Συστήματα Παράλληλης Επεξεργασίας
Τρίτη Εργαστηριακή Άσκηση

Αναστασία-Χριστίνα Λίβα 03119029
Γιώργος Μυστριώτης 03119065
Νικόλας Σταματόπουλος 03119020

Δεκέμβριος 2023

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

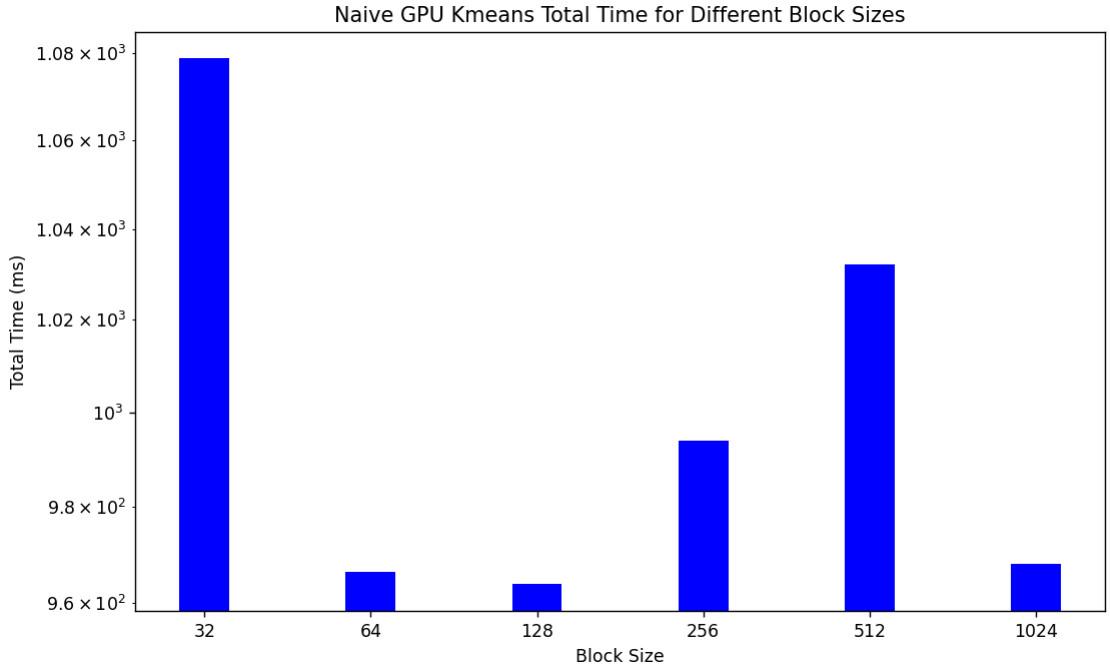
Naive version

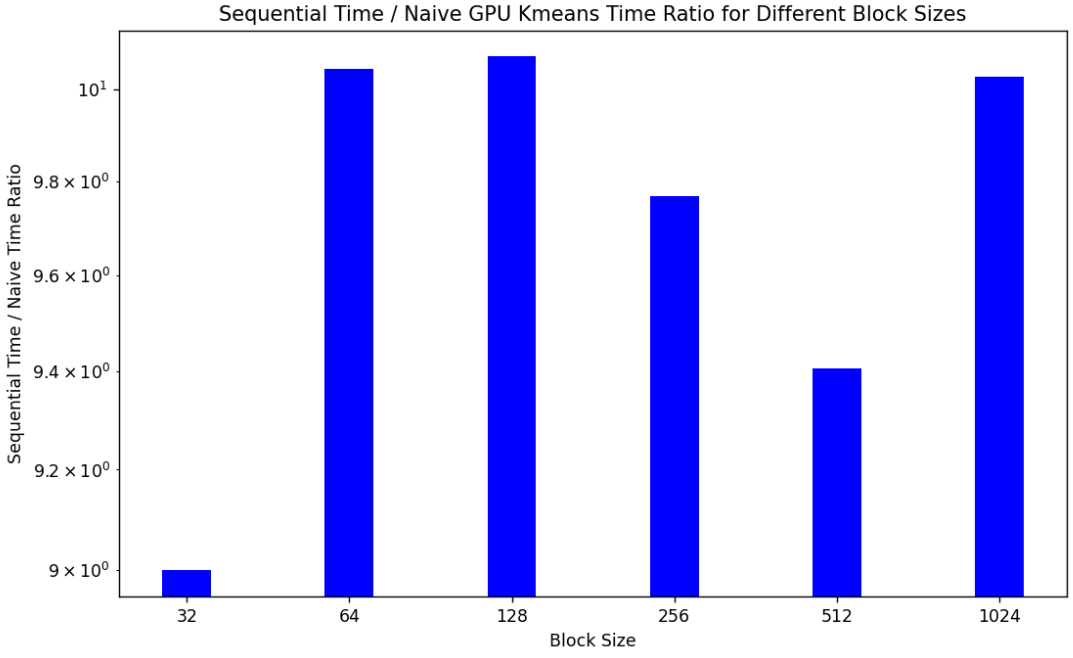
Διαμορφώνοντας κατάλληλα τα ζητούμενα κομάτια του κώδικα (TODOs), με σκοπό να εκτελέσουμε τον υπολογισμό των νέων clusters στην GPU, τρέχουμε την Naive υλοποίηση και παίρνουμε τα παρακάτω αποτελέσματα:

Algorithm	**Block Size**	**Total Time (ms)**
Sequential Kmeans	-	9708.274841
Naive GPU Kmeans	32	1078.787088
Naive GPU Kmeans	64	966.363907
Naive GPU Kmeans	128	963.807106
Naive GPU Kmeans	256	993.892908
Naive GPU Kmeans	512	1032.127142
Naive GPU Kmeans	1024	968.080997

Πίνακας 1: Execution times for Naive Kmeans algorithm and block sizes.

Και επίσης παρουσιάζονται τα ζητούμενα διαγράμματα:





Η τελική διαμόρφωση του κώδικα φαίνεται στον φάκελο των παραδοτέων.

1. Όπως φαίνεται από το διάγραμμα του speedup, σε κάθε περίπτωση πετυχαίνουμε βελτίωση της απόδοσης με χρήση της GPU. Μικρότερο είναι το speedup για block size 32, ενώ την καλύτερη απόδοση πετυχαίνουμε για block size 128. Παρόλαυτα, σε κάθε περίπτωση οι χρόνοι είναι αρκετά κοντά για όλα τα block sizes. Η βελτίωση αυτή οφείλεται στο γεγονός πως ο υπολογισμός των νέων clusters γίνεται με παράλληλο τρόπο στην GPU και επιστρέφονται οι τιμές ώστε να ολοκληρωθεί κάθε loop στην CPU. Παρότι έχουμε μεταφορά δεδομένων μεταξύ CPU - GPU, το κέρδος που έχουμε από το παράλληλο computation στην GPU αντισταθμίζει τον χρόνο που χάνουμε για μεταφορά δεδομένων και έτσι παίρνουμε πολύ καλύτερα αποτελέσματα.

2. Όπως αναφέραμε, το καλύτερο χρόνο πετυχαίνουμε για block size 128 και τον χειρότερο για block size 32. Το warp size είναι σταθερά 32 threads. Συνεπώς, όταν έχουμε πολλαπλά warps σε ένα block μπορούμε να πετύχουμε καλύτερο occupancy, όπου occupancy είναι ο αριθμός active warps ανά block. Αυτό μας δίνει καλύτερο utilization των SM και άρα καλύτερους χρόνους. Για παράδειγμα, κάποιο warp μπορεί να μην είναι διαθέσιμο για εκτέλεση (δηλαδή να γίνει active) επειδή κάποιο thread του περιμένει κάποιο access στην κύρια μνήμη ή επειδή για κάποιο thread του έτυχε να αργήσει η εκτέλεσή του. Όταν όμως έχουμε μεγαλύτερα blocks και άρα περισσότερα threads per block και warps per block, αν ένα warp κάνει stall λόγω memory access, ένα άλλο μπορεί να πάρει την θέση του μέσω του warp scheduler. Βέβαια αντίστοιχη συμπεριφορά έχουμε και για τα μικρά block sizes, όπως το 32, αφού κάθε SM αναλαμβάνει την εκτέλεση πολλαπλών blocks. Όμως, θέτοντας ένα ολόκληρο block inactive, για να επαναφερθεί σε λειτουργεία, θα πρέπει να ξαναγίνουν allocate οι registers του (και η shared memory του σε περίπτωση που είχαμε). Επιπλέον, κάθε SM μπορεί να αναλάβει ένα συγκεκριμένο αριθμό blocks. Έχοντας μικρό block size αυτό σημαίνει πως θα έχουμε και μικρό αριθμό active warps. Ενώ για μεγαλύτερα blocks έχουμε και περισσότερα active threads για πιο γρήγορους υπολογισμούς, αλλά και περισσότερα warps για βελτίωση του occupancy.

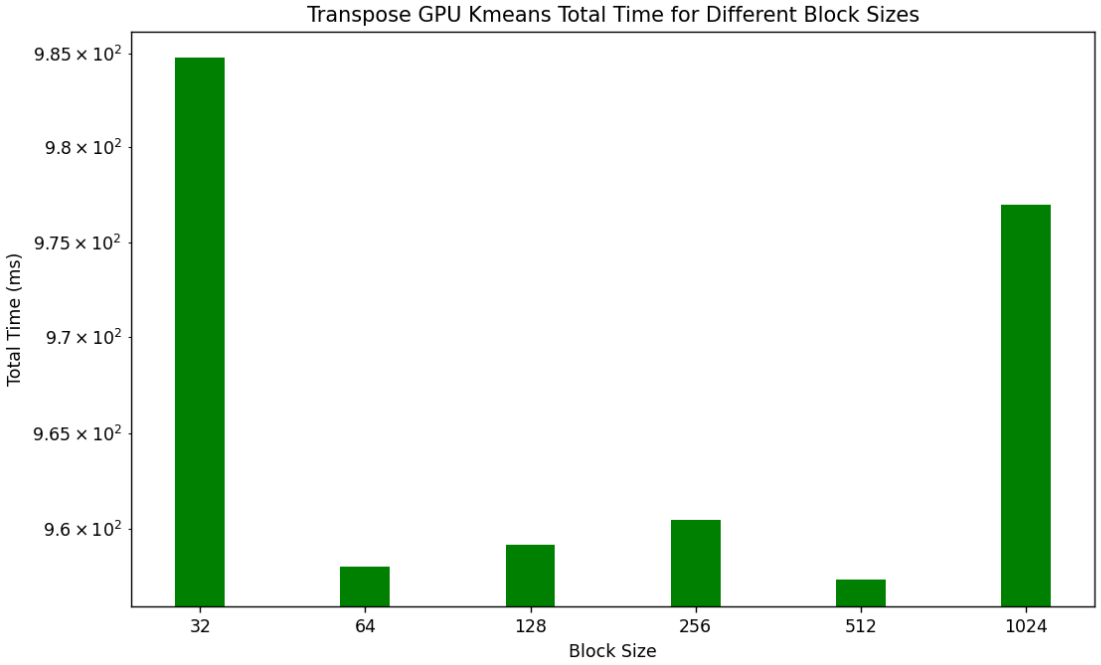
Transpose version

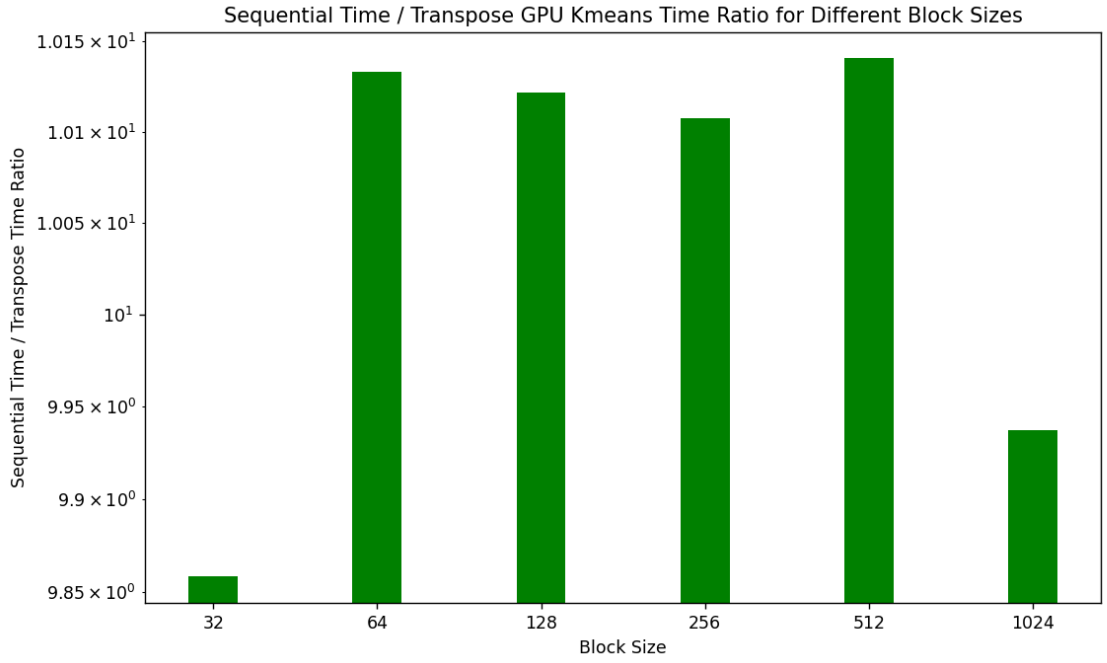
Στην Transpose υλοποίηση αλλάζουμε την δομή των δεδομένων από row-based σε column-based indexing, δηλαδή για παράδειγμα έχουμε objects[NumObjs][NumCoords] και πλέον έχουμε και dimObjects[NumCoords][NumObjs]. Τα αποτελέσματα για αυτή την υλοποίηση φαίνονται παρακάτω:

Algorithm	**Block Size**	**Total Time (ms)**
Sequential Kmeans	-	9708.274841
Transpose GPU Kmeans	32	984.7579
Transpose GPU Kmeans	64	958.055019
Transpose GPU Kmeans	128	959.161043
Transpose GPU Kmeans	256	960.49118
Transpose GPU Kmeans	512	957.370996
Transpose GPU Kmeans	1024	976.952076

Πίνακας 2: Execution times for Transpose GPU Kmeans algorithm and block sizes.

Και επίσης παρουσιάζονται τα ζητούμενα διαγράμματα:





Η τελική διαμόρφωση του κώδικα φαίνεται στον φάκελο των παραδοτέων.

1. Όπως φαίνεται από τα διαγράμματα, με την transpose έκδοση του αλγορίθμου, πετυχαίνουμε ακόμα καλύτερους χρόνους σε σχέση με την naive. Αντίστοιχα, όμως, με την naive, έχουμε παρόμοια κατανομή των χρόνων σε σχέση με το block size. Δηλαδή, έχουμε τον μεγαλύτερο χρόνο για block size 32, ενώ τον μικρότερο για block size 512. Η μόνη διαφορά είναι πως στην naive υλοποίηση είχαμε αργό χρόνο για block size 512, ενώ εδώ είχαμε αργό χρόνο για block size 1024, εξαιρώντας το μικρό block size 32. Οι μετρήσεις αυτές όμως μπορεί να οφείλονται σε outlying μετρήσεις και να μην αντιπροσωπεύουν το πραγματικό αποτέλεσμα, αφού όπως αναφέραμε και προηγουμένως μπορούμε να επωφελειθούμε από το μεγαλύτερο block size. Συνεπώς το block size έχει τον ίδιο ρόλο που έχει και στην naive version.

2. Όμως, αναφέραμε πως υπήρξε speedup σε σχέση με την naive έκδοση. Αυτό οφείλεται στην αλλαγή δομής των δεδομένων, αλλά και στον τρόπο με τον οποίο κάνουν access τα threads παράλληλα την μνήμη. Πιο αναλυτικά, σε ένα block, ό-ταν ένα thread κάνει access την μνήμη, τα γειτονικά του κάνουν και αυτά access τις γειτονικές θέσεις. Αυτό το εκμεταλλεύεται η transpose έκδοση, αφού έχοντας διαστάσεις objects[NumCoords][NumObjects], όταν ένα thread (δηλαδή ένα object) κάνει access την μία διάστασή του, αυτόματα κάνουν και τα υπόλοιπα threads του block σε ένα action. Αντίθετα, στην naive έκδοση, κάθε thread θα πρέπει να κάνει το δικό του action για ανάγνωση των συντεταγμένων του, αφού έχουμε πρώτα τις συντεταγμένες του 1ου object, μετά του 2ου κ.ο.κ.. Όσον αφορά τα clusters, εκεί δεν έχουμε κάποιο speedup, αφού όλα τα threads θα πρέπει να διαβάσουν διαδοχικά όλες τις τιμές των συντεταγμένων των clusters. Όμως, όπως φαίνεται και από τα αποτελέσματα, η ταυτόχρονη αυτή πρόσβαση στη μνήμη σε κάθε loop, μας δίνει ένα επιπλέον speedup σε σχέση με την naive έκδοση.

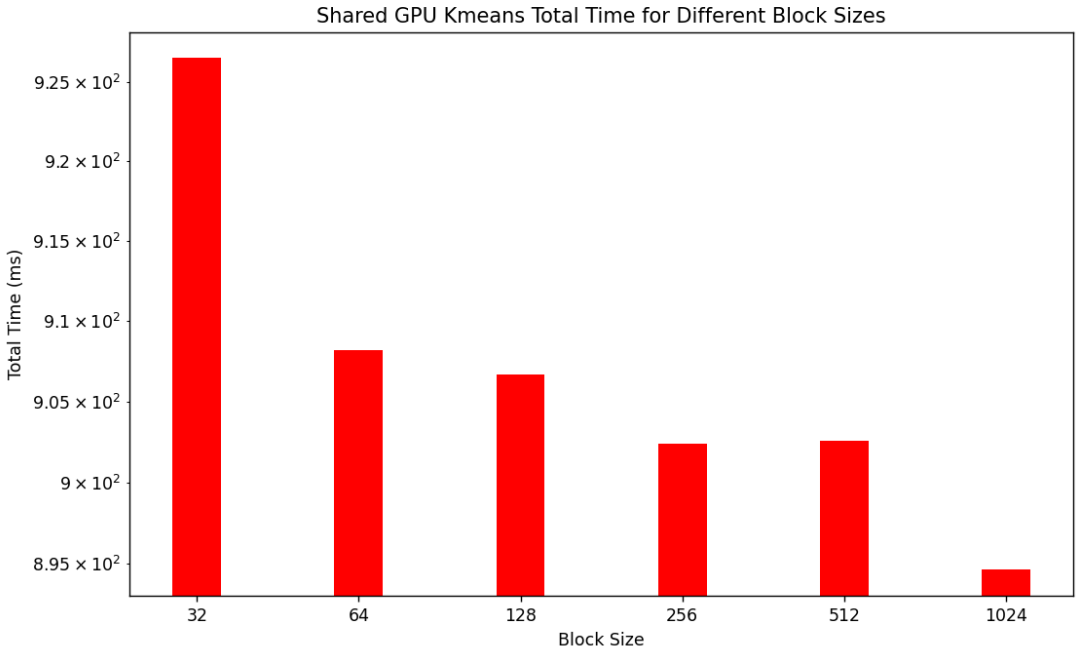
Shared version

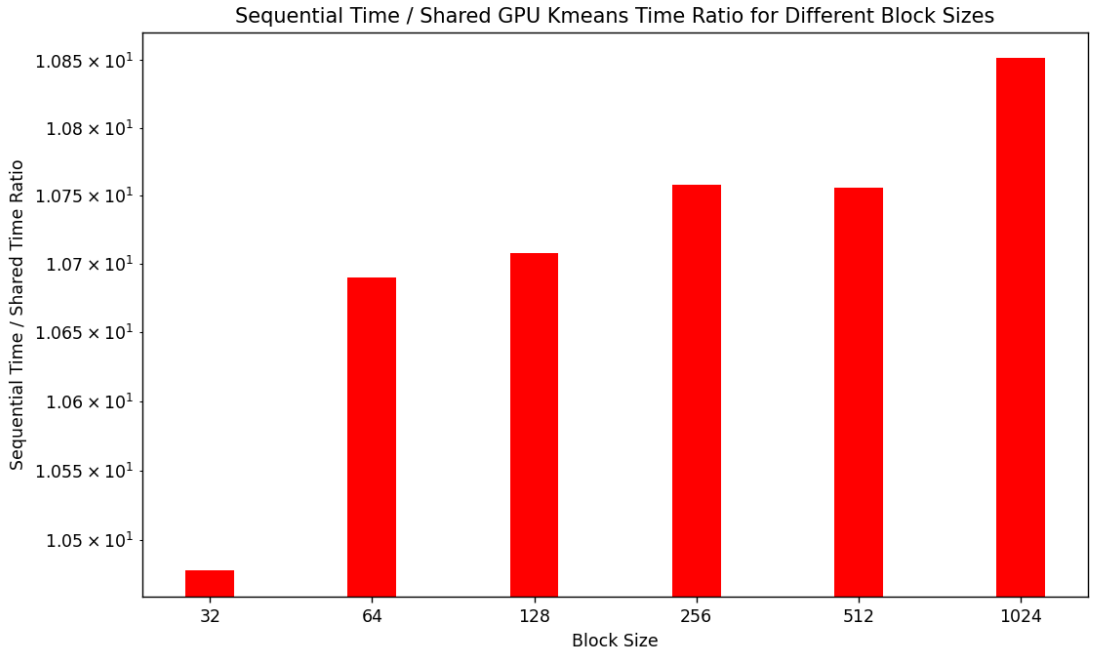
Στην Shared υλοποίηση κάνουμε χρήση του shared memory των blocks ώστε τα κέντρα των clusters, που διαβάζονται συχνά από τα threads, να γίνονται accessed πιο γρήγορα. Παράλληλα κρατάμε την μορφή δεδομένων της Transpose εκδοχής. Για να μεταφέρουμε τα δεδομένα στο shared memory του κάθε block, βάζουμε το κάθε thread του να γράψει ένα κομμάτι του πίνακα. Δηλαδή αν τα threads μας είναι περισσότερα από τα στοιχεία του deviceClusters τότε το καθένα γράφει από ένα, αλλιώς γράφουν όλα ανά blockDim. Τα αποτελέσματα για αυτή την υλοποίηση φαίνονται παρακάτω:

Algorithm	**Block Size**	**Total Time (ms)**
Sequential Kmeans	-	9708.274841
Shared GPU Kmeans	32	926.521063
Shared GPU Kmeans	64	908.156157
Shared GPU Kmeans	128	906.657934
Shared GPU Kmeans	256	902.411938
Shared GPU Kmeans	512	902.587891
Shared GPU Kmeans	1024	894.640207

Πίνακας 3: Execution times for Shared GPU Kmeans algorithm and block sizes.

Και επίσης παρουσιάζονται τα ζητούμενα διαγράμματα:



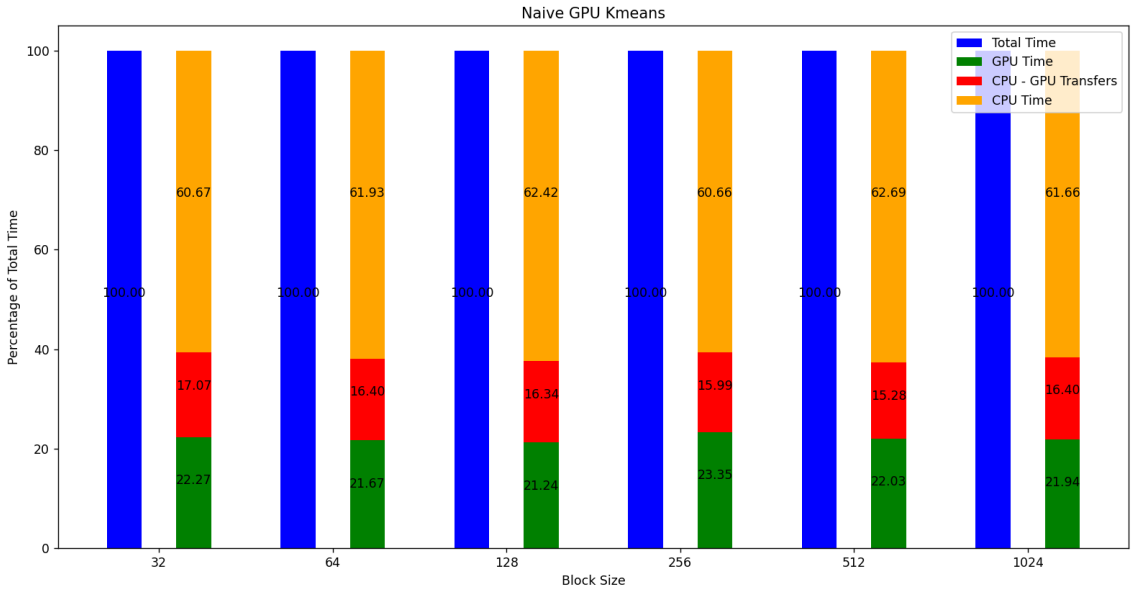


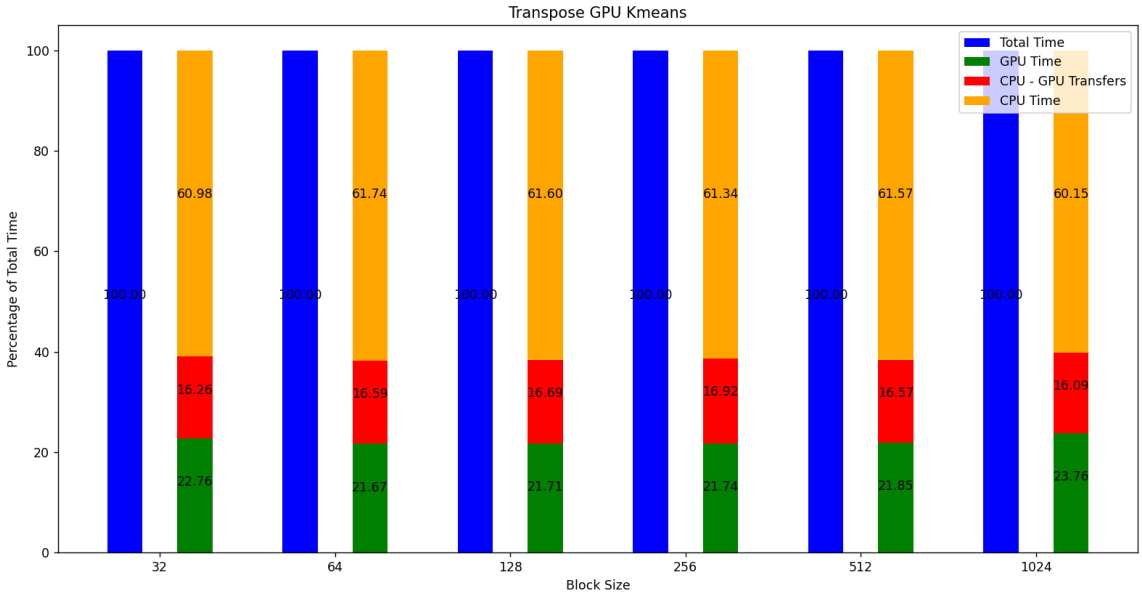
Η τελική διαμόρφωση του κώδικα φαίνεται στον φάκελο των παραδοτέων.

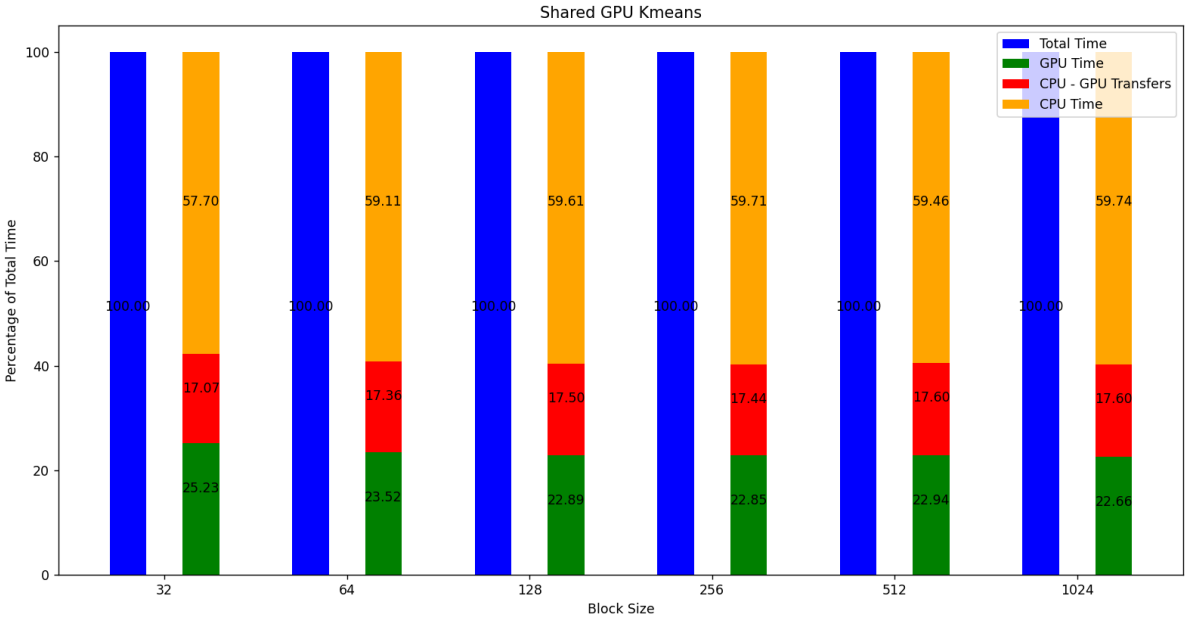
1. Όπως βλέπουμε από τον πίνακα των αποτελεσμάτων, με την shared έκδοση του αλγορίθμου έχουμε τον ταχύτερο χρόνο για block size = 1024, και είναι ο ταχύτερος όλων των χρόνων που έχουμε πετύχει. Αυτό επιτυγχάνεται μέσω της χρήσης του shared memory του κάθε block. Η μνήμη αυτή μας επιτρέπει να έχουμε γρήγορη πρόσβαση σε δεδομένα από όλα τα threads του block. Στην περίπτωσή μας, όλα τα threads διαβάζουν όλες τις συντεταγμένες των κέντρων. Όπως αναφέραμε βάζουμε κάθε block να αντιγράψει στη shared memory του τα κέντρα του K-means, προσέχοντας να συγχρονιστούν τα threads πριν την έναρξη των υπολογισμών, με χρήση __syncthreads(). Κοιτώντας τώρα τα διαγράμματα που αφορούν τον χρόνο έναντι του block size, μπορούμε να δούμε μια διαφορετική κατανομή σε σχέση με αυτή που είχαμε μέχρι τώρα. Βλέπουμε πως όσο μεγαλώνει το block size τόσο καλύτερους χρόνους και speedup παίρνουμε. Το αποτέλεσμα αυτό είναι λογικό, αφού αρχικά, για μεγαλύτερα block sizes γράφουμε πιο γρήγορα τα δεδομένα στη shared memory, ενώ παράλληλα περισσότερα active warps θα τα χρησιμοποιήσουν στη συνέχεια για τους υπολογισμούς τους. Επιπλέον, δεν εκτελούνται write operations, οπότε δεν έχουμε contest για τα κοινά δεδομένα. Συνεπώς επωφελούμαστε συνεχώς για μεγαλύτερο block size, αυξάνοντας το occupancy και μειώνοντας τον χρόνο αντιγραφής.

Σύγκριση υλοποίησεων / bottleneck Analysis

Για την αναλυτικότερη μελέτη και σύγκριση των υλοποιήσεων τοποθετούμε timers σε κατάλληλα σημεία εντός του while loop, με σκοπό να μετρήσουμε GPU Time, CPU Time και CPU - GPU Transfer Time. Η προσθήκη των timers φαίνεται στις υλοποιήσεις που υπάρχουν στον φάκελο των παραδοτέων. Συνεπώς έτσι παρουσιάζουμε τους χρόνους αυτούς ως ποσοστά του συνολικού χρόνου:



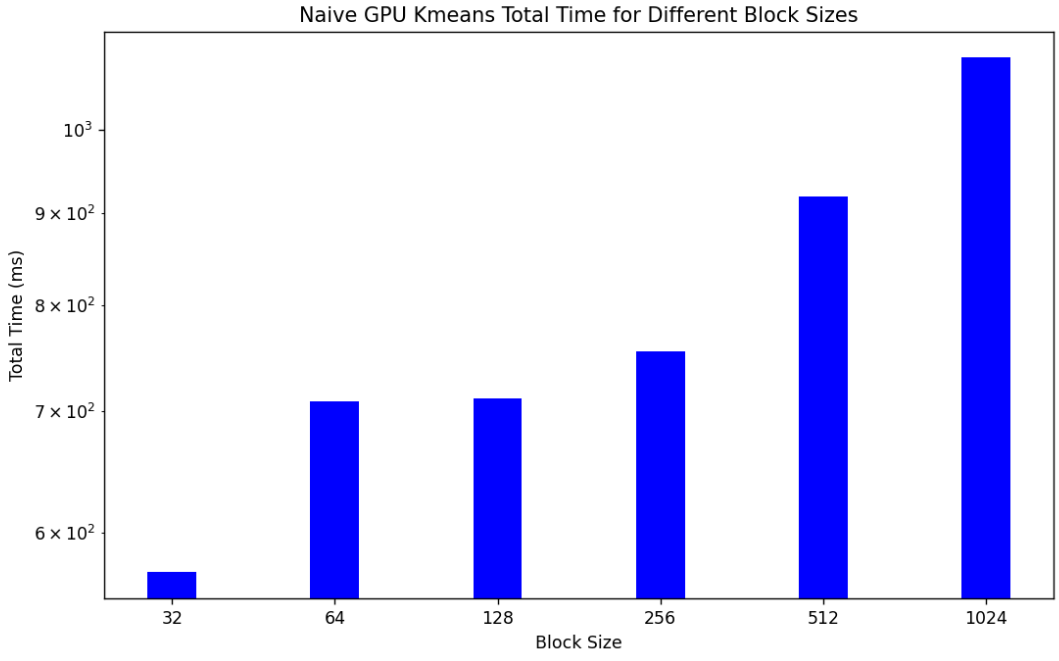

Transpose GPU Kmeans
Total Time
GPU Time
CPU - GPU Transfers
CPU Time
Percentage of Total Time
Block Size
0
20
40
60
80
100
32
64
128
256
512
1024
100.00
100.00
100.00
100.00
100.00
100.00
60.98
61.74
61.60
61.34
61.57
60.15
16.26
16.59
16.69
16.92
16.57
16.09
22.76
21.67
21.71
21.74
21.85
23.76

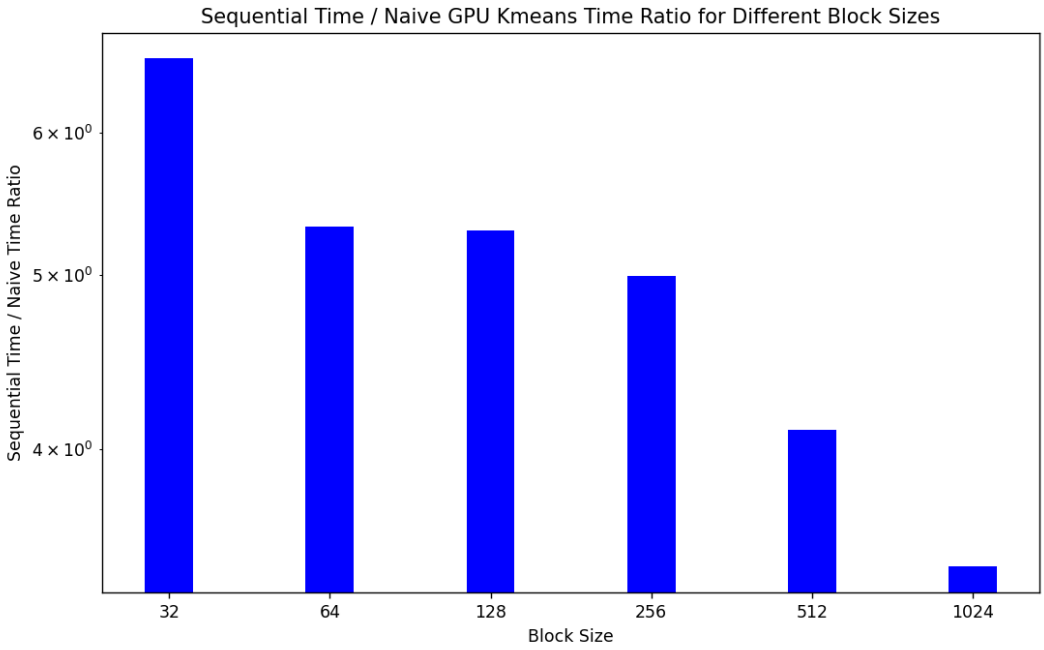

Shared GPU Kmeans
Total Time
GPU Time
CPU - GPU Transfers
CPU Time
Percentage of Total Time
Block Size
0
20
40
60
80
100
32
64
128
256
512
1024
100.00
100.00
100.00
100.00
100.00
100.00
57.70
59.11
59.61
59.71
59.46
59.74
17.07
17.36
17.50
17.44
17.60
17.60
25.23
23.52
22.89
22.85
22.94
22.66

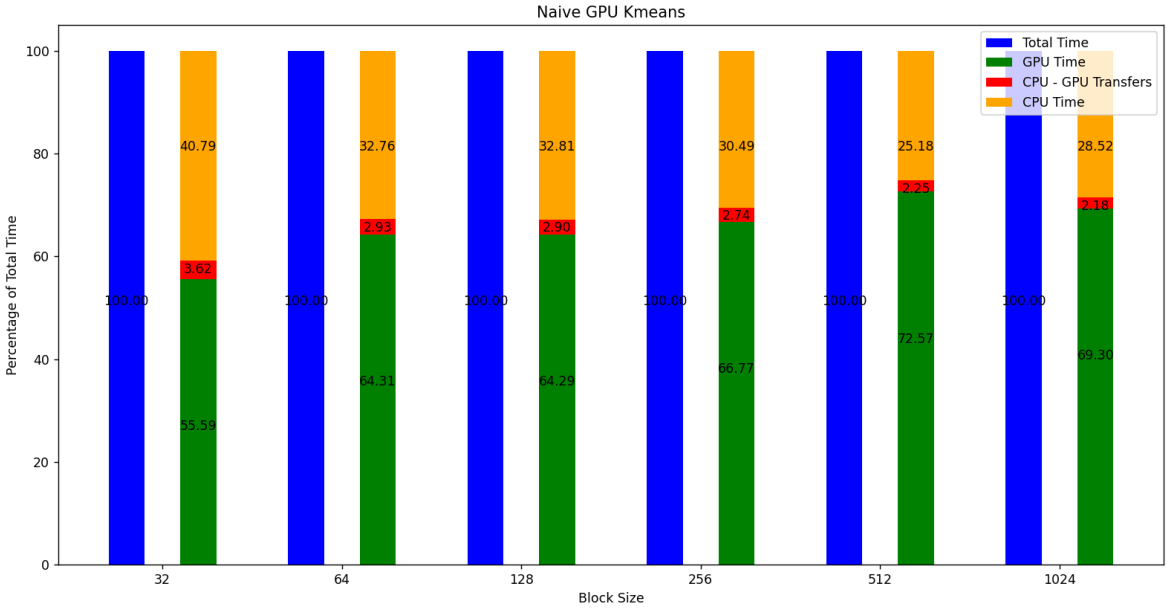
1. Όπως μπορούμε να δούμε από τα διαγράμματα, το μεγαλύτερο μέρος χρόνου, αφορά χρόνο εκτέλεσης στη CPU. Αυτό αποτελεί σοβαρό bottleneck στις υλοποιήσεις μας, αφού όσο και να βελτιώσουμε τους αλγορίθμους μας η εκτέλεσή τους στην GPU δεν αφορά το μεγαλύτερο μέρος του χρόνου και άρα θα δούμε μικρή βελτίωση. Αντίστοιχα, όπως είναι επιθυμητό, μόνο ένα πολύ μικρό χρονικό διάστημα αφορά μετακίνηση δεδομένων μεταξύ CPU και GPU, που σημαίνει πως δεν χάνεται μεγάλος χρόνος άσκοπα σε μεταφορές. Η CPU εκτελεί υπολογισμούς για τα νέα cluster centers με βάση τα memberships που υπολογίστηκαν στην GPU. Συνεπώς, αν μπορούσαμε να μειώσουμε τον χρόνο στη CPU, είτε με παραλληλοποίηση είτε με εκτέλεση όλων των υπολογισμών στην GPU, θα είχαμε καλύτερους χρόνους με καλύτερη αξιοποίηση των πόρων μας. Ας μην ξεχνάμε κι όλας πως για tasks που αφορούν ίδια operations πολλαπλές φορές η στρατηγική SIMT που υποστηρίζεται από την GPU μπορεί να φανεί ιδιαίτερα χρήσιμη για την μείωση του χρόνου εκτέλεσης.

2. Αν τώρα αλλάξουμε τον αριθμό των συντεταγμένων σε 16, οπότε έχουμε συνθήκες 256,16,16,10, θα έχουμε μικρότερο αριθμό objects αλλά περισσότερες συντεταγμένες, άρα κάθε object απαιτεί μεγαλύτερο computation για να βρει το κοντινότερο cluster του. Έτσι παράγουμε και πάλι τα διαγράμματα για την μελέτη του νέου αυτού configuration:

Naive:



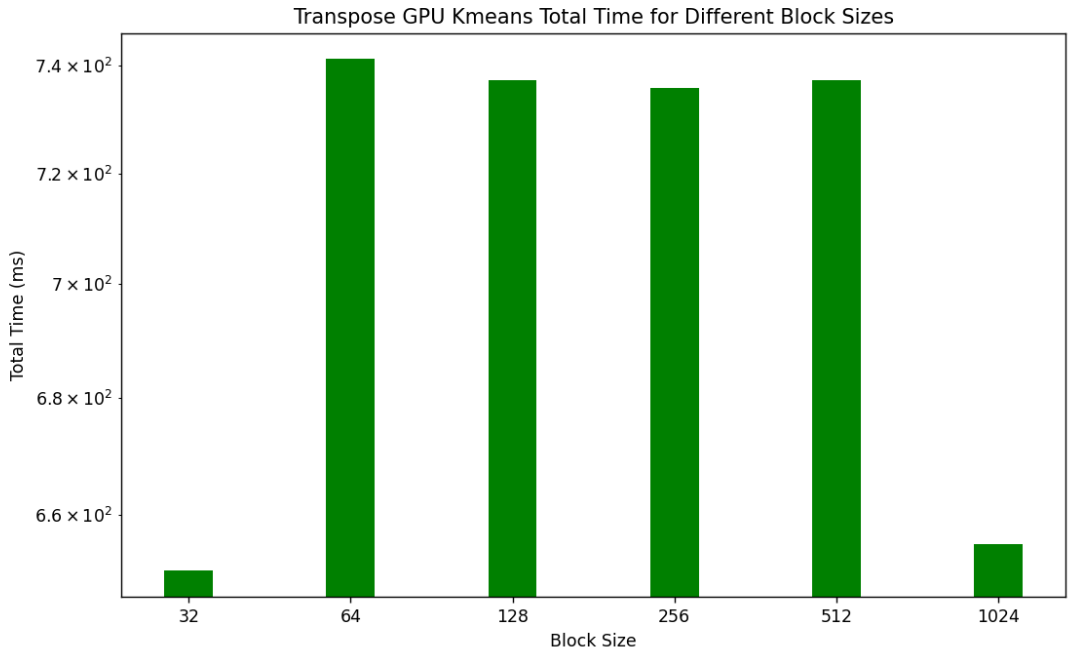


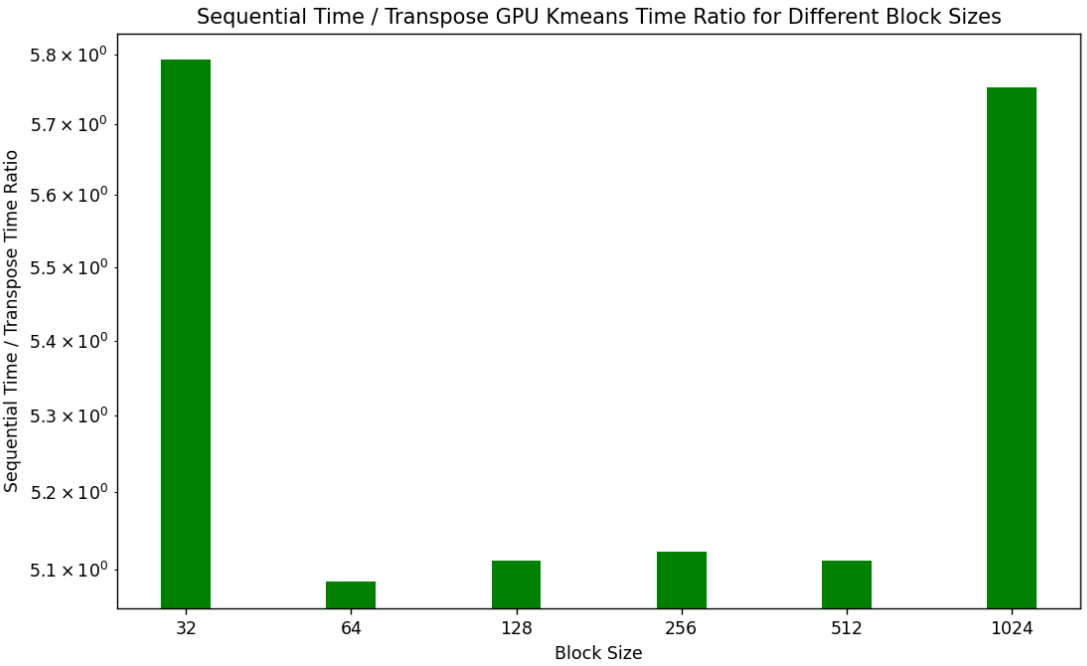


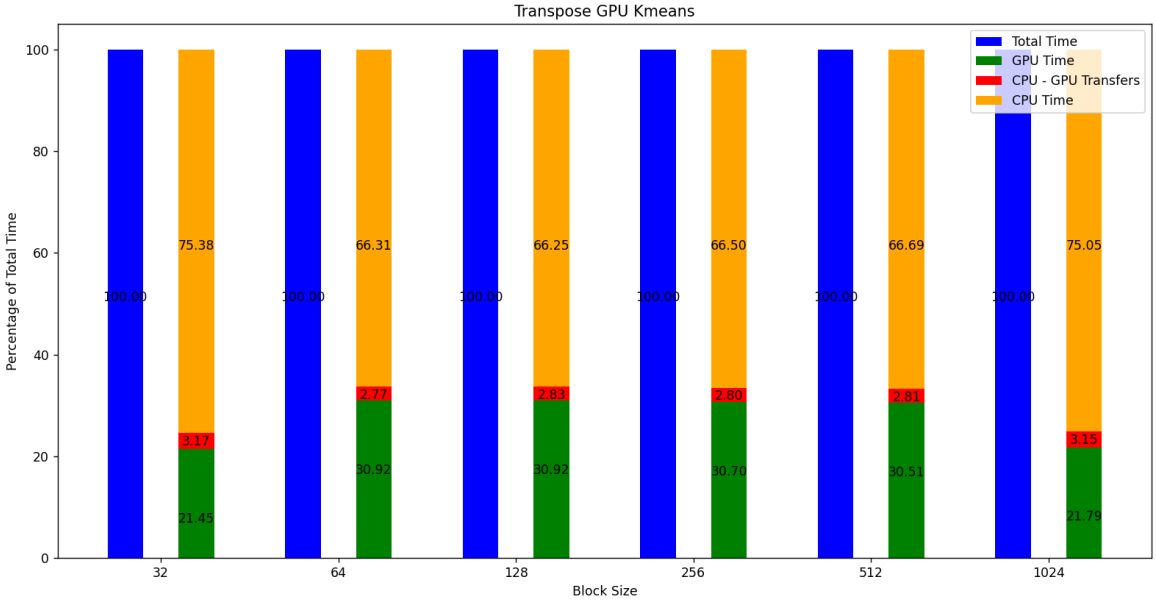
α) Αρχικά για την Naive εκδοχή μπορούμε να δούμε πως πλέον τον καλύτερο χρόνο πετυχαίνουμε για block size 32. Αυτό συμβαίνει επειδή στο νέο μας configuration έχουμε σημαντικά λιγότερα objects (λόγω του σταθερού size, αλλά αυξημένων coordinates) και άρα λιγότερα threads, από τα οποία όμως το καθένα απαιτεί μεγαλύτερο computation time. Ειδικά στην περίπτωσή μας όπου έχουμε πλέον 16x16 double αριθμούς για τα κέντρα και δεν χρησιμοποιούμε shared memory ή κάποια γρήγορη cache της GPU, στα μεγάλα blocks έχουμε μεγάλο contest για του registers αλλά και πολλαπλά αργά memory accesses, με αποτέλεσμα να έχουμε συνεχώς activation/deactivation των warps των οποίων τα threads περιμένουν να γίνουν available τα registers ή να έρθουν δεδομένα από τη μνήμη. Αντίθετα, στα μικρά blocks έχοντας λιγότερα threads και άρα μικρότερο contest. Ενώ στο προηγούμενο configuration,

τα μεγάλα block sizes εκμεταλλεύονταν το μεγαλύτερο occupancy, εδώ υποφέρουν από το μεγαλύτερο computation per thread, που απαιτεί και περισσότερη πρόσβαση σε registers. Σημειώνουμε πως το αυξημένο computation time μπορεί να φανεί και στο παραπάνω διάγραμμα, όπου το GPU time αποτελεί το 64% - 70% του συνολικού χρόνου.

Transpose:

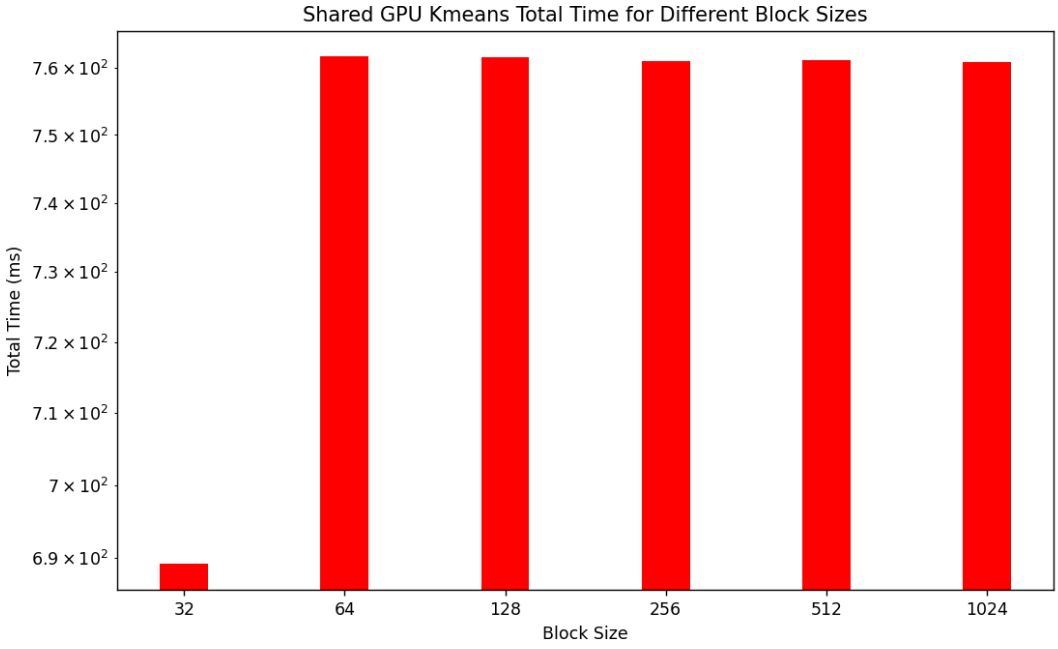


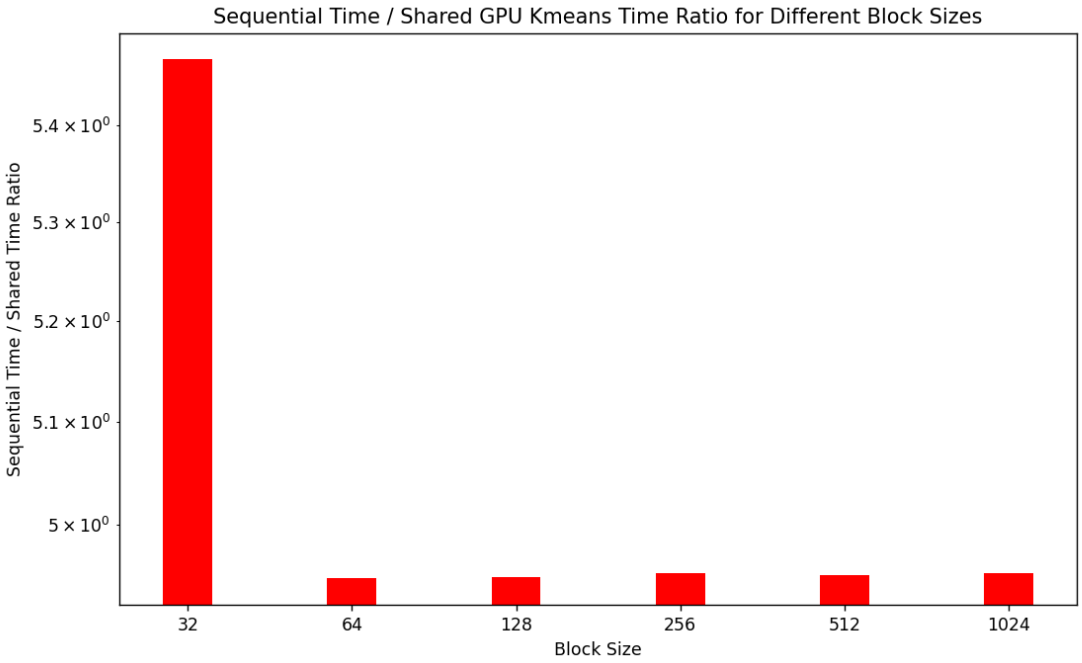


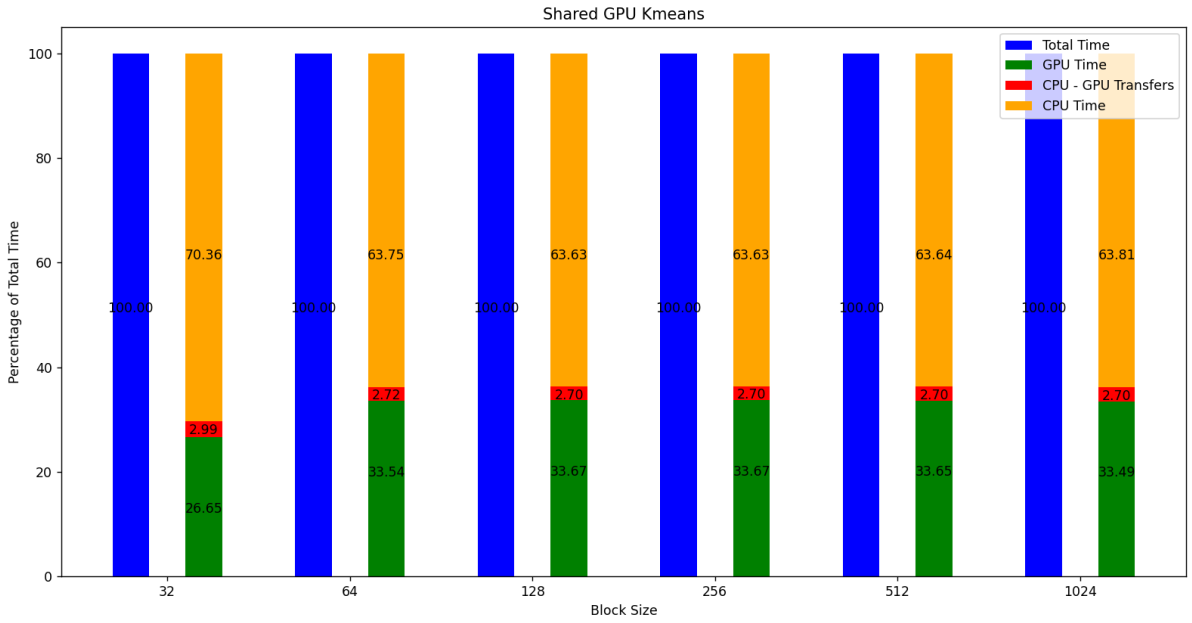


β) Όσον αφορά την Transpose έκδοση, παίρνουμε κατά μέσο όρο καλύτερα αποτελέσματα. Εκμεταλλευόμενη το παράλληλο memory access εντός ενός warp μειώνουμε σημαντικά το συνολικό computation time, έχοντας κάνει transpose του πίνακές μας. Παρότι το contest για registers παραμένει μεγάλο, κάνοντας συνεχόμενα access μνήμης μπορούμε να μειώσουμε τα συνολικά accesses και άρα να βελτιώσουμε την επίδοση της εκτέλεσης. Κοιτώντας το διάγραμμα των ποσοτών χρόνων, μπορούμε να δούμε πως, ενώ έχει μειωθεί και ο συνολικός χρόνος (1ο διάγραμμα), έχει μειωθεί και το ποσοστό χρόνου στην GPU. Άρα και πάλι το bottleneck είναι η CPU.

Shared:







γ) Στην Shared έκδοση του αλγορίθμου βλέπουμε πως πετυχαίνουμε λίγο χειρότερα αποτελέσματα από την Transpose. Κοιτώντας τις μετρήσεις μας για τους διάφορους χρόνους βλέπουμε πως το CPU time και το CPU - GPU transfer time είναι το ίδιο και στις δύο υλοποιήσεις. Αντίθετα, το GPU time στην Transpose είναι περίπου 226ms ενώ στην Shared 255ms. Συνεπώς, παρότι χρησιμοποιούμε την shared memory για να κάνουμε πιο γρήγορα access τα δεδομένα μας, βλέπουμε αργότερους χρόνους. Πιθανότατα, ο χρόνος που απαιτείται για την δέσμευση μνήμης και την αντιγραφή των δεδομένων στην κάθε shared memory κοστίζει περισσότερο από όσο κερδίζουμε από την γρηγορότερη πρόσβαση. Παρόλαυτά, λόγω του αυξημένου computation ο ανταγωνισμός για τους registers παραμένει υψηλός.

Έτσι είναι λογικό να συμπεράνουμε πως η συγκεκριμένη shared υλοποίηση δεν είναι κατάλληλη για οποιοδήποτε configuration δώσουμε. Αν ο όγκος δεδομένων των κέντρων είναι πέρα από ένα συγκεκριμένο όριο, χάνουμε τα πλεονεκτήματα που είδαμε στο πρώτο μας configuration.