέν πλέγματος. Η συνάρτηση αυτή , αφού ελέγξει ώστε να μην "εκτελούν" την `getTheSpin()` `threads` που "περισσεύουν", καλεί την `__device__ getTheSpin()` , ώστε κάθε `thread` να υπολογίσει την ροπή του σημείου που του αντιστοιχεί. Ενώ πλέον η `pointer_swap()` μέσα στην `ising()`, αλλάζει τους `pointers` για τους αντίστοιχους πίνακες της κάρτας γραφικών (`device`). \* \*
- **V2:** Σε αυτή την υλοποίηση, σε σχέση με την παραπάνω , ο κώδικας του `kernel nextStateCalculation()`, τροποποιείται έτσι ώστε ένα `thread` να υπολογίζει το `spin` για ένα σύνολο από σημεία , το οποίο εξαρτάται τις παραμέτρους του `grid`. Τα σημεία αυτά απέχουν κατά τον X και τον Y άξονα απόσταση ίση με `strideX` και `strideY`, αντίστοιχα , η οποία υπολογίζεται στην παραπάνω μέθοδο και εξαρτάται από τις παραμέτρους του `grid` . Οι διαστάσεις του `grid` και των `block` του , επιλέγονται από τα `#define` που υπάρχουν στο πάνω μέρος του κώδικα . Ενώ οι τιμές που έχουν επιλεγεί δίνουν βέλτιστη απόδοση.
- **V3:** Η συγκεκριμένη υλοποίηση βασίζεται στην V2 (οπότε στην δομή της ισχύει ότι και για την παραπάνω) ωστόσο έχει τροποποιηθεί , έτσι ώστε τα `thread` ενός `block` , για τον υπολογισμό του spin του σημείου που υπολογίζουν , να μην χρειάζεται να διαβάζουν πληροφορία από τον πίνακα G από την *ολική μνήμη(global memory)* της κάρτας γραφικών, αλλά από ένα τμήμα του G (μόνο το απαραίτητο σύνολο σημείων), που έχει μεταφερθεί στην *shared memory* του μπλοκ από τα ίδια τα `thread` του `block` στην αρχή του `kernel nextStateCalculation()`, με στόχο την επίτευξη γρηγορότερης επίδοσης. Επίσης για `blockDim.x>5` και `blockDim.y>5`, μεταφέρεται και ο πίνακας των βαρών (`weights`), δίνοντας μια επιπλέον πολύ μικρή βελτίωση στον χρόνο εκτέλεσης του προγράμματος (σχεδόν αμελητέα) .

Περισσότερες λεπτομέρειες σχετικά με τον κώδικα και κάθε υλοποίηση υπάρχουν στα σχόλια του κώδικα.

### 2. Μετρήσεις, γραφήματα, σχόλια χρόνων εκτέλεσης και συμπεράσματα.

#### I. Το σύστημα στο οποίο πραγματοποιήθηκαν υλοποίηση και μετρήσεις:

Η υλοποίηση και οι μετρήσεις έγιναν σε ένα laptop με επεξεργαστή i5-2410M και 4 GB RAM , το οποίο έχει κάρτα γραφικών NVIDIA gt 540m με μνήμη 2GB. Η κάρτα έχει Compute Capability (CC) 2.1 , με αποτέλεσμα να μην είναι εγκατεστημένη στο σύστημα η τελευταία έκδοση CUDA (10.2) , άλλα η CUDA 8 και λόγω CC του υλικού , δεν ήταν εφικτό να χρησιμοποιηθεί Unified Memory (π.χ. `cudaMallocManaged()` ). Η κάρτα διαθέτει 2 Multiprocessors(MPs) με 48 cuda threads ο καθένας. Σε κάθε MP μπορούν να "τρέχουν" έως 1536 thread. Περισσότερες πληροφορίες για την κάρτα γραφικών του συστήματος βρίσκονται [εδώ](#).

#### II. A) Επιλογή των διαστάσεων των `block` του V1(μετρήσεις V1 για διαφορά `block size`)

Οι μετρήσεις για την επιλογή του μεγέθους του *block* στην V1 έγιναν για  $n=1000$  και  $k=20$  μέσω του *profiler* της Nvidia για CUDA ( *nvprof*) και μετρήθηκε ο χρόνος εκτέλεσης του *kernel* ( *nextStateCalculation()*). Παρακάτω οι μετρήσεις :

| Block Size | V1 kernel – Time | Calls | ( avg     | min      | max) --   |
|------------|------------------|-------|-----------|----------|-----------|
| 1x1        | 21.8176s         | 20    | (1.09088s | 1.09082s | 1.09099s) |
| 2x2        | 5.73757s         | 20    | (286.88ms | 286.73ms | 287.38ms) |
| 4x4        | 1.62129s         | 20    | (81.065ms | 80.999ms | 81.237ms) |
| 8x8        | 558.15ms         | 20    | (27.908ms | 27.865ms | 27.956ms) |
| 12x12      | 472.54ms         | 20    | (23.627ms | 23.578ms | 23.669ms) |
| 16x16      | 469.43ms         | 20    | (23.471ms | 23.396ms | 23.538ms) |
| 20x20      | 544.69ms         | 20    | (27.234ms | 27.187ms | 27.289ms) |
| 24x24      | 618.80ms         | 20    | (30.940ms | 30.872ms | 31.002ms) |
| 28x28      | 721.74ms         | 20    | (36.087ms | 36.064ms | 36.121ms) |
| 32x32      | 740.46ms         | 20    | (37.023ms | 36.994ms | 37.042ms) |

Από τις μετρήσεις αυτές βλέπουμε ότι στο V1 έχουμε βέλτιστη επίδοση για *block* με διαστάσεις 16x16 , επομένως θα θέσουμε αυτές τις διαστάσεις για τις μετρήσεις τις ενότητας III (ενδέχεται για πρόβλημα με διαφορετικό  $n$  , καθώς οι διαστάσεις του *grid* εξαρτώνται από το  $n$  , να μην είναι βέλτιστο αυτό το *configuration*) .

Πίνακας 1:Χρόνοι εκτέλεσης υλοποίησης V1,μεταβαλλόμενου *block size* , $n=1000,k=20$ ).

### B) Επιλογή των διαστάσεων του *Grid* και των *block* για τα V2,V3.

Οι διαστάσεις για το *Grid* και τα *block* επιλέχθηκαν έπειτα από μετρήσεις του χρόνου που απαιτεί το *kernel*( *nextStateCalculation()*) να εκτελεστεί στις υλοποιήσεις V2 και V3 , για ένα σύνολο διαστάσεων των *block* και του *grid*. Οι μετρήσεις έγιναν για  $n=1000$  και  $k=20$  μέσω του *profiler* της Nvidia για CUDA ( *nvprof*) . Παρακάτω οι μετρήσεις:

| Grid Size | Block Size | V2 kernel – Time | Calls | ( avg      | min      | max) --   | V3 kernel- Time | Calls | ( avg     | min      | max) --   |
|-----------|------------|------------------|-------|------------|----------|-----------|-----------------|-------|-----------|----------|-----------|
| 4x4       | 2x2        | 5.26403s         | 20    | ( 263.20ms | 262.87ms | 263.59ms) | 2.63122s        | 20    | (131.56ms | 130.78ms | 132.47ms) |
| 4x4       | 4x4        | 1.54508s         | 20    | (77.254ms  | 75.560ms | 78.823ms) | 709.48ms        | 20    | (35.474ms | 35.407ms | 35.688ms) |
| 4x4       | 8x8        | 480.02ms         | 20    | ( 24.001ms | 23.761ms | 24.353ms) | 252.44ms        | 20    | (12.622ms | 12.586ms | 12.658ms) |
| 4x4       | 12x12      | 439.43ms         | 20    | ( 21.971ms | 21.944ms | 22.016ms) | 226.08ms        | 20    | (11.304ms | 11.268ms | 11.341ms) |
| 4x4       | 16x16      | 369.01ms         | 20    | ( 18.451ms | 18.400ms | 18.500ms) | 198.69ms        | 20    | (9.9343ms | 9.9202ms | 9.9518ms) |
| 4x4       | 20x20      | 369.52ms         | 20    | ( 18.476ms | 18.454ms | 18.502ms) | 202.81ms        | 20    | (10.140ms | 10.126ms | 10.154ms) |
| 4x4       | 24x24      | 347.42ms         | 20    | ( 17.371ms | 17.320ms | 17.389ms) | 196.05ms        | 20    | (9.8024ms | 9.7665ms | 9.8429ms) |
| 4x4       | 28x28      | 378.79ms         | 20    | ( 18.939ms | 18.916ms | 18.959ms) | 228.46ms        | 20    | (11.423ms | 11.406ms | 11.439ms) |
| 4x4       | 32x32      | 352.28ms         | 20    | ( 17.614ms | 17.594ms | 17.625ms) | 209.67ms        | 20    | (10.484ms | 10.453ms | 10.532ms) |
| 1x1       | 24x24      | 684.44ms         | 20    | ( 34.222ms | 34.183ms | 34.259ms) | 397.88ms        | 20    | (19.894ms | 19.808ms | 19.968ms) |
| 2x2       | 24x24      | 362.27ms         | 20    | ( 18.114ms | 18.089ms | 18.128ms) | 217.87ms        | 20    | (10.894ms | 10.866ms | 10.909ms) |
| 4x4       | 24x24      | 347.42ms         | 20    | ( 17.371ms | 17.320ms | 17.389ms) | 196.05ms        | 20    | (9.8024ms | 9.7665ms | 9.8429ms) |
| 6x6       | 24x24      | 353.75ms         | 20    | ( 17.687ms | 17.672ms | 17.698ms) | 196.78ms        | 20    | (9.8392ms | 9.7757ms | 9.8609ms) |
| 8x8       | 24x24      | 356.72ms         | 20    | ( 17.836ms | 17.824ms | 17.845ms) | 217.32ms        | 20    | (10.866ms | 10.853ms | 10.878ms) |
| 12x12     | 24x24      | 361.29ms         | 20    | ( 18.065ms | 18.044ms | 18.084ms) | 212.12ms        | 20    | (10.606ms | 10.583ms | 10.635ms) |
| 16x16     | 24x24      | 364.44ms         | 20    | ( 18.222ms | 18.198ms | 18.235ms) | 198.66ms        | 20    | (9.9331ms | 9.8995ms | 9.9456ms) |
| 20x20     | 24x24      | 370.92ms         | 20    | ( 18.546ms | 18.530ms | 18.557ms) | 222.81ms        | 20    | (11.140ms | 11.114ms | 11.151ms) |

πίνακας 2:Χρόνοι εκτέλεσης υλοποιήσεων V2,V3 ,μεταβαλλόμενων *block size* και *grid size* ( $n=1000$  ,  $k=20$ ).

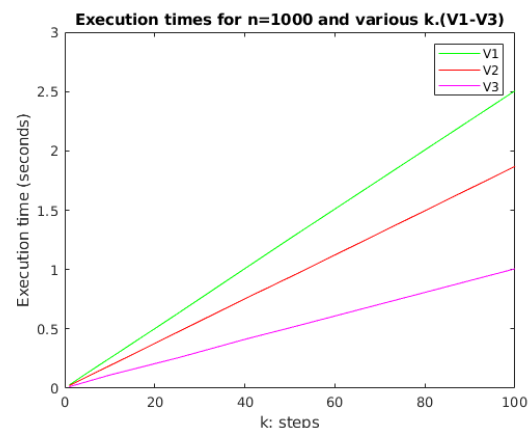
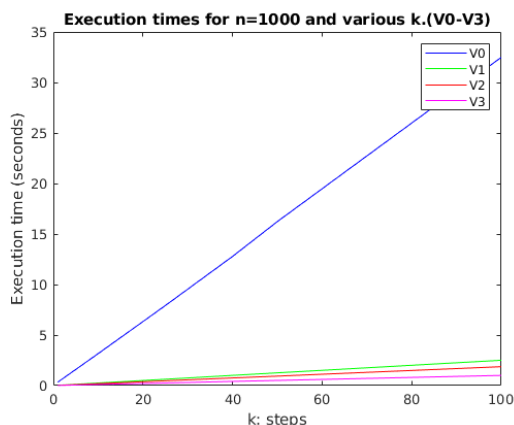
Από τις παραπάνω μετρήσεις εύκολα μπορούμε να καταλάβουμε ότι έχουμε βέλτιστη απόδοση για *block size* 24x24 και *grid size* 4x4. Επίσης βλέπουμε ότι οι διαστάσεις του *block* παίζουν σημαντικότερο ρόλο από αυτές του *grid*. Το παραπάνω *Configuration* χρησιμοποιείται παρακάτω .

--> Λόγω τετράγωνης συμμετρίας του προβλήματος επιλέξαμε τετράγωνου σχήματος *block* και *grid*.

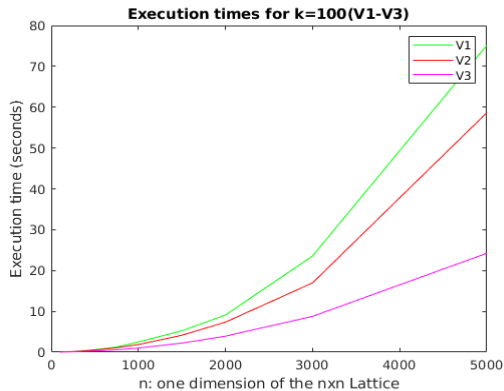
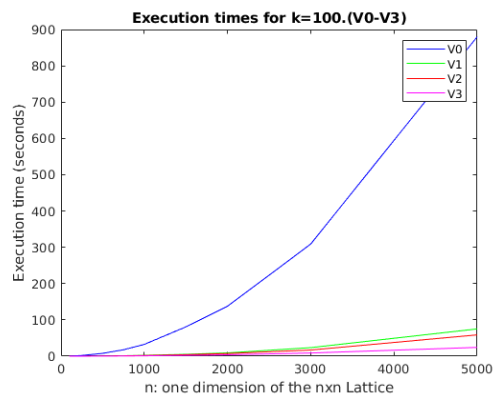
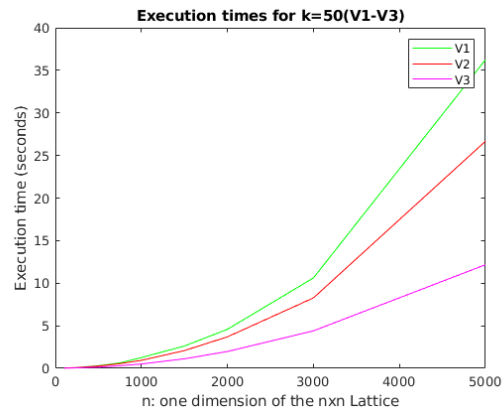
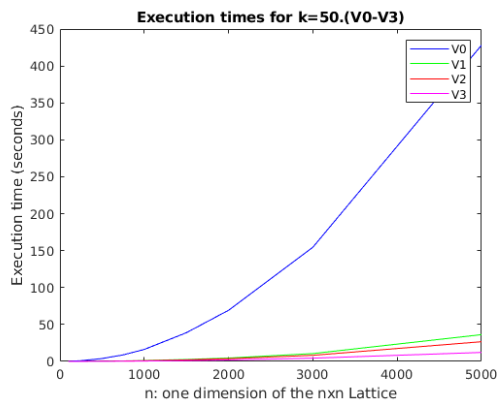
### III.Μετρήσεις χρόνου εκτέλεσης κάθε υλοποίησης για διάφορα $n$ και $k$ .

Στις μετρήσεις παρακάτω , υπολογίζεται ο συνολικός χρόνος εκτέλεσης του προγράμματος (συνάρτηση *ising()* ) και όχι μόνο , ο χρόνος της συνάρτησης (*kernel*) *nextStateCalculation()* ,καθώς μας αφορά η λύση του συνολικού προβλήματος.

#### A) Γραφήματα για σταθερό $n$ και μεταβαλλόμενο $k$ .



#### B) Γραφήματα για σταθερό $k$ , μεταβαλλόμενο $n$ .



### C) Πίνακες μετρήσεων:

| n   | k   | V3       | V2       | V1       | V0       |
|-----|-----|----------|----------|----------|----------|
| 100 | 100 | 0.033891 | 0.057151 | 0.041795 | 0.3153   |
| 100 | 50  | 0.017193 | 0.032012 | 0.026332 | 0.158297 |
| 100 | 40  | 0.022354 | 0.030095 | 0.026814 | 0.134378 |
| 100 | 25  | 0.00673  | 0.011818 | 0.009053 | 0.082458 |
| 100 | 10  | 0.002649 | 0.00542  | 0.003874 | 0.040409 |
| 100 | 5   | 0.001794 | 0.004217 | 0.002398 | 0.020819 |

| n    | k   | V3       | V2       | V1       | V0        |
|------|-----|----------|----------|----------|-----------|
| 1000 | 100 | 1.006457 | 1.869143 | 2.506302 | 32.487474 |
| 1000 | 50  | 0.509197 | 0.936039 | 1.260833 | 16.227674 |
| 1000 | 40  | 0.411954 | 0.753887 | 1.008276 | 12.782729 |
| 1000 | 25  | 0.254887 | 0.471384 | 0.627108 | 7.92002   |
| 1000 | 10  | 0.111413 | 0.190261 | 0.252865 | 3.161162  |
| 1000 | 5   | 0.056263 | 0.097338 | 0.12896  | 1.594185  |
| 1000 | 1   | 0.01244  | 0.023128 | 0.027221 | 0.343522  |

| n   | k   | V3       | V2       | V1       | V0       |
|-----|-----|----------|----------|----------|----------|
| 500 | 100 | 0.278743 | 0.51884  | 0.602692 | 7.815543 |
| 500 | 50  | 0.144472 | 0.264958 | 0.306216 | 3.916218 |
| 500 | 40  | 0.121067 | 0.221278 | 0.253803 | 3.136734 |
| 500 | 25  | 0.078536 | 0.141124 | 0.160674 | 1.958934 |
| 500 | 10  | 0.031029 | 0.059899 | 0.073745 | 0.786562 |
| 500 | 5   | 0.01988  | 0.029453 | 0.038279 | 0.396289 |
| 500 | 1   | 0.003871 | 0.009502 | 0.00731  | 0.081865 |

| n    | k   | V3        | V2        | V1        | V0        |
|------|-----|-----------|-----------|-----------|-----------|
| 5000 | 100 | 24.216919 | 58.645838 | 75.134646 | 879.95927 |
| 5000 | 50  | 12.168551 | 26.675558 | 36.24042  | 428.1436  |
| 5000 | 40  | 9.697005  | 19.860809 | 27.466842 | 343.80212 |
| 5000 | 25  | 6.101061  | 11.698138 | 16.04878  | 211.55215 |
| 5000 | 10  | 2.468314  | 4.617197  | 6.015017  | 88.452974 |
| 5000 | 5   | 1.244379  | 2.336082  | 2.980778  | 44.467088 |
| 5000 | 1   | 0.279649  | 0.503482  | 0.61225   | 9.064908  |

### D) Παρατηρήσεις-Σχόλια-Συμπεράσματα:

- Αρχικά βλέπουμε ότι για σταθερό αριθμό σημείων  $n$  μεταβαλλόμενου  $k$ , το πρόβλημα είναι σχεδόν γραμμικό, καθώς αυξάνονται τα βήματα για την εξέλιξη του μοντέλου, σχεδόν ακριβώς στον ίδιο βαθμό αυξάνεται και ο χρόνος εκτέλεσης, το οποίο είναι και αναμενόμενο. Αντίθετα για σταθερό  $k$ , αυξανόμενου  $n$ , η αύξηση του χρόνου εκτέλεσης τείνει σε εκθετική, με την  $V3$ , να παίρνει ομαλότερη κλίση από τις άλλες 3 υλοποιήσεις καθώς αυξάνει το  $n$ .
- Για κάθε συνδυασμό  $n$  και  $k$ , και οι 3 υλοποιήσεις ( $V1-V3$ ) της CUDA είναι πολύ γρηγορότερες από αυτές της ακολουθιακής ( $V0$ ) υλοποίησης, με την  $V1$  να είναι έως και 13 περίπου φορές (π.χ.  $n=1000$ ,  $k=100$ ) γρηγορότερη της  $V0$ . Η  $V2$  είναι έως και πάνω από 18 φορές γρηγορότερη από την  $V0$ , ενώ τέλος η  $V3$  φτάνει να είναι έως και πάνω από 35 φορές γρηγορότερη της  $V0$  (π.χ. για  $n=5000$ ,  $k=100$  η  $V3$  είναι πάνω από 36 φορές γρηγορότερη!).
- Αναφορικά με την απόδοση των υλοποιήσεων σε CUDA, αν και για μικρό αριθμό σημείων ( $n=100$ ) η  $V1$ , μοιάζει να είναι γρηγορότερη της  $V2$ , για όλες τις άλλες μετρήσεις η  $V2$  υπερτερεί της  $V1$  ξεκάθαρα και η  $V3$  της  $V2$ . Στην  $V2$  έχουμε επιτάχυνση έως και 25% σε σχέση με τον χρόνο εκτέλεσης της  $V1$  ( $n=10000$ ,  $k=100$ ), ενώ στην  $V3$  συναντάμε επιτάχυνση έως και πάνω από 50% για μερικές υλοποιήσεις σε σχέση με την  $V2$  (π.χ.  $n=5000$ ,  $k=100$ ).
- Η σαφής υπεροχή της  $V2$ , έναντι της  $V1$ , οφείλεται στο γεγονός ότι "οργανώνει" πολύ ορθότερα τα *threads* στους πυρήνες της *gpu*. Αντίθετα η  $V3$ , υπερέχει της  $V2$ , καθώς ενώ "οργανώνει" με ίδιο τρόπο τα *threads*, τα *block* της για τους υπολογισμούς λαμβάνουν δεδομένα από την πολύ γρηγορότερη μνήμη (*shared memory*) των *blocks* και όχι από την *global memory*.
- Επίσης είναι ξεκάθαρο τόσο από τα γραφήματα όσο και από τους πίνακες, ότι όσο αυξάνει το φορτίο (δηλαδή αυξάνουν τα  $n$  και  $k$ ), τόσο πιο έντονες είναι οι διαφορές στον χρόνο εκτέλεσης μεταξύ των υλοποιήσεων και τόσο μεγαλύτερο το "όφελος" που αποκομίζουμε από τις γρηγορότερες υλοποιήσεις.

\*(Περισσότερα γραφήματα, πίνακες και csv καθώς και το σκριπτάκι του matlab για τα γραφήματα, στο repo, στον φάκελο data)\*

