Συστήματα Παράλληλης Επεξεργασίας

Εργαστηριακή αναφορά 5

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

Γιάννος Παράνομος - 03118021
Νικήτας Τσίννας - 03118187
Καούκης Γιώργος - 03119006

Ιανουάριος 2023

1 Εισαγωγή

Στόχος της άσκησης είναι η υλοποίηση και βελτιστοποίηση του αλγορίθμου K-means σε μια κάρτα γραφικών NVIDIA με τη χρήση του εργαλείου CUDA. Προς αυτό το σκοπό, θα φτιάξετε μια αρχική υλοποίηση που να λειτουργεί σωστά, και στη συνέχεια με βάση αυτή θα δοκιμάσετε κάποιες συνήθεις βελτιώσεις επίδοσης για κάρτες γραφικών καθώς και θα αξιολογήσετε την τελική επίδοση του παράλληλου προγράμματος.

2 Naive Version

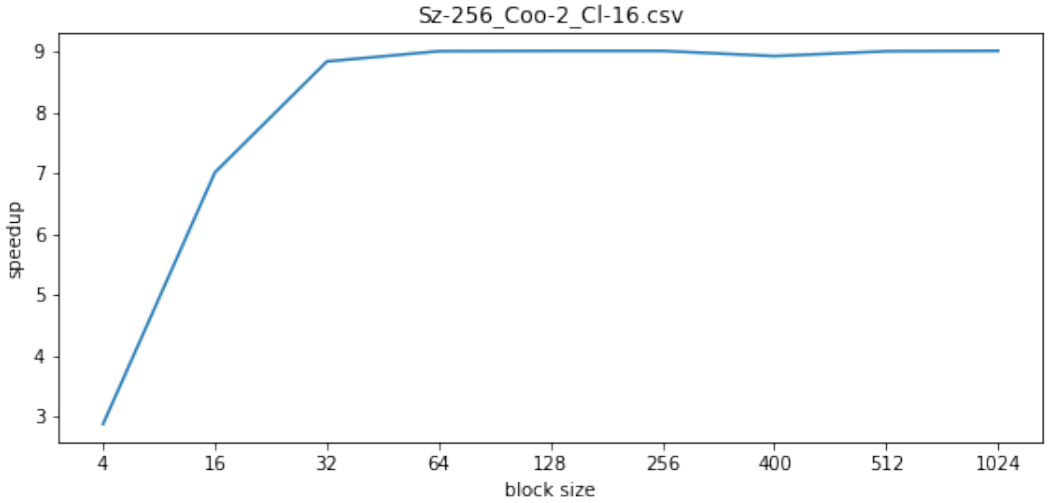
Σε αυτή την έκοδοση αναθέτουμε στην κάρτα γραφικών το πιο υπολογιστικά βαρύ κομμάτι του αλγορίθμου, δηλαδή τον υπολογισμό των nearest clusters σε κάθε iteration. Μετά την εκτέλεση για το configuration Size, Coords, Clusters, Loops = 256, 2, 16, 10 και για block_size = 32, 64, 128, 256, 512, 1024 πήραμε μετρήσεις και δημιουργήσαμε το παρακάτω barplot χρόνου και διάγραμμα speedup. Σε αυτή την έκδοση βλέπουμε μέχρι και 9x επιτάχυνση από την σειριακή έκδοση.

Παρατηρούμε πως οι χρόνοι εκτέλεσης για κάθε block size δεν διαφέρουν σημαντικά. Δηλαδή, το speedup δεν παρουσιάζει ιδιαίτερη κλιμάκωση. Ωστόσο, η επίδοση κλιμακώνει μέχρι και για block size 64. Ειδικά για μικρότερα block sizes (4,16) το speedup είναι πολύ μικρότερο.

Γενικά, μικρός αριθμός threads ανά block ισοδυναμεί με περισσότερα blocks ανά SM στην GPU. Δηλαδή, το μερίδιο κοινής μνήμης που αντιστοιχεί σε κάθε block του SM είναι σημαντικά μικρότερο αφού η μνήμη διαιρείται σε περισσότερα μέρη. Το κάθε thread σε ένα block εκετελεί πράξεις ώστε να βρει σε ποιο cluster ανήκει. Δηλαδή, κάθε thread θέλει να διαβάσει όλα τα cluster centers σε κάθε iteration καθώς και τις συντεταγμένες που του αντιστοιχούν (object coordinates). Για να γίνει αυτό αποδοτικά, θα πρέπει η μνήμη του block να είναι αρκετή ώστε να χωρέσει όσο

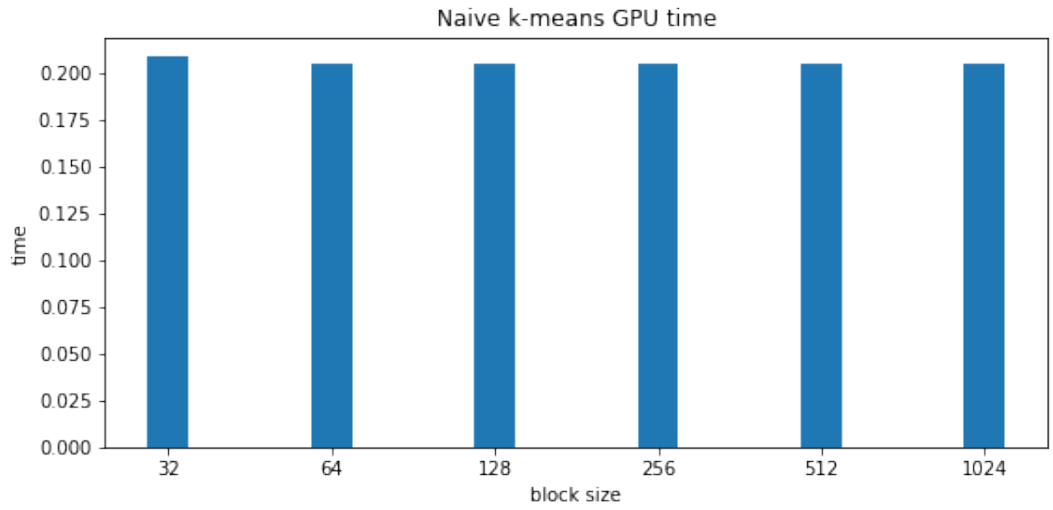
το δυνατόν περισσότερα δεδομένα (δηλαδή clusters και objects) χωρίς να χρειαστεί να κάνει πολλές φορές fetch από το device memory, κάτι που στοιχίζει στις GPU. Δηλαδή, θέλουμε να καλύψουμε το performance loss της μεταφοράς.

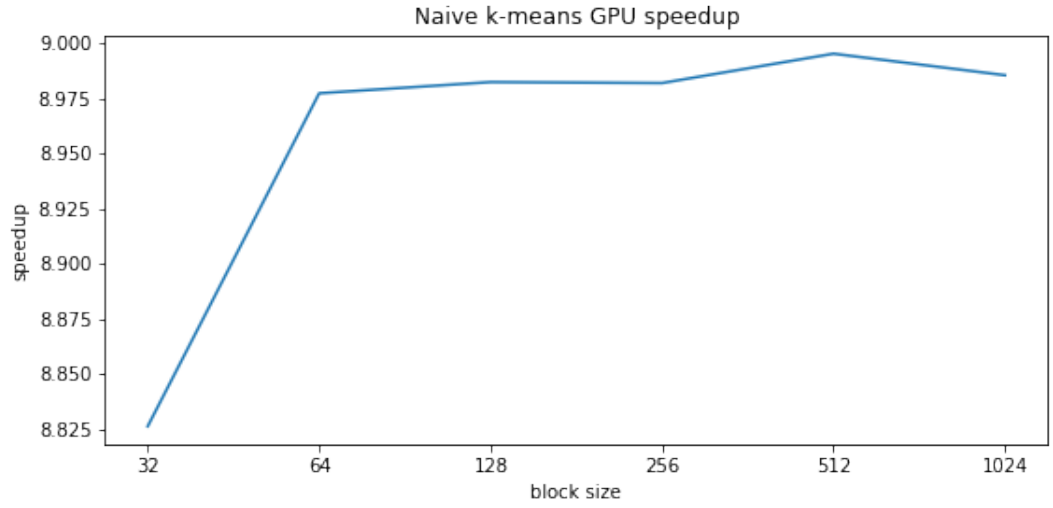
Στα παρακάτω διαγράμματα, όπου έχουμε συμπεριλάβει για λόγους εποπτείας τα Block Sizes 4, 16, μπορούμε να δούμε ξεκάθαρα τον ρόλο του Block Size. Παρατηρούμε ότι από Block Size 4 σε 16, έχουμε speedup x4. Αντίστοιχα όταν έχουμε Block Size=32, παρατηρούμε speedups x2 σε σχέση με το Block Size 16, αφού τώρα φέρνουμε κατευθείαν όλα τα δεδομένα που χρειάζονται.



Στη συνέχεια, αυξάνοντας περεταίρω το Block Size σε 64, παρατηρούμε το βέλτιστο speedup για τις δεδομένες παραμέτρους, όμως τρέχοντας το πρόγραμμα για διάφορους συνδυασμούς, παρατηρούμε ότι σε κάθε περίπτωση το βέλτιστο speedup πετυγχάνεται για Block Size=128 (δεν έχουμε loss αυξάνοντάς το από 64 σε 128). Μάλιστα ψάχνοντας σε documentation από παράλληλες υλοποιήσεις αντίστοιχων προβλημάτων, βλέπουμε ότι το Block Size 128 προτείνεται καθώς φαίνεται να ισορροπεί τα οφέλη από τις παρακάτω παραμέτρους που επηρεάζουν την επιλογή του Block Size για ένα παράλληλο πρόγραμμα.

- Εάν ο αριθμός των threads σε ένα Block είναι πολύ μικρός, ενδέχεται να μην χρησιμοποιούνται πλήρως οι διαθέσιμοι πόροι υλικού, οδηγώντας δυνητικά σε χαμηλότερη απόδοση. Από την άλλη πλευρά, εάν ο αριθμός των νημάτων είναι πολύ μεγάλος, μπορεί να οδηγηθούμε σε διαμάχη πόρων ή/και υπερβολική χρήση μνήμης, η οποία μπορεί επίσης να υποβαθμίσει την απόδοση.
- Ένα μεγαλύτερο μέγεθος μπλοκ μπορεί να είναι ωφέλιμο για την βελτιστοποίηση του παραλληλισμού και την μείωση latency χρόνου για ορισμένες λειτουργίες, όπως οι προσβάσεις στη μνήμη. Ωστόσο, μπορεί επίσης να οδηγήσει σε μεγαλύτερο regiter usage ανά thread και χαμηλότερο occupancy, γεγονός που θα μπορούσε να περιορίσει τον αριθμό των Block που μπορούν να εκτελούνται ταυτόχρονα στο υλικό.



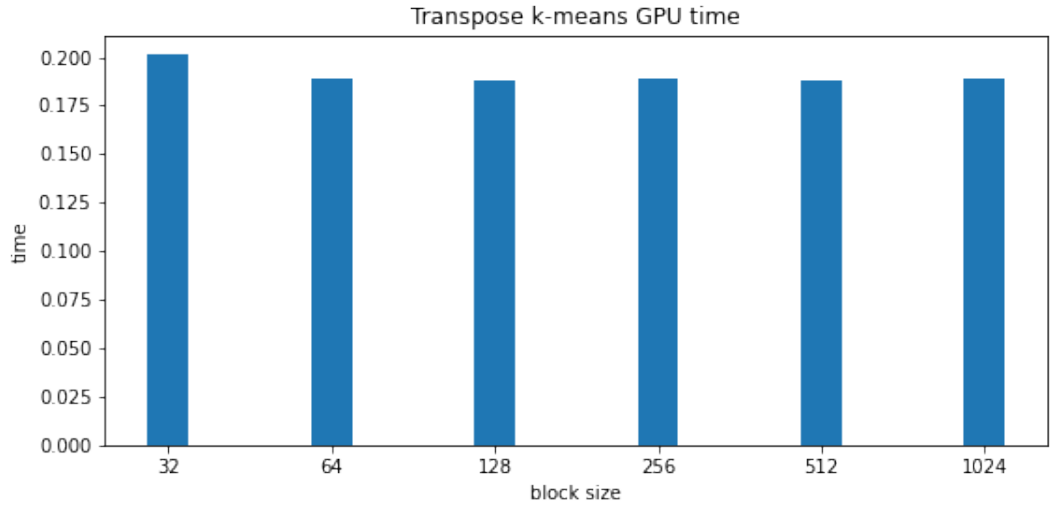


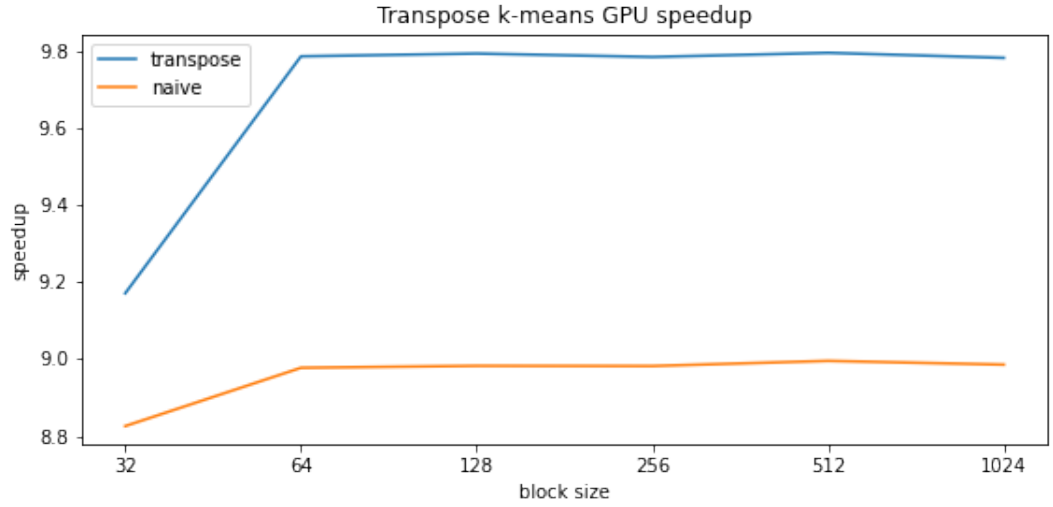
3 Transpose Version

Σε αυτή την έκδοση αλλάζουμε απλώς την δομή των δεδομένων που έχουν διαστάσεις (e.g. objects & clusters) απο row-based σε column-based indexing. Με αυτόν τον τρόπο βελτιώνουμε τον χρόνο προσπέλασης της μνήμης στην gpu.

Πιο συγκεκριμένα, επειδή τα threads τρέχουν παράλληλα, θέλουμε σε κάθε clock cycle το κάθε thread να διαβάζει τα δεδομένα που θέλει απευθείας και ταυτόχρονα με τα άλλα threads από την μνήμη του block. Με row-based indexing δεν το καραφέρνουμε αυτό, γιατί διπλανές θέσεις μνήμης αφορούν ένα object/cluster και επομένως η μνήμη του block μπορεί να μην έχει διαθέσιμα δεδομένα για όλα τα threads. Αντίθετα, με column-based indexing διπλανές θέσεις μνήμης αφορούν διαφορετικά objects και φορτώνονται στην μνήμη ίδιες συντεταγμένες για διαφορετικά objects. Αυτό είναι βέλτιστο καθώς τρέχουν παράλληλα υπολογισμοί για πολλά αντικείμενα με την ίδια συντεταγμένη για κάθε clock cycle.

Δεν παρατηρούμε κάποια διαφορά με το naive version σε ό,τι αφορά την κλιμάκωση ως προς το block size. Όμως, η επιτάχυνση είναι σαφώς βελτιωμένη. Παρακάτω φαίνονται τα διαγράμματα στα οποία μπορούμε να παρατηρήσουμε την ομοιότητα της καμπύλης με την naive και την διαφορά της στο πλάτος.
===insert threads per block explanation======



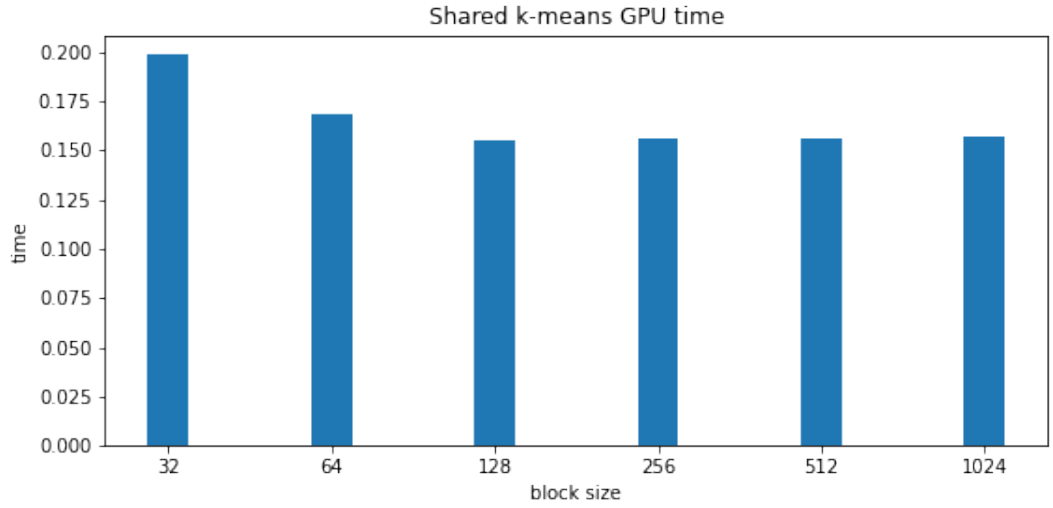


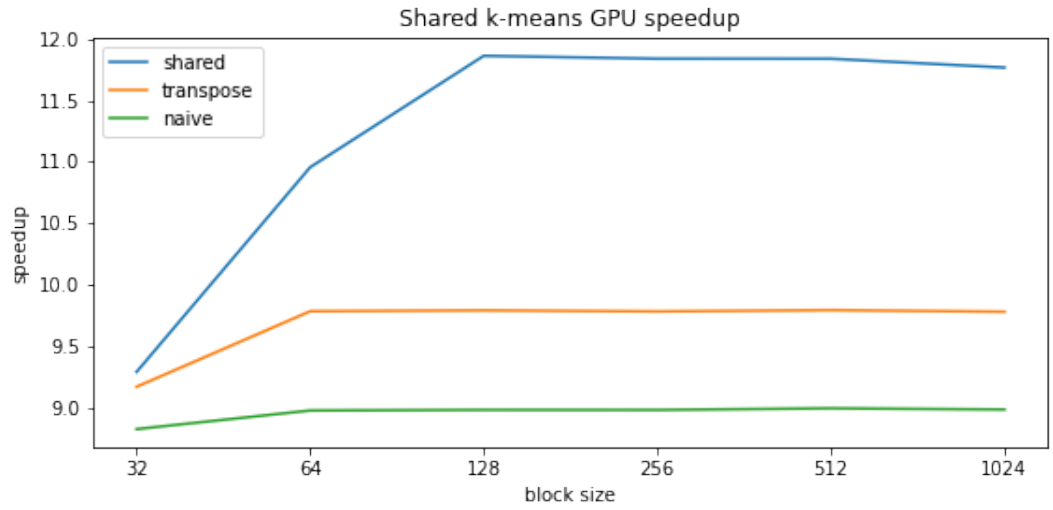
4 Shared Version

Σε αυτή την έκδοση τοποθετούμε τα clusters στην διαμοιραζόμενη μνήμη της GPU ώστε τα στοιχεία κάθε Thread Block να έχουν πιο γρήγορη πρόσβαση σε αυτά. Έτσι, το κάθε block μπορεί να φέρει μια φορά τα δεδομένα από την κύρια μνήμη στη διαμοιραζόμενη και έτσι να γλιτώσουμε πολλά memory accesses.
Αναφορικά στο μέγεθος της κοινής μνήμης, κάθε Thread Block απαιτεί μνήμη ίση με dimensions*numClusters*sizeof(float), το οποίο ισούται με 128 bytes για υλοποίηση με dimensions=2 και numClusters=16.

Πέραν της επιτάχυνσης που περιγράφηκε, μπορούμε να βελτιστοποιήσουμε την υλοποίησή μας, φροντίζοντας να γίνει πραγματικά παράλληλη η φόρτωση των δεδομένων στην κοινή μνήμη, όπως αυτή περιγράφεται στο for loop: ”for (int i=threadIdx.x; i<numClusters; i+=blockDim.x)” Αυτό είναι εφικτό σε περίπτωση που έχω Thread Block Size >= numClusters, το οποίο για την αναφερόμενη περίπτωση είναι 16. Σε κάθε περίπτωση εμείς χρησιμοποιούμε έτσι κι αλλιώς πολλαπλάσια του 32, καθώς αυτό είναι το cuda warp size και προτείνεται για βέλτιστη απόδοση. Σημειώνουμε ότι το ορισμένο shared memory size αποτελεί το ”κάτω όριο” της απαιτούμενης διαμοιραζόμενης μνήμης που θα χρησιμοποιήσει το κάθε block. Στην πραγματικότητα όταν αυξάνουμε τα Threads per Block, απαιτούνται παραπάνω πόροι και άρα ο πυρήνας μας θα απαιτεί συνολικά μεγαλύτερα ποσά shared memory για ναποθήκευση ενδιάμεσων τιμών κλπ.

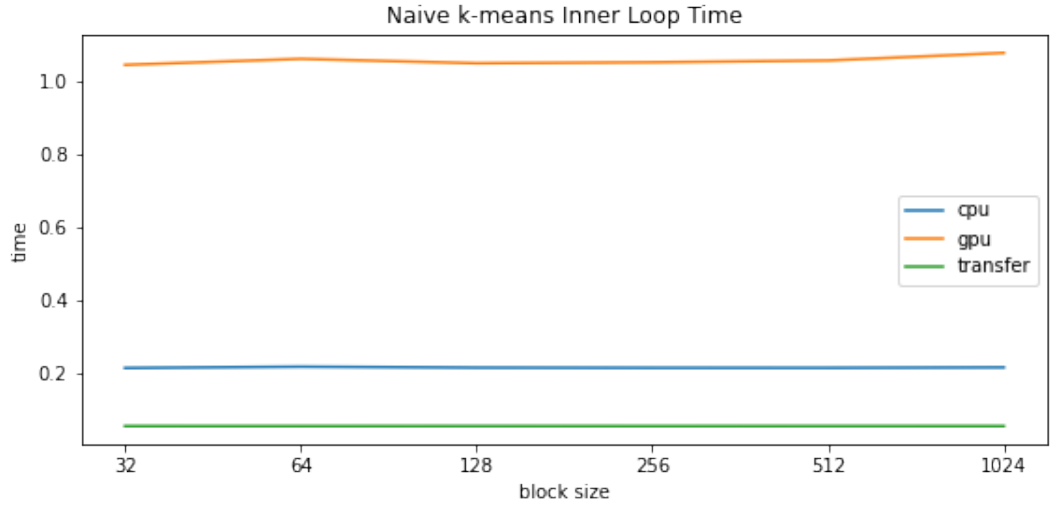
Στην παρούσα υλοποίηση, παρατηρούμε αντίστοιχης μορφής speedup όπως και στην transposed, με τις διαφορές ότι, στην περίπτωσή μας, το speedup κλιμακώνεται μέχρι τα 128 threads per block, ενώ είναι καλύτερο για κάθε αντίστοιχη τιμή threads per block.
Αυτό αιτιολογείται καθώς η χρήση κοινής μνήμης μειώνει τα data access bottlenecks που παρατηρούσαμε πριν, κάτι που βελτιώνει την επίδοση για μεγαλύτερες τιμές threads per block (σε αντίθετη περίπτωση θα είχα περισσότερα παράλληλα φορτώματα στη μνήμη με αποτέλεσμα να έχω καθυστερήσεις). Παρόλαυτά περισσότερα thread blocks προκαλούν μεγαλύτερη χρήση κοινής μνήμης, το οποίο εξηγεί γιατί δεν έχω κλιμάκωση περαν του 128.

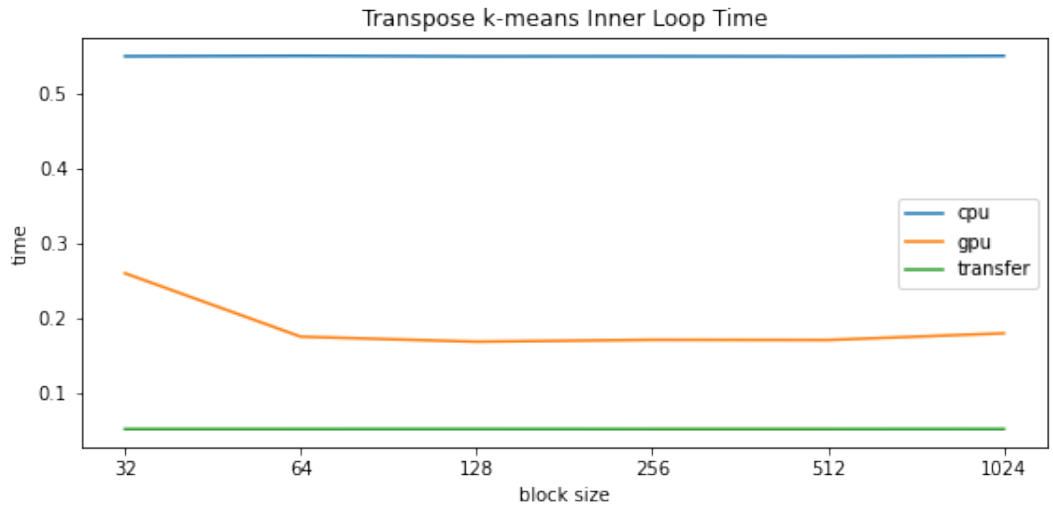


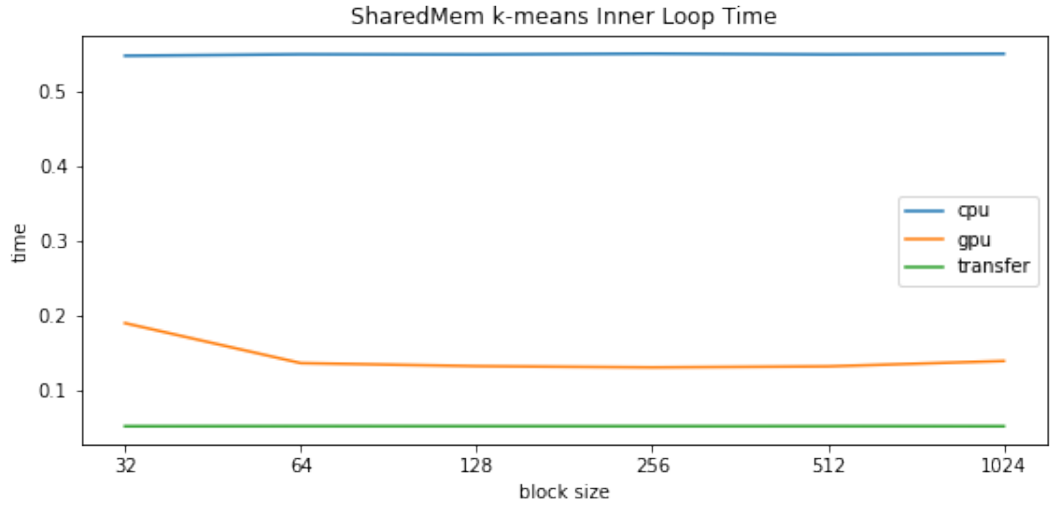


5 Σύγκριση υλοποιήσεων / bottleneck Analysis

Σε αυτό το σημείο συγκρίνουμε τις 3 υλοποιήσεις ως προς τους χρόνους εκτέλεσης της CPU, της GPU καθώς και τον χρόνο μεταφοράς δεδομένων μεταξύ της GPU και CPU. Δοκιμάσαμε configuration Size, Coords, Clusters, Loops = 256, 16, 16, 10 για block_size = 32, 64, 128, 256, 512, 1024 για όλες τις υλοποιήσεις. Παρακάτω παρουσιάζονται τα διαγράμματα των αποτελεσμάτων.







Στην περίπτωση του Naive παρατηρούμε πως το bottleneck το προκαλεί η GPU η οποία έχει το σημαντικότερο μερίδιο του χρόνου εκτέλεσης, σε αντίθεση με την Transpose version, όπου είναι σημαντικά μειωμένος. Όπως εξηγήσαμε και παραπάνω, με το column-based indexing μειώνουμε τα memory fetches και μεγιστοποιούμε τον παραλληλισμό των blocks μέσα στα SMs.

Πιο συγκεκριμένα, σχετικά με την σύγκριση των δύο versions, ο CPU χρόνος της Naive είναι μικρότερος από της Transpose. Αυτό είναι απολύτως λογικό άλλωστε, εφόσον στην περίπτωση του transpose φορτώνουμε την CPU με την ”εργασία” του μετασχηματισμού των στηλών σε γραμμές δύο φορές: πριν σταλθούν τα δεδομένα στην GPU και αφότου έχουν σταλθεί τα αποτελέσματα από την GPU.

Τώρα, σε ό,τι αφορά την SharedMem version παρατηρούμε πως ο χρόνος της CPU δεν έχει μεταβληθεί. Το μόνο που έχει αλλάξει είναι πως ο χρόνος εκτέλεσης μέσα στην GPU έχει μειωθεί σε σχέση με την transpose έκδοση κατά 0.1 δευτερόλεπτο περίπου, που είναι αναμενόμενο άλλωστε λόγω της γρηγορότερης πρόσβασης στα δεδομένα.

Σε αυτό το σημείο, πρέπει να αναφέρουμε πως η κλιμάκωση βάσει τον αριθμό του block size φαίνεται να επηρεάζεται μόνο από τον χρόνο εκτέλεσης μέσα στην GPU και οι χρόνοι μεταφοράς και CPU δεν επηρεάζουν καθόλου στο speedup. Επίσης, για όλες τις εκδόσεις του αλγορίθμου, οι χρόνοι μεταφοράς είναι σχεδόν ίδιοι, το οποίο είναι και λογικό εφόσον σε όλες τις περιπτώσεις μεταφέρεται ο ίδιος όγκος δεδομένων.