3η εργασία στα παράλληλα και διανεμημένα συστήματα

Connected components label propagation σε GPU

Μιχαήλ Μπαλτζάκης 10989

Link: https://github.com/MixalisBaltzakis/commit_for_third_exercise.git

Στην εργασία αυτή ασχοληθήκαμε με το πρόβλημα connected components label propagation σε GPU. Έχει γίνει εφαρμογή και χρονομέτρηση του αλγορίθμου με τις εξής 3 τεχνικές παραλληλοποίησης: Thread per row, Warp per row και Block per row.

Thread per row: Για κάθε κόμβο χρησιμοποιούμε ένα μόνο thread για να διαβάσουμε τους γείτονές του και να ενημερώσουμε τη τιμή του.

Warp per row: Για κάθε κόμβο χρησιμοποιούμε 32 threads, που είναι ίσο με το μέγεθος ενός warp, για να διαβάσουμε τους γείτονές του και να ενημερώσουμε τη τιμή του.

Block per row: Για κάθε κόμβο χρησιμοποιούμε block_dim threads, που είναι ίσο με το μέγεθος ενός block, για να διαβάσουμε τους γείτονές του και να ενημερώσουμε τη τιμή του.

Στα benchmarks έχουμε επιλέξει να τρέξουμε και τις 3 τεχνικές με 128 threads per block έτσι ώστε να έχουμε καλή απόδοση αλλά και σαφή διαχωρισμό των τεχνικών. Θέλουμε να εξασφαλίσουμε την πλήρη αξιοποίηση των πόρων της GPU. Πρέπει τα threads που τρέχουν παράλληλα να είναι πάντα στο μέγιστο δυνατό και ταυτόχρονα όμως να έχουμε σαφή διαχωρισμό των τεχνικών 2 και 3 γιατί αν μειώναμε το threads per block στο 64 οι δύο τεχνικές θα άρχιζαν να συγκλίνουν σε μία. Κάτι μεγαλύτερο από το 128 π.χ 256 είναι λίγο υπερβολή για τα δεδομένα μας καθώς στους περισσότερους γράφους δεν έχουμε μέσο όρο γειτόνων που να ξεπερνάει το 100. Έχουμε και τον περιορισμό της RAM στο Collab που δε μας αφήνει να τρέξουμε μεγάλους social graphs.

<u>Δομή προγράμματος:</u>

Βασικές συναρτήσεις:

CSCMatrix *load_problem_serial: Φορτώνει τον γράφο διαβάζοντας 2 binary αρχεία και ένα .txt που περιέχουν τους row pointers τους column indices και τους αριθμούς των node_number και nnz. Τα binary έχουν προκύψει από την επεξεργασία του .mat αρχείου του suitesparse με ένα python script. (Στο πρόβλημά μας επειδή έχουμε undirected graphs το CSC ταυτίζεται με το CSR).

sort_unique_int: Χρησιμοποιείται στο τέλος για να δούμε πόσες unique ετικέτες περιέχει το labels. Η ταύτιση του αριθμού των ετικετών με το πλήθος των SCC (από

το SuiteSparse) αποτελεί ισχυρή ένδειξη για την ορθότητα του αλγορίθμου και την απουσία λογικών σφαλμάτων. Η sort unique int χρησιμοποιεί για ταξινόμηση τη βοηθητική radixSort που έχουμε ορίσει για να πετύχει μια καλή ταχύτητα.

Kernels:

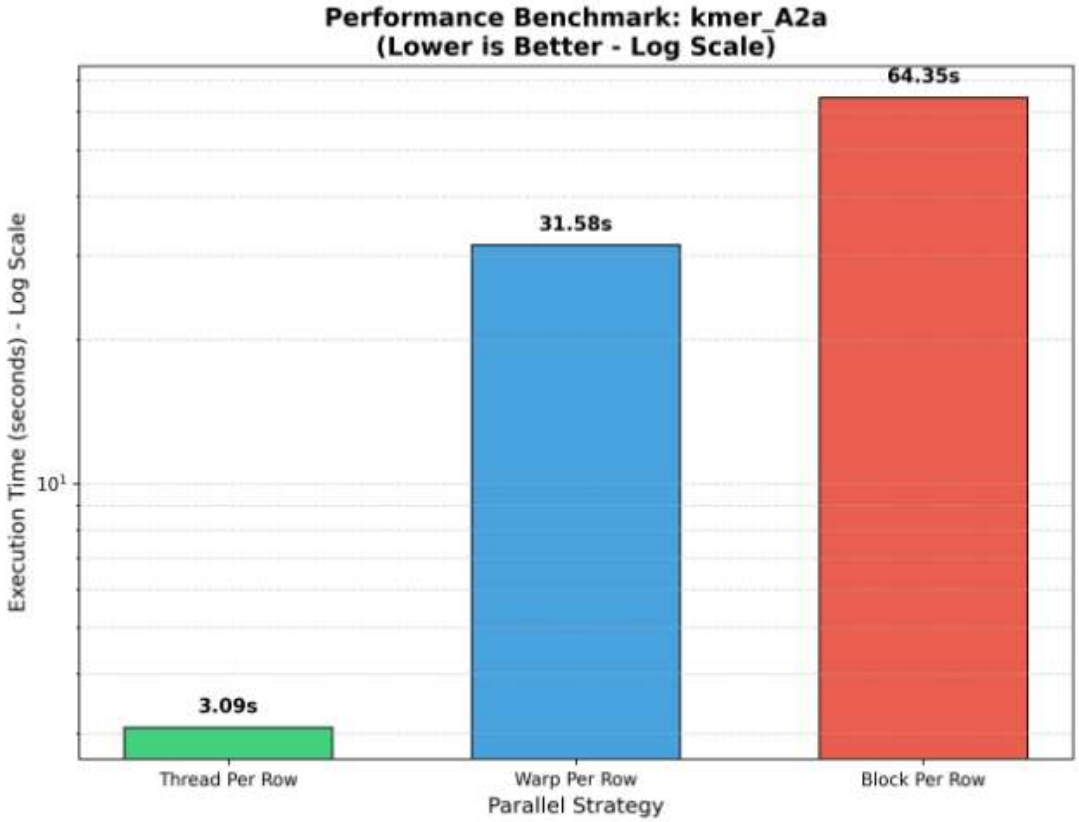
Thread_per_row_kernel: Είναι ο kernel που τρέχει στη GPU. Στη main ξεκινάμε τόσα threads όσα τον αριθμό των κόμβων του γράφου μας χωρίζοντάς τα σε blocks. Ο kernel ξεκινάει βρίσκοντας το global id του κάθε thread από τον τύπο global_id = blockDim.x * blockIdx.x + threadIdx.x; και έπειτα αυτό το id το αντιστοιχίζει σε ένα συγκεκριμένο κόμβο. Όλα τα threads των block που έχουν τη δυνατότητα να τρέξουν παράλληλα υπολογίζουν ταυτόχρονα το καθένα τον κόμβο του. Σημαντικό σημείο είναι ότι όλος ο κώδικας είναι εμφωλευμένος σε μια if(global_id < node_number) γιατί στη main ξεκινάνε μερικά παραπάνω threads από το node_number τα οποία αγνοούμε.

Warp_per_row_kernel: Εδώ στη main ξεκινάμε συνολικά WARP_SIZE(32) * node_number threads συνολικά. Τα threads χωρίζονται σε blocks όπου κάθε block είναι υπεύθυνο για threads_per_block/ WARP = 4 κόμβους στη περίπτωσή μας. Κάθε Thread βρίσκει το global id του, το global warp του άρα σε ποια ομάδα που υπολογίζει ποιο κόμβο ανήκει καθώς και τη θέση του μέσα στο warp του (0-31). Αν έχουμε πάνω από 32 γείτονες τα thread του warp τους διαβάζουν κυκλικά. Στη συνέχεια γίνεται warp reduction σε επίπεδο registers και έτσι αποφασίζεται το ολικό ελάχιστο από τα τοπικά ελάχιστα των 32 threads. Τέλος μόνο το πρώτο thread του warp γράφει στο labels το ολικό ελάχιστο (γιατί μόνο εκείνο το έχει) και αλλάζει την κατάσταση εξόδου. Και εδώ κάνουμε if (warp_id >= node_number) return; ώστε να αγνοήσουμε τα περίσσια warps.

Block_per_row_kernel: Σε αυτή τη παραλλαγή αφιερώνουμε ένα ολόκληρο block σε κάθε κόμβο. Οπότε συνολικά στη main θα έπρεπε να ξεκινήσουμε node_number blocks. Επειδή όμως οι γράφοι που εργαζόμαστε έχουν δεκάδες εκατομμύρια κόμβους κάτι τέτοιο μπορεί να αποβεί μοιραίο για τη ταχύτητα καθώς πάρα πολλή ώρα αφιερώνεται στον scheduler ώστε να οργανώσει τα blocks παρά στο ωφέλιμο έργο της GPU. Μία λύση σε αυτό είναι το Grid Stride. Στη Main υπολογίζουμε τους SMs της κάρτας μας και ξεκινάμε num_SMs * 8 blocks (sweet spot) έτσι, αν τα warps ενός block καθυστερήσουν, γίνεται ακαριαία αλλαγή (Zero-cost switching) σε warps άλλου block που βρίσκεται ήδη στον SM), διατηρώντας την GPU πάντα στο μέγιστο φόρτο. Έτσι πλέον τα block που έχουμε υπολογίζουν τους κόμβους κυκλικά και όχι μόνο ένα το καθένα. Στον kernel ξεκινάμε δηλώνοντας ως extern τον πίνακα της shared memory που θα χρησιμοποιήσουμε ώστε να περάσουμε το μέγεθός του ως όρισμα. Έπειτα βρίσκουμε το local id του thread, τη θέση του warp μέσα στο block και τη θέση του Thread μέσα στο warp. Όλα αυτά τα μεγέθη είναι τοπικά για κάθε

block. Η εύρεση του ολικού ελαχίστου τώρα χωρίζεται σε 2 μέρη: την εύρεση του τοπικού ελαχίστου του κάθε warp του block και στο επόμενο βήμα την εύρεση του ολικού ελαχίστου συγκρίνοντας τα τοπικά ελάχιστα των warps. Στο πρώτο βήμα όπως και πριν κάνουμε warp reduction με αναδιπλώσεις σε επίπεδο καταχωρητών για κάθε warp του block μας αφού πρώτα τα threads του block έχουν διαβάσει τους γείτονες κυκλικά. Στο τέλος της διαδικασίας αυτής το πρώτο Thread κάθε warp έχει το τοπικό ελάχιστο το οποίο και γράφει στη shared memory. Στη συνέχεια **μόνο το πρώτο warp** με το τέχνασμα: block_min = (tid < num_warps) ? s_warp_mins[tid] : INT_MAX; Δίνει στα πρώτα num_warps(αριθμός warps ανά block) threads από ένα τοπικό ελάχιστο στο καθένα ενώ όλα τα υπόλοιπα(32-num_warps) έχουν μία ψεύτικη τιμή INT_MAX ώστε να μην επηρεάζει στη σύγκριση(στο πεδίο block_min πάντα). Έτσι γίνεται πάλι ένα warp reduce και το πρώτο thread του πρώτου warp του block καταλήγει με το ολικό ελάχιστο. Όλη αυτή η διαδικασία γίνεται κυκλικά από τα block που έχουμε εκκινήσει. Πριν αλλάξουμε κόμβο που υπολογίζει κάποιο block απαιτείται συγχρονισμός όπως και πριν διαβάσουμε τη shared memory.

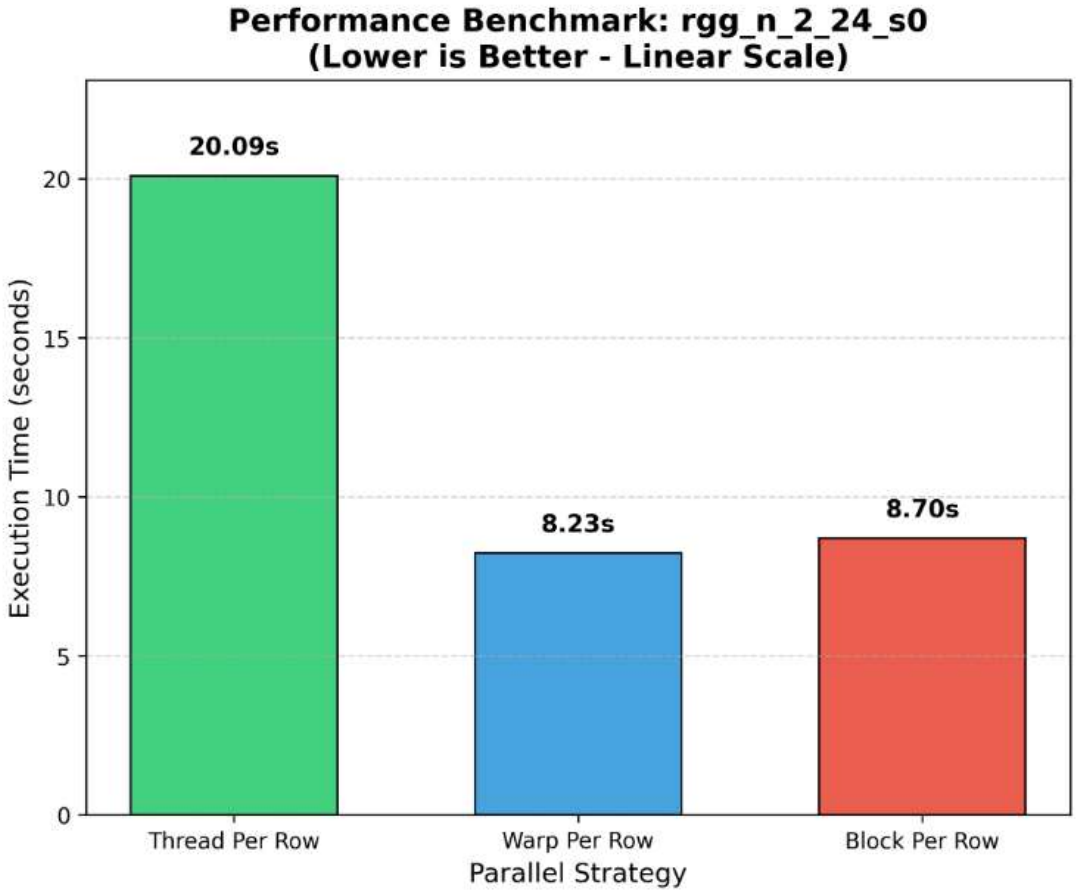
Διάγραμμα απόδοσης του kmer_A2a:



Στο διάγραμμα παρατηρούμε τη δύναμη του thread per row σε γράφους που έχουν πολύ μικρό μέσο αριθμό γειτόνων και δεν έχουν κόμβους με χιλιάδες γείτονες ενώ άλλους με πολύ λίγους(αν συμβαίνει το δεύτερο ο thread per row είναι πολύ αδύναμος καθώς όλα τα threads του warp περιμένουν εκείνον τον ένα που έχει

ακόμη δουλειά να τελειώσει. Στις άλλες 2 τεχνικές το ξεπερνάμε αυτό το πρόβλημα καθώς όλο το warp εργάζεται πάνω στον ίδιο κόμβο) . Ο kmer_A2a (n=170.M,nnz=360M) έχει μέσο αριθμό γειτόνων λίγο μεγαλύτερο από 2. Από την άλλη οι warp per row και block per row είναι απαγορευτικοί σε τέτοιους γράφους καθώς επιστρατεύουν warps και blocks των 32 και 128 threads αντίστοιχα για να διαβάσουν μέσο όρο 2 γείτονες. Έτσι σχεδόν όλα τα threads περιμένουν μερικά να ενημερώσουν τη τιμή του κόμβου δίχως εκείνα να παράγουν ωφέλιμο έργο σε ποσοστά αξιοποίησης προσεγγιστικά 2/32 και 2/128 αντίστοιχα. Για αυτό προκύπτει και η πολύ μεγάλη απόκλιση στους χρόνους που βλέπουμε.

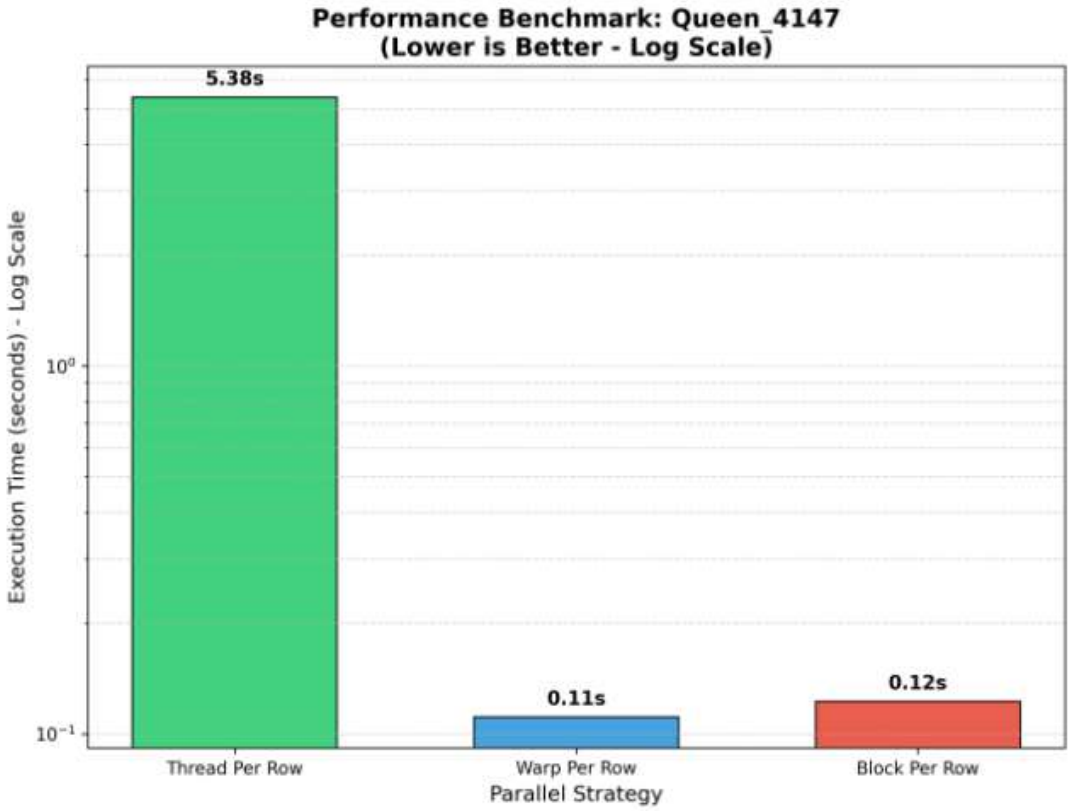
Διάγραμμα απόδοσης του rgg_n_2_24_s0:



Στην περίπτωση του rgg_n_2_24_s0 (n=16.7M, nnz=265M) έχουμε διαφορετικά αποτελέσματα. Πρόκειται για έναν ισορροπημένο γεωμετρικό γράφο χωρίς μεγάλα hubs, όπου όλοι οι κόμβοι έχουν περίπου ίσο αριθμό γειτόνων και κυριαρχούν οι τοπικές συνδέσεις. Ο μέσος αριθμός γειτόνων είναι 16, σαφώς μεγαλύτερος από το 2 που είχαμε στον προηγούμενο γράφο, γεγονός που επιτρέπει στις μεθόδους Warp per Row και Block per Row να είναι 2.5x γρηγορότερες. Αντίθετα, η Thread per Row υποφέρει από Non-Coalesced Memory Access. Επειδή κάθε thread αναλαμβάνει διαφορετικό κόμβο, ζητούνται ταυτόχρονα διάσπαρτες διευθύνσεις μνήμης. Καθώς η GPU μεταφέρει δεδομένα σε πακέτα (transactions) των 128 bytes, σπαταλάται τεράστιο εύρος ζώνης για να μεταφερθούν λίγα μόνο χρήσιμα δεδομένα ανά

πακέτο, προκαλώντας συμφόρηση. Οι άλλες δύο μέθοδοι, εκμεταλλευόμενες τη συνεργασία των threads και το ότι στη δομή CSR οι γείτονες είναι συνεχόμενοι στη μνήμη ομαδοποιούν τις προσβάσεις και μειώνουν δραματικά τα απαιτούμενα transactions.

Διάγραμμα απόδοσης του Queen_4147:



Εδώ γίνεται ακόμη πιο αισθητή η αδυναμία της μεθόδου Thread per Row σε γράφους με υψηλότερο μέσο βαθμό. Ο Queen_4147 (n=4.1M, nnz=316M) έχει μέσο βαθμό περίπου 80 και δομή πλέγματος (mesh). Σε αυτή την περίπτωση, πέρα από το πρόβλημα του Non-Coalesced Memory Access που παραμένει, προκύπτει και σοβαρό θέμα latency hiding. Οι 80 επαναλήψεις που πρέπει να εκτελέσει σειριακά κάθε thread είναι υπερβολικές για να "κρυφτούν" μέσω της εναλλαγής threads (context switching), καθώς το workload ανά thread είναι πολύ βαρύ(ο κυκλος αιτημα → αναμονη → switch → επαναφορα στον SM πρέπει να γίνει πολλές φορές οδηγώντας τον scheduler σε κορεσμό , καθώς ξεμένει από διαθέσιμα threads και όλα βρίσκονται σε κατάσταση αναμονής).Αντίθετα, στις άλλες δύο τεχνικές, οι ομάδες των 32 (Warp) και 128 (Block) threads συνεργάζονται για τον ίδιο κόμβο, μοιράζοντας το φορτίο. Κάθε thread καταλήγει να διαβάζει ελάχιστους γείτονες (2-3), επιτρέποντας την ακαριαία ολοκλήρωση της επεξεργασίας χωρίς συμφόρηση. Το αποτέλεσμα είναι η δραματική επιτάχυνση της τάξεως του 50x.