ΑΡΙΣΤΟΤΕΛΕΙΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΟΝΙΚΗΣ

Τμημα ΗΜΜΥ. Τομεας Ηλεκτρονικης

Παραλληλα και Διανεμημενα Συστηματα

Εργασία 3: Bitonic sort with CUDA

Ανάλυση και υλοποίηση του αλγόριθμου bitonic sort για υποσύστημα γραφικών με χρήση CUDA

Συντάκτης:
Χρήστος Χουτουρίδης [8997]
cchoutou@ece.auth.gr

Διδάσκων:
Νικόλαος Πιτσιάνης
nikos.pitsianis@eng.auth.gr

20 Φεβρουαρίου 2025

1. Εισαγωγη

Η παρούσα εργασία αποτελεί συνέχεια της προηγούμενης υλοποίησης του ίδιου αλγόριθμου σε κατανεμημένα συστήματα με τη χρήση MPI. Στην τρέχουσα έκδοση ο αλγόριθμος καλείται να εκτελεστεί σε υποσυστήματα γραφικών (GPU) με χρήση CUDA. Τα υποσυστήματα γραφικών μας δίνουν την δυνατότητα για πολύ μεγάλο αριθμό παραλληλοποίησης, κάτι που αποτελεί σημαντική διαφοροποίηση σε σχέση με την MPI έκδοση. Πρακτικά ο τρόπος με τον οποίο χρησιμοποιείται το υποσύστημα γραφικών σε αυτή την εργασία είναι σαν accelerator συγκεκριμένων συναρτήσεων. Η υλοποίηση του αλγόριθμου έγινε σε τρεις φάσεις ακολουθώντας τις υποδείξεις της εκφώνησης. Στην πρώτη φάση, το υποσύστημα γραφικών απλώς επιταχύνει το κύριο σώμα του αλγόριθμου, ενώ στις επόμενες έγινε προσπάθεια τροποποίησης αυτού ώστε να εκμεταλλευτούμε τα χαρακτηριστικά και τους πόρους του υποσυστήματος γραφικών, με αποδοτικότερο τρόπο.

1.1. Παραδοτέα

Τα παραδοτέα της εργασίας αποτελούνται από:

- Την παρούσα αναφορά.
- Το σύνδεσμο με το αποθετήριο που περιέχει τον κώδικα για την παραγωγή των εκτελέσιμων, της αναφοράς και τις μετρήσεις.

2. Υλοποιηση

Πριν ξεκινήσουμε με τις λεπτομέρειες του αλγόριθμου, καλό θα ήταν να περιγράψουμε τις βασικές συναρτήσεις και δομές που χρησιμοποιούνται στην εργασία, ώστε να βοηθήσουμε στην καλύτερη κατανόηση της υλοποίησης.

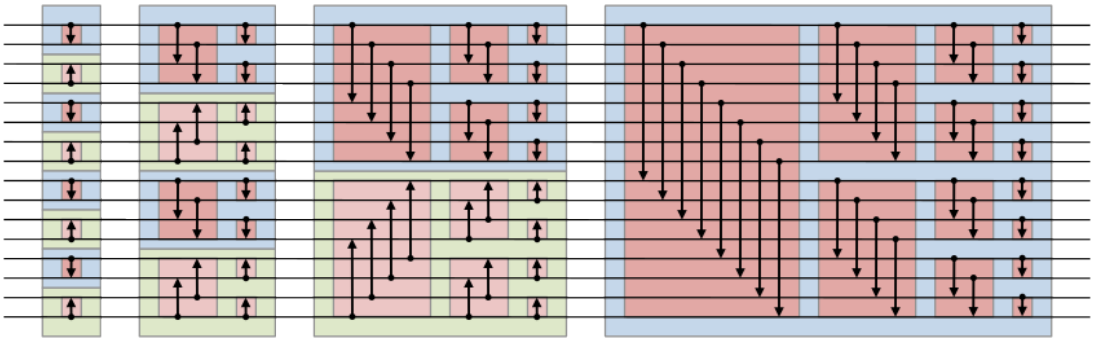
- **`Log`**: Πρόκειται για ένα τύπο ο οποίος μας δίνει logging δυνατότητες τις οποίες μπορεί ο χρήστης να ενεργοποιήσει από τη γραμμή εντολών με την παράμετρο `-v` ή `--verbose`.
- **`Timing`**: Η τάξη αυτή προσφέρει δυνατότητες χρονομέτρησης μίας ή και πολλαπλών κλήσεων αθροίζοντας τους χρόνους (accumulation). Επίσης προσφέρει δυνατότητες για πολλαπλές εκτελέσεις ολόκληρης της ταξινόμησης όπου οι χρόνοι μπορούν να ταξινομηθούν και να επιστραφεί ο ενδιάμεσος. Ο χρήστης μπορεί να επιλέξει την εμφάνιση των αποτελεσμάτων αλλά και τον αριθμό των εκτελέσεων από τη γραμμή εντολών με την παράμετρο `--perf`.
- **`bitonicSort()`**: Πρόκειται για την βασική συνάρτηση που υλοποιεί τον αλγόριθμο της εργασίας και είναι σε μορφή template για διαφορετικούς τύπους δεδομένων προς ταξινόμηση. Έχει υλοποιηθεί 3 φορές, μία για κάθε έκδοση της εκφώνησης V0, V1, V2 στο αρχείο **`bitonicsort.hpp`**.
- **`inBlockStep()/interBlockStep()`**: Ειδικά για τις εκδόσεις V1 και V2 όπου χρησιμοποιείται loop unrolling, το κύριο σώμα του αλγόριθμου bitonic sort έχει χωριστεί σε δύο συναρτήσεις. Μία που εκτελείτε εκτός του unrolling (*interBlockStep()*) και μία που εκτελεί το ίδιο το unrolling (*inBlockStep()*).

Όλοι οι παραπάνω τύποι είναι templates, και βρίσκονται στα αντίστοιχα headers (.hpp) αντί για αρχεία .cpp. Η συνάρτηση *main()* που υλοποιεί τον αλγόριθμο μαζί με τον validator και το command line interface βρίσκονται στο αρχείο main.cpp. Στο αρχείο config.h υπάρχουν οι compile time ρυθμίσεις της υλοποίησης όπου ο χρήστης μπορεί να επιλέξει για παράδειγμα τον τύπο δεδομένων προς ταξινόμηση.

2.1. Πρώτη έκδοση (V0)

Η πρώτη έκδοση υλοποιεί τον βασικό αλγόριθμο επιταχύνοντας απλώς το κύριο σώμα του στην κάρτα γραφικών. Καθώς θεωρούμε πως ο αναγνώστης είναι ήδη εξοικειωμένος με την διτονική ταξινόμηση, δεν θα αναλύσουμε την λειτουργία της σε μεγάλο βάθος. Για πίνακα μεγέθους 2^N ο αλγόριθμος συνοψίζεται στα εξής:

- Εκτελούνται N ακολουθίες ανταλλαγών με αύξον αριθμό: $m = 1, 2, \ldots, N$.
- Η κάθε ακολουθία m αποτελείται από ανταλλαγές μεταξύ γειτόνων, των οποίων η απόσταση ξεκινάει από 2^{m-1} και μειώνεται με διαδοχικές ακέραιες διαιρέσεις με το 2, εωσότου γίνει 1: $2^{m-1}, 2^{m-2}, ..., 1$.
- Οι ανταλλαγές χωρίζουν τα μεγαλύτερα και τα μικρότερα στοιχεία με στόχο στο τέλος της κάθε ακολουθίας τα στοιχεία του πίνακα να αποτελούν διαδοχικές διτονικές ακολουθίες. Μέτά την πρώτη ακολουθία ανταλλαγών να έχουμε $\frac{N}{2}$ διτονικές ακολουθίες, μετά την δεύτερη $\frac{N}{4}$ ακολουθίες και ούτω κάθε εξής, έως ότου στην τελευταία να έχουμε $\frac{N}{2N} = \frac{1}{2}$ διτονική, δηλαδή μια πλήρως ταξινομημένη λίστα.



Σχήμα 1: 4 ακολουθίες ανταλλαγών για ταξινόμηση 16 στοιχείων.

Η εικόνα 1 παρουσιάζει αυτή τη λογική[1].

2.1.1. Πλήθος απαιτούμενων διεργασιών (threads) και καταμερισμός στον πίνακα

Στην παρούσα εργασία, επιθυμούμε οι ανταλλαγές να λάβουν χώρα εντός του υποσυστήματος γραφικών και να εκτελεστούν από τις διεργασίες (threads) του υποσυστήματος γραφικών. Σε αντίθεση όμως με την προηγούμενη εργασία, όπου για M υποσύνολα του πίνακα είχαμε M MPI διεργασίες και όλες απαιτούνταν να εκτελέσουν διαχωρισμό μεγίστων – ελαχίστων, **εδώ η κάθε διεργασία μπορεί να διαχωρίσει 2 στοιχεία. Δηλαδή για ταξινόμηση ενός πίνακα N στοιχείων απαιτούνται $\frac{N}{2}$ διεργασίες**.

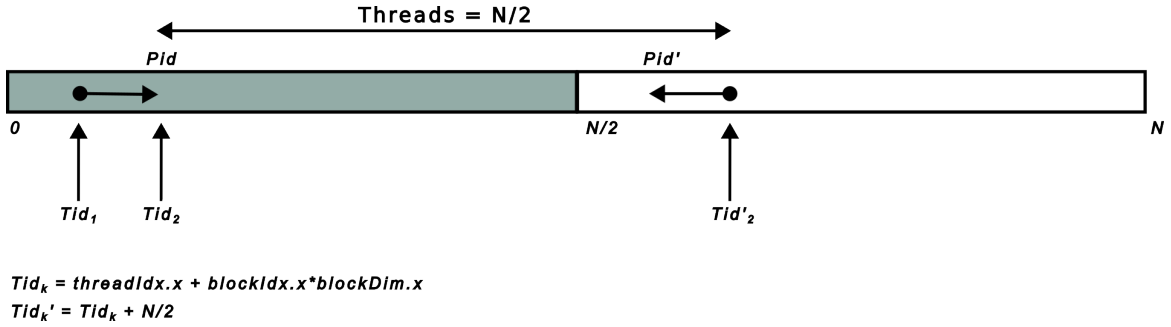
Βλέπουμε δηλαδή πως δεν είναι απαραίτητο να αναθέσουμε μία διεργασία σε κάθε στοιχείο του πίνακα. Με αυτό σαν βάση προβήκαμε στον πρώτο καταμερισμό των διεργασιών στα στοιχεία του πίνακα. Έτσι χωρίσαμε τον πίνακα σε δύο μέρη. Στο πρώτο μέρος η διευθυνσιοδότηση έγινε ακολουθώντας την αρίθμηση των threads, ενώ για το δεύτερο το διπλάσιο αυτής.

Όπως γίνεται φανερό *έπρεπε να βρεθεί ένας τρόπος ώστε οι μισές διεργασίες να διευθυνσιοδοτήσουν το πρώτο μέρος και οι άλλες μισές το δεύτερο*. Ο διαχωρισμός έγινε ελέγχοντας τον γείτονα (partner) για την κάθε ανταλλαγή.

Η λογική πίσω από αυτό τον έλεγχο είναι ότι αν ένα thread με διεύθυνση Tid_1 έχει να ανταλλάξει μέ έναν γείτονα με διεύθυνση $Pid = Tid_2$, τότε το thread που αρχικά πέφτει στη διεύθυνση Tid_2 έχει να ανταλλάξει με το Tid_1. Η ανταλλαγή όμως αυτή έχει δρομολογηθεί στο Tid_1 και έτσι το συγκεκριμένο thread μπορεί να μεταφερθεί στο δεύτερο μισό του πίνακα στην αντίστοιχη θέση Tid'_2 και να ανταλλάξει με τον γείτονα που προκύπτει σε εκείνη τη θέση Pid'. Ο γείτονας σε εκείνη τη θέση φυσικά είναι ο αντίστοιχος του Tid_1. Το διάγραμμα 2 παρουσιάζει αυτόν τον καταμερισμό.

Έχοντας κατανέμει λοιπόν τα threads στον πίνακα, η υλοποίηση (*bitonicSort()*) εκτελεί το διπλό βρόχο που

[1] Πηγή wikipedia



Σχήμα 2: Διευθυνσιοδότηση των threads (V0).

περιγράψαμε παραπάνω και **αναθέτει το κάθε βήμα το οποίο εκτελεί τις ανταλλαγές στη κάρτα γραφικών**. Το βήμα αυτό αντιστοιχεί στις **κάθετες γραμμές** του σχήματος 1. Αυτό γίνεται με την κλήση του CUDA kernel *bitonicStep<<<Nbl, Nth>>>()*, όπου δημιουργούμε N_{bl} blocks από N_{th}, τέτοια ώστε $N_{bl} \cdot N_{th} = \frac{N}{2}$ threads, τα οποία εκτελούν το σώμα της συνάρτησης. Ο χρήστης μπορεί να επιλέξει τον αριθμό των threads ανά block (ή διαφορετικά blocksize) από τη γραμμή εντολών με το όρισμα `-b` ή `--block-size`. Έτσι το κάθε thread εκτελεί μια ανταλλαγή με την άνω διευθυνσιοδότηση και ο αλγόριθμος τερματίζει στο τέλος του διπλού βρόχου. Το πλεονέκτημα αυτής της μεθόδου είναι η απλότητά της, κάτι που προφανώς πληρώνουμε με έναν αρκετά μεγάλο αριθμό κλήσεων προς την κάρτα γραφικών το οποίο προσθέτει overhead, όπως θα δούμε και παρακάτω.

2.2. Δεύτερη έκδοση (V1)

Από την εκφώνηση της εργασίας ήδη υπάρχουν οδηγίες για βελτιστοποιήσεις. Η πρώτη βελτιστοποίηση λοιπόν αφορά αυτές ακριβώς τις κλήσεις των kernels, των συναρτήσεων δηλαδή που επιταχύνονται από την κάρτα γραφικών. Γίνεται δηλαδή η **θεώρηση** από την εκφώνηση, ότι **η ελαχιστοποίηση αυτών των κλήσεων θα βελτιστοποιήσει τη διαδικασία**. Φυσικά, αυτό μένει πάντα να αποδειχτεί στην πράξη.

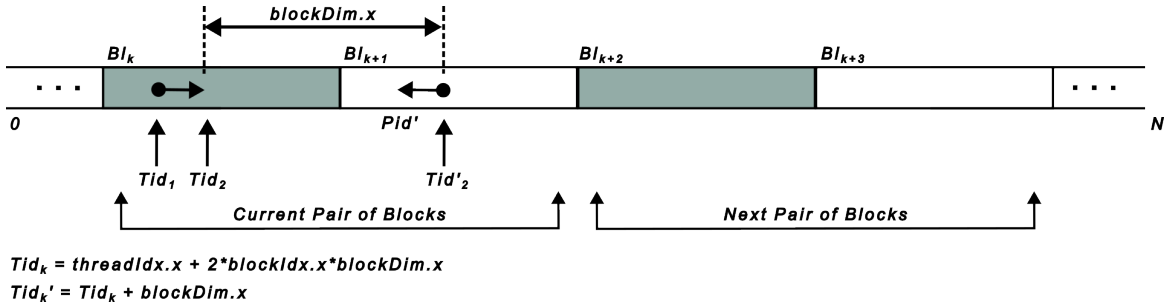
Ο τρόπος που προτείνεται ανήκει στην κατηγορία βελτιστοποιήσεων που λέγεται loop unrolling, όπου ένα μέρος του βρόχου αφαιρείται από τον έλεγχο του βρόχου και καλείται "με το χέρι". Στη δική μας περίπτωση το μέρος που αφαιρείται μπορεί να μπει μέσα στην κλήση kernel αξιοποιώντας καλύτερα τους πόρους που μπορεί να χρησιμοποιήσει, μειώνοντας έτσι και τον αριθμό κλήσεων.

Ενώ όμως στην έκδοση V0 η κάθε κλήση kernel αφορούσε ένα στιγμιότυπο (μια κάθετη γραμμή) του αλγόριθμου, εδώ οι κλήσεις αφορούν περισσότερα. Ο κάθε kernel δηλαδή θα εκτελέσει τμήματα περισσοτέρων της μίας γραμμής του σχήματος 1. Αυτό όμως δημιουργεί data race, καθώς οι τιμές αυτών των στοιχείων διαμοιράζονται μεταξύ των διαφορετικών κλήσεων.

Για να λύσουμε αυτό το πρόβλημα εργαστήκαμε έτσι ώστε:

- Να **περιορίσουμε** το εύρος των δεδομένων που "βλέπει" ο κάθε kernel.
- Να **συγχρονίσουμε** τις κλήσεις εσωτερικά του kernel στο κάθε στιγμιότυπο.

Για το λόγο αυτό αρχικά αλλάξαμε τον τρόπο της διευθυνσιοδότησης από την έκδοση V0. Η διευθυνσιοδότηση του πίνακα πλέον δεν ακολουθεί την αρίθμηση των thread, αλλά αντίθετα τα threads δείχνουν μόνο στα ζυγά blocks αφήνοντας ελεύθερο "ένα – παρά – ένα" block. Δεδομένου όμως ότι ο αριθμός των threads ενός block αρκεί για να ανταλλάξει 2 blocks του πίνακα, μεταθέτουμε τα πλεονάζοντα threads απόσταση ένα block. Με αυτό τον τρόπο έχουμε περιορίσει τα threads μέσα στις δυάδες των block. Έτσι το εύρος διευθύνσεων που διευθυνσιοδοτείται από τα threads του block είναι όσο πιο κοντά γίνεται. Το σχήμα 3 παρουσιάζει αυτή τη μεθοδολογία.



Σχήμα 3: Διευθυνσιοδότηση των threads (V1).

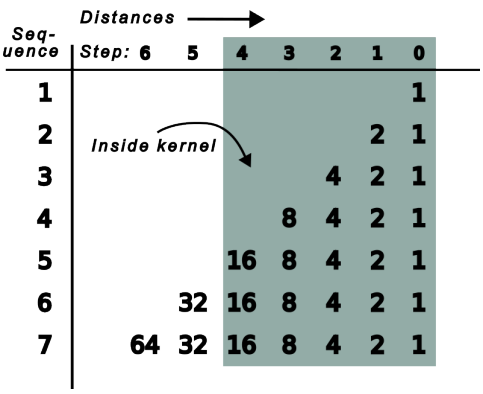
Επομένως, για να λύσουμε το πρόβλημα του data race πρέπει να βεβαιωθούμε πως η κάθε κλήση του kernel αφορά αποστάσεις που βρίσκονται εντός των δυάδων των blocks. Όπως είδαμε όμως και στην παράγραφο 2.1 για την m-οστή ακολουθία, η απόσταση των ανταλλαγών μεταξύ των γειτόνων ξεκινάει από 2^{m-1}, όπου $m-1$ είναι το αρχικό βήμα του εσωτερικού βρόχου (έστω S_0). Άρα για το βήμα S_i του εσωτερικού βρόχου από το οποίο οι ανταλλαγές βρίσκονται εσωτερικά των δυάδων των block αρκεί:

$$BlockSize = N_{th} \geq 2^{m_i - 1} = 2^{S_i} \Rightarrow \log_2(N_{th}) \geq S_i$$

Όπου το S_i είναι και το μέγιστο βήμα από το οποίο μπορεί να ξεκινήσει το loop unrolling.

Στο σχήμα 4 φαίνεται ένα παράδειγμα ταξινόμησης πίνακα 128 θέσεων και block size = 16, όπου τα βήματα με σκιαγράφηση μπορούν να υλοποιηθούν μέσα στη συνάρτηση kernel. Μπορεί κανείς να παρατηρήσει πως τα βήματα με αποστάσεις $16, 8, 4, 2, 1$ είναι τα βήματα $4, 3, 2, 1, 0$ όπου $\log_2(N_{th}) = \log_2(16) = 4$.

Για να υλοποιήσουμε αυτή την προσέγγιση, σπάσαμε την *bitonicStep()* σε δύο εκδόσεις. Η μία (*interBlockStep()*) καλείται για αποστάσεις μεγαλύτερες από δύο block size και η άλλη (*inBlockStep()*)για μικρότερες. Επίσης, η δεύτερη υλοποιεί και το μέρος του βρόχου που αναλογεί στα βήματα εντός kernel.



Σχήμα 4: Βήματα διτονικής ταξινόμησης πίνακα 128 θέσεων (block size = 16).

Το μόνο που μένει είναι πλέον το πρόβλημα του **συγχρονισμού**. Εδώ χρειάζεται να κρατήσουμε τον συγχρονισμό που είχαμε σε κάθε κάθετη γραμμή του σχήματος 1, απλώς αυτός θα λάβει χώρα εσωτερικά του kernel. Η συνάρτηση που μας δίνει αυτή τη δυνατότητα είναι η *__syncthreads()*, την οποία και καλούμε μετά από την ολοκλήρωση της κάθε ακολουθίας.

Όπως είναι φανερό ο διαχωρισμός μεταξύ των βημάτων που μπορούν να γίνουν εσωτερικά και αυτών εξωτερικά του kernel έχει άνω όριο, αλλά όχι κάτω. Στην υλοποίησή μας παρόλα αυτά δεν δίνουμε τη δυνατότητα ρύθμισης αυτού του ορίου. Αυτό γιατί ούτως ή άλλως υπάρχει ρύθμιση για το μέγεθος του block, το οποίο και χρησιμοποιούμε για να κάνουμε και αυτό το διαχωρισμό.

2.3. Τρίτη έκδοση (V2)