Εργασία 3.5: CUDA

Υλοποίηση αλγορίθμων APSP

Κωνσταντίνος Σαμαράς-Τσακίρης

29 Φεβρουαρίου 2016

Στόχος

Η υλοποίηση 3 πυρήνων CUDA που να επιλύουν το πρόβλημα APSP:

1. Με τον πιο απλό τρόπο
2. Με χρήση κοινής μνήμης
3. Με επεξεργασία >1 στοιχείων ανά νήμα

Σχόλια

Ο κώδικας βρίσκεται στο Github: https://github.com/Oblynx/parallel-course-projects/tree/p3. Η αναφορά[1] βρίσκεται εδώ: http://repository.upenn.edu/cgi/viewcontent.cgi?article=1213&context=hms

1 Μέθοδος παραλληλοποίησης

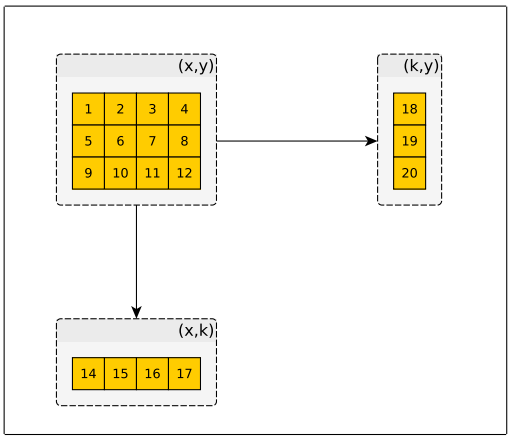
1.1 Απλή ("simple_gpu")

Ο σειριακός αλγόριθμος Floyd-Warshall είναι φημισμένης απλότητας (βλ. Dijkstra) και η πρώτη προσέγγιση παραλληλοποίησής του είναι εξίσου απλή. Ο εξωτερικός βρόχος παραμένει σειριακός και παραλληλοποιείται η πρόσβαση στα στοιχεία του πίνακα σε κάθε βήμα. Κάθε στοιχείο αντιστοιχείται σε ένα νήμα, το οποίο εκτελεί ό,τι και ο σειριακός για συγκεκριμένη θέση στον πίνακα. Ο πυρήνας αυτός καλείται με το μέγιστο δυνατό block size, εν προκειμένω 32, και σε ένα πλέγμα μεγέθους ανάλογου του μεγέθους του προβλήματος.

1.2 Χρήση κοινής μνήμης ("block_gpu")

Μια πρώτη προσπάθεια βελτίωσης της πρόσβασης στην κύρια μνήμη εξετάζει τις θέσεις του πίνακα από τις οποίες καλούνται δεδομένα στην εκτέλεση του απλού πυρήνα, για να εντοπίσει πού η πρόσβαση δεν είναι συνενωμένη. Όπως φαίνεται στο σχήμα 1, αν θεωρήσουμε ότι ένα thread block βρίσκεται γύρω από τη θέση (x, y), θα κληθούν από τη μνήμη οι ομάδες δεδομένων (x, k) και (k, y). Τα δεδομένα στο (x, k) θα χρησιμοποιηθούν από κάθε επόμενο warp, άρα η τοπική τους αποθήκευση είναι πολύ σημαντική. Η πρόσβαση σε αυτή την περιοχή μνήμης όμως είναι εξ αρχής συνενωμένη.

Οι GPU με compute capability τουλάχιστον 2 υλοποιούν στο υλικό caching της κύριας μνήμης.



Σχήμα 1: Floyd-Warshall access pattern

Αυτό καλύπτει τις ανάγκες τοπικής φύλαξης των δεδομένων (x, k), άρα η χρήση της κοινής μνήμης θα πρέπει να στοχεύσει στην αποθήκευση των δεδομένων της περιοχής (k, y). Αν θεωρήσουμε ότι το πλάτος της περιοχής (x, y) είναι 32, τότε κάθε warp καλεί μονάχα 1 στοιχείο από την περιοχή (k,y), ενώ τα στοιχεία που βρίσκονται στις δεξιότερες στήλες, $(k+1, y), (k+2, y), ...$, και τα οποία η μνήμη θα μπορούσε να παράσχει στον ίδιο χρόνο, δε χρησιμοποιούνται παρά μόνο στον επόμενο πυρήνα.

Η παρατήρηση αυτής της δομής προκαλεί το ερώτημα σχεδίασης ενός εναλλακτικού αλγορίθμου που να επεξεργάζεται τα δεδομένα του πίνακα σε στάδια και σε μικρές κάθε φορά περιοχές που εφεξής θα καλούνται **πλακίδια**.

Block algorithm

Εντόπισα έναν τέτοιο ακριβώς αλγόριθμο στο άρθρο[1] που παρέχεται μαζί με την εργασία (apsp_blockalgo.pdf), επομένως εδώ δε θα τον περιγράψω αναλυτικά.

Συνοπτικά, ο αλγόριθμος χωρίζει τον πίνακα σε B πλακίδια μεγέθους n και εφαρμόζει τα εξής για B επαναλήψεις:

1. Φάση 1: Επίλυση του προβλήματος APSP στο τρέχον (κύριο) πλακίδιο, δηλαδή εντοπισμός των ελάχιστων μονοπατιών που δεν ξεφεύγουν από τα όρια του πλακιδίου. Μέγεθος πλέγματος: 1×1
2. Φάση 2: Για κάθε πλακίδιο στην ίδια στήλη ή γραμμή με το κύριο, εξεύρεση των ελάχιστων μονοπατιών που χρησιμοποιούν αποκλειστικά κόμβους του κυρίου πλακιδίου ως ενδιάμεσους. Μέγεθος πλέγματος: $(B-1) \times 2$
3. Φάση 3: Για όλα τα υπόλοιπα πλακίδια, εξεύρεση των ελάχιστων μονοπατιών που χρησιμοποιούν αποκλειστικά κόμβους του κυρίου πλακιδίου ως ενδιάμεσους. Μέγεθος πλέγματος: $(B-1) \times (B-1)$

1.3 Πολλαπλά στοιχεία ανά νήμα ("multi_xy", "multi_y")

Πάνω στον προηγούμενο αλγόριθμο δοκιμάστηκαν 2 προσεγγίσεις τέτοιας μορφής. Κάθε νήμα (x, y) αναλαμβάνει τα στοιχεία:

1. $(2x, 2y), (2x+1, 2y), (2x, 2y+1), (2x+1, 2y+1)$, δηλαδή ένα τετράγωνο 2×2 στοιχείων
2. $(x, 2y), (x, 2y+1)$, δηλαδή ένα ορθογώνιο 2×1 στοιχείων

Ο αλγόριθμος έμεινε ολόιδιος, με εξαίρεση την αλλαγή του block size. Στην πρώτη περίπτωση και οι 2 διαστάσεις του υποδιπλασιάστηκαν, ενώ στη 2η μόνο η διάσταση y.

2 Έλεγχος ορθότητας

Με δεδομένη την απλότητα του σειριακού αλγορίθμου, ο έλεγχος ορθότητας πραγματοποιείται επιλύοντας το πρόβλημα σειριακά και συγκρίνοντας το αποτέλεσμα κάθε παράλληλου αλγορίθμου με αυτό. Σε περίπτωση που βρεθεί σφάλμα, το πρόγραμμα ειδοποιεί και σταματά. Ο έλεγχος ορθότητας και η χρονοβόρα σειριακή επίλυση γενικότερα μπορούν να αποφευχθούν με τον ορισμό της παραμέτρου "NO_TEST" κατά τη μεταγλώττιση.

3 Μετρήσεις

Μέγεθος προβλήματος

Η τάξη πολυπλοκότητας του αλγορίθμου είναι $O(n^3)$. Στις πειραματικές μετρήσεις χρόνου επιβεβαιώνεται η έντονη μη γραμμικότητα. Σε μέγεθος προβλήματος 2^{12} ο απλός αλγόριθμος GPU έχει επιτάχυνση $\times 3$,

ενώ οι κατά πλακίδια βελτιώνουν πάνω σε αυτό σε βαθμό $\times 3.5$, με συνολική επιτάχυνση $\times 9.5$ προς το σειριακό αλγόριθμο. Σε μεγεθος 2^{13}, η επιτάχυνση του απλού αλγορίθμου GPU είναι $\times 2$, των πλακιδίων σε σχέση με τον απλό $\times 3.5 - \times 4$ και των πλακιδίων σε σχέση με το σειριακό $\times 7.5$.

Το άρθρο[1] που προτείνει τον αλγόριθμο δηλώνει επιτάχυνση $\times 60 - \times 130$ σε σχέση με μια απλή σειριακή υλοποίηση, αλλά μόνο $\times 2 - \times 4$ απέναντι σε μια καλά ρυθμισμένη υλοποίηση CPU. Ενώ ως προς μια απλή υλοποίηση GPU δηλώνουν επιτάχυνση $\times 5 - \times 6.5$, που είναι σχετικά κοντά σε αυτό που πετυχαίνει τούτη η υλοποίηση. Πιστεύω πως η τάξης μεγέθους διαφορά στην επιτάχυνση ως προς το σειριακό αλγόριθμο σε σχέση με το άρθρο οφείλεται στη διαφορά στην ποιότητα του υλικού ανάμεσα στο CPU και GPU σε αυτόν τον υπολογιστή, υπόθεση που δικαιολογεί τη σχετικά κοντινή τιμή επιτάχυνσης ανάμεσα στον απλό και με πλακίδια αλγόριθμο GPU.

Block size effect

Στα 2 πειράματα που φαίνονται στο πλαίσιο 3 οι αλγόριθμοι πολλαπλών στοιχείων ανά νήμα εμφανίζουν διαφορετική συμπεριφορά – στο run1 είναι πιο αργοί από τον απλό αλγόριθμο κατά πλακίδια, ενώ στο run2 συμβαίνει το αντίστροφο. Αυτό οφείλεται στο διαφορετικό block size που δοκιμάστηκε για αυτούς: ενώ το block size του "block_gpu" παραμένει σταθερό στο 16×16, οι "multi_xy" και "multi_y" στο run1 έχουν block size 16×16, ενώ στο run2 έχουν 32×32.

Τα τελικά block sizes που επιλέχθηκαν, 32×32 για τους αλγορίθμους "multi" και 16×16 για το "block_gpu" προέκυψαν μετά από πειράματα και ανάλυση με τον NVidia Visual Profiler. Είναι βέλτιστα μόνο για την κάρτα στην οποία δοκιμάστηκαν.

Συνδεσιμότητα γράφου

Οι μετρήσεις δείχνουν ότι το ποσοστό συνδέσεων στο γράφο δεν επηρεάζει την ταχύτητα εκτέλεσης του αλγορίθμου. Αυτό είναι αναμενόμενο, γιατί η μοναδική διαφορά ανάμεσα στην επίλυση ενός προβλήματος μικρής συνδεσιμότητας σε σχέση με ένα πρόβλημα μεγάλης είναι η συχνότητα εκτέλεσης μίας αποθήκευσης στη μνήμη. Ειδικά στην περίπτωση της χρήσης κοινής μνήμης, το κόστος αυτό ανάγεται σε 1 κύκλο μηχανής, είναι επομένως αμελητέο.

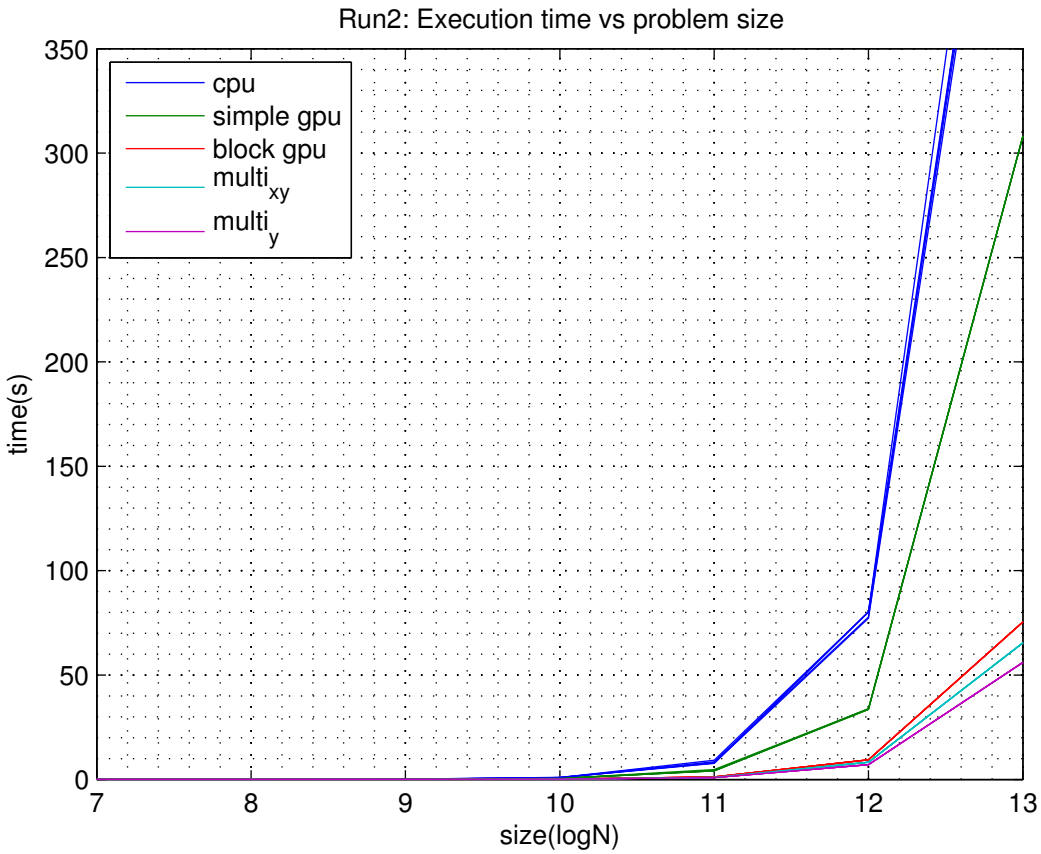
CPU	simpleGPU	blockGPU	multi_xy	multi_y
2.6049	0.0625	0.0070	0.0069	0.0069

Πίνακας 1: Τυπικές αποκλίσεις χρόνου εκτέλεσης για διαφορετικές συνδεσιμότητες γράφου

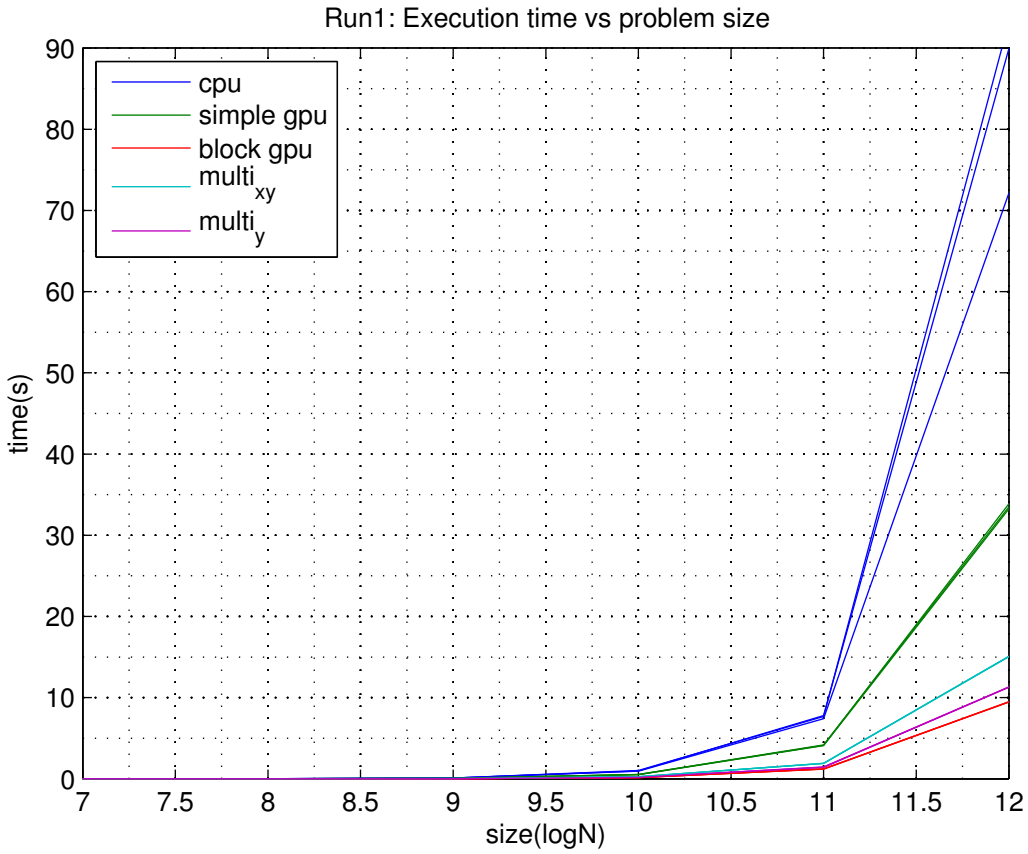
Ο αλγόριθμος Floyd-Warshall είναι προτιμητέος στους πυκνούς γράφους, όπου το πλήθος των ακμών είναι ανάλογο του N^2. Αντίθετα, αν το πλήθος των ακμών είναι ανάλογο του N (αραιός γράφος), το APSP μπορεί να επιλυθεί αποδοτικότερα με επαναληπτική εφαρμογή του αλγορίθμου του Dijkstra από κάθε κορυφή.

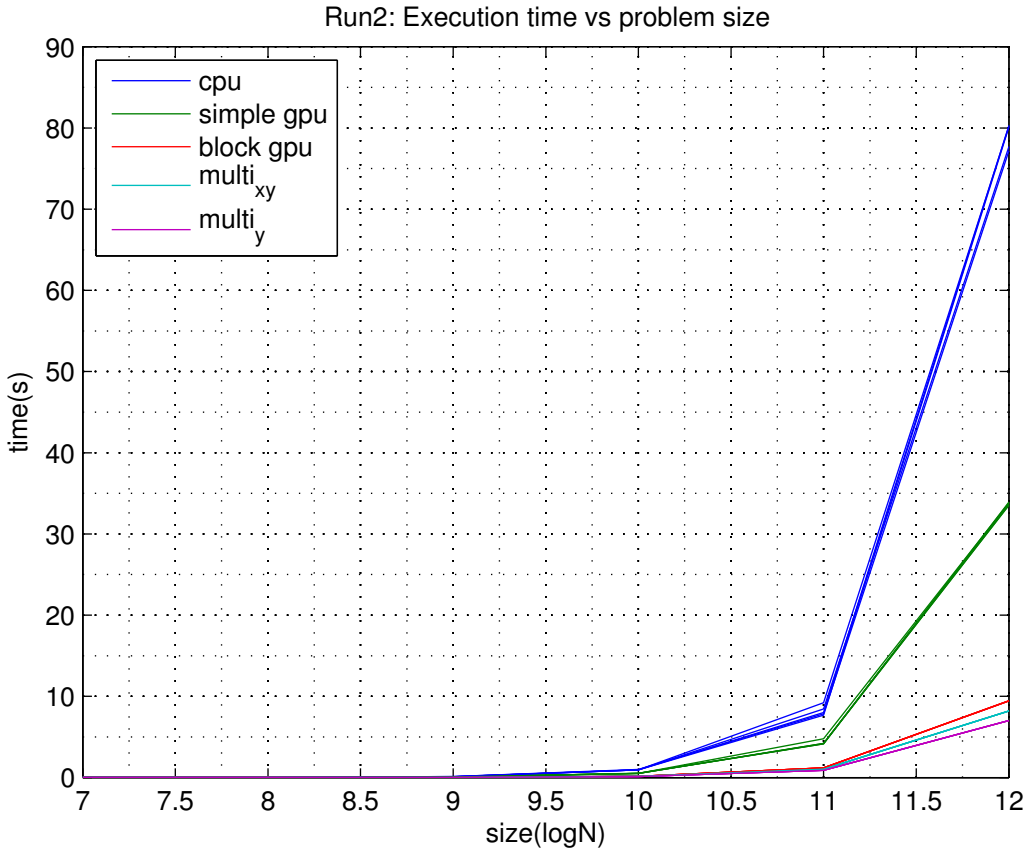
Αναφορές

[1] Katz, G. J., & Kider, J. T. (2008). All-Pairs Shortest-Paths for Large Graphs on the GPU. Proceedings of the 23rd ACM SIGGRAPH/ EUROGRAPHICS Symposium on Graphics Hardware (GH '08), 47-55. http://dx.doi.org/10.2312/EGGH/EGGH08/047-055



Σχήμα 2: Run2: Χρόνος εκτέλεσης προς μέγεθος προβλήματος





Σχήμα 3: Σύγκριση run1&2 στους ίδιους άξονες