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

21 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()*).

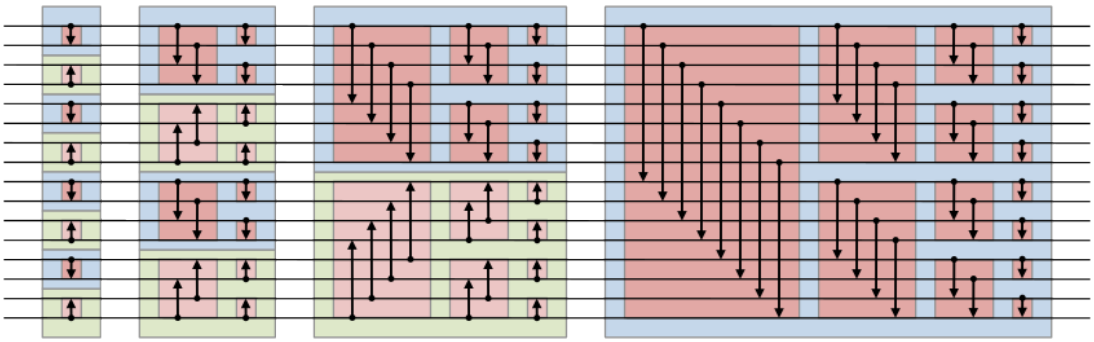
Όλοι οι παραπάνω τύποι είναι templates, και βρίσκονται στα αντίστοιχα headers (.hpp) αντί για αρχεία .cpp. Η συνάρτηση *main()* που υλοποιεί τον αλγόριθμο μαζί με τον validator και το command line interface βρίσκονται στο αρχείο main.cpp. Στο αρχείο config.h υπάρχουν οι compile time ρυθμίσεις της υλοποίησης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

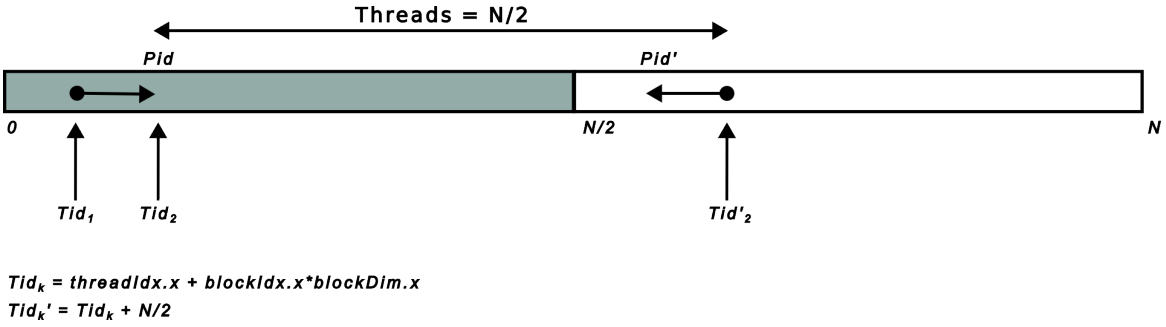
Η εικόνα 1 παρουσιάζει αυτή τη λογική[1].

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.



Σχήμα 2: Διευθυνσιοδότηση των threads (V0).

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.

[1] Πηγή wikipedia

Έχοντας κατανέμει λοιπόν τα threads στον πίνακα, η υλοποίηση (*bitonicSort()*) εκτελεί το διπλό βρόχο που 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 $\boldsymbol{N_{bl}}$ blocks από $\boldsymbol{N_{th}}$, τέτοια ώστε $\boldsymbol{N_{bl} \cdot N_{th}} = \frac{N}{2}$ 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.

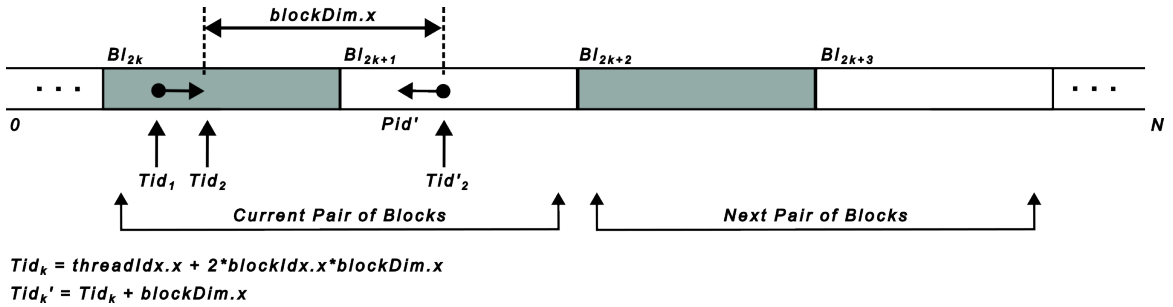
Ο τρόπος που προτείνεται ανήκει στην κατηγορία βελτιστοποιήσεων που λέγεται loop unrolling, όπου ένα μέρος του βρόχου αφαιρείται από τον έλεγχο του βρόχου και καλείται "με το χέρι". Στη δική μας περίπτωση το μέρος που αφαιρείται μπορεί να μπει μέσα στην κλήση kernel αξιοποιώντας καλύτερα τους πόρους 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 $m-1$

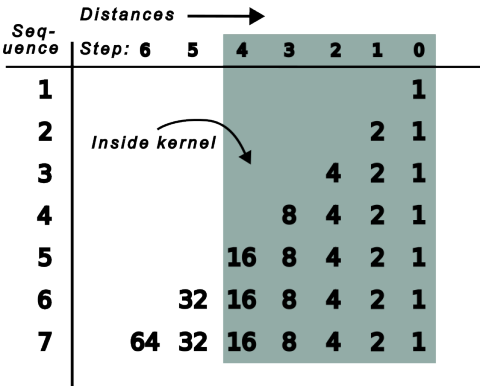
είναι το αρχικό βήμα του εσωτερικού βρόχου (έστω S_0). Άρα για το βήμα S_i του εσωτερικού βρόχου από το οποίο οι ανταλλαγές βρίσκονται εσωτερικά των δυάδων των block αρκεί:

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

Όπου το $S_i = \log_2(N_{th})$ 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 ίσες και μικρότερες. Επίσης, η δεύτερη υλοποιεί και το μέρος του βρόχου που αναλογεί στα βήματα εντός kernel. Ο παρακάτω κώδικας υλοποιεί αυτή τη προσέγγιση:



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

```
size_t Nth = config.blockSize;
size_t Nbl = NBlocks(size);

auto Stages          = static_cast<size_t>(log2(size));
auto InnerBlockSteps = static_cast<size_t>(log2(Nth));
for (size_t stage = 1; stage <= Stages; ++stage) {
    size_t step = stage - 1;
    for ( ; step > InnerBlockSteps; --step) {
        interBlockStep<<<Nbl, Nth>>>(dev_data, size, step, stage);
        cudaDeviceSynchronize();
    }
    inBlockStep<<<Nbl, Nth>>>(dev_data, size, step, stage);
    cudaDeviceSynchronize();
}
```

Το μόνο που μένει πλέον είναι 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 του κάθε βήματος.

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. Έτσι **αν ο χρήστης επιλέξει block size τότε όλες οι ανταλλαγές που χωρούν σε αυτό το μέγεθος θα λάβουν χώρα εντός του kernel**.

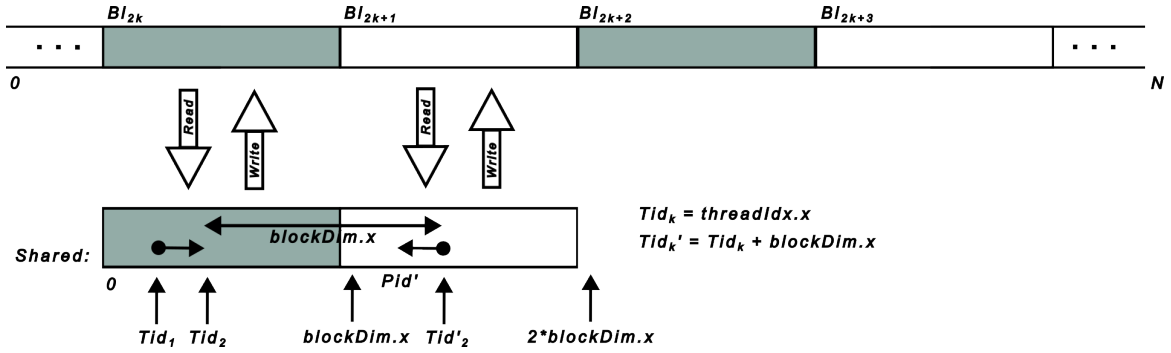
2.3. Τρίτη έκδοση (V2)

Η δεύτερη βελτιστοποίηση που προτείνεται από την εκφώνηση αφορά τη **χρήση της κοινής μνήμης** μεταξύ των threads στο ίδιο block. Η μνήμη αυτή (shared memory) μπορεί, ανάλογα πάντα και με την αρχιτεκτονική, να είναι μία ή και δύο τάξεις μεγέθους γρηγορότερη από την global. Ενώ βέβαια αυτό φαντάζει **ειδυλλιακό**, θα πρέπει να τονίσουμε ότι για την συγκεκριμένη βελτιστοποίηση θα χρειαστούν αρκετές παραχωρήσεις. Για αυτό το λόγο είδαμε αυτή την λύση με **αρκετό σκεπτικισμό**.

Τα βασικά βήματα που πρέπει να ακολουθήσουμε ώστε να χρησιμοποιήσουμε τη κοινή μνήμη είναι:

- **Αντιγραφή** των στοιχείων που ανταλλάσσονται από τα threads του block στη shared μνήμη.
- **Προσαρμογή της διευθυνσιοδότησης** των threads εντός του block ώστε να αντιστοιχούν στα όρια της shared μνήμης.
- **Εκτέλεση** του αλγόριθμου της έκδοσης **V1** στην τοπική μνήμη με τις προσαρμοσμένες διευθύνσεις
- **Αντιγραφή** των στοιχείων πίσω στη global μνήμη.

Το σχήμα 5 παρουσιάζει αυτή την προσέγγιση.



Σχήμα 5: Διευθυνσιοδότηση των threads (V2).

Βλέπουμε λοιπόν πως **ο αλγόριθμος είναι πρακτικά ο ίδιος**, απλώς **οι ανταλλαγές** στα βήματα εντός του block **λαμβάνουν χώρα στη shared μνήμη** αντί της global. Επίσης πολύ εύκολα μπορούμε να αντιληφθούμε πως οι απαιτήσεις σε shared μνήμη είναι δυο φορές το μέγεθος των blocks. Τη μνήμη αυτή τη δεσμεύουμε δυναμικά μέσω ορίσματος στον kernel *inBlockStep«<Nbl, Nth, kernelMemSize»>()*.

Η αλλαγή για την δεύτερη έκδοση βρίσκεται επί της ουσίας στη συνάρτηση *inBlockStep()*. Η συνάρτηση *interBlockStep()* παραμένει ως έχει, καθώς εργάζεται με αποστάσεις μεγαλύτερες αυτών που χωρούν στη shared μνήμη που έχουμε δεσμεύσει για τα blocks.

3. Επιδοσεις και βελτιστοποιηση

Όπως φαίνεται και από το σχήμα 5, στην δεύτερη έκδοση, ενώ χρησιμοποιούμε την shared μνήμη για τις ανταλλαγές εντός block, αυτό **δεν φαίνεται να περιορίζει την πρόσβαση στη global μνήμη**. Μάλιστα η πρόσβαση είναι πλέον ντετερμινιστική. Το κάθε thread κάνει 2 αναγνώσεις και 2 εγγραφές, ενώ στην έκδοση V1 οι εγγραφές για παράδειγμα γίνονται μόνο όταν χρειάζεται. Επίσης η λογική που εκτελείται εντός thread είναι περισσότερη καθώς πρέπει να γίνουν οι αντιγραφές αλλά και η προσαρμογή των διευθύνσεων. Θα πρέπει λοιπόν το κέρδος από τις γρήγορες ανταλλαγές στη shared μνήμη να είναι πολύ μεγάλο, ούτως ώστε να αντισταθμίσουμε αυτό το κόστος. Αυτός είναι και ο λόγος που όπως αναφέραμε βλέπουμε τη 2η έκδοση με σκεπτικισμό.

Αρχικά για να διερευνήσουμε τις επιδόσεις εκτελέσαμε τοπικά σε μία GTX 1650 τις εκδόσεις και είδαμε ότι η 2η έκδοση υστερεί κατά τι. Έπειτα επιβεβαιώσαμε τις ανησυχίες μας στη συστοιχία όπου στην ampere A100 πάλι η V2 ήταν πιο αργή και μόνο στην Tesla P100 πιο γρήγορη. Για να διαπιστώσουμε όμως επακριβώς τι συμβαίνει χρησιμοποιήσαμε το εργαλείο ncu ώστε να πάρουμε το προφίλ των εκτελέσεων. Οι έξοδοι του εργαλείου είναι διαθέσιμοι στους καταλόγους RC1 και RC2 στο αποθετήριο της εργασίας εδώ.

Ενδιαφέρον παρουσιάζουν οι εξής μετρήσεις για την *inBlockStep()* που αποτελεί και την διαφορά μεταξύ των εκδόσεων:

Metric	V1	V2	diff
gpu time duration.sum	184 µs	**230 µs**	+25%
average sectors per request-mem global ld.ratio	3.47	4.00	-
average sectors per request-mem global st.ratio	4.71	4.00	-
mem shared cmd read.sum	0	525k	-
mem shared cmd write.sum	0	314k	-
smps average warp latency stalled barrier.pct	159%	**116%**	-27.0%
smsp cycles active.avg	253k	**315k**	+24.5%
smsp cycles active.sum	16.2M	**20.2M**	+24.7%
smsp inst executed.avg	132k	**189k**	+43.2%
smsp inst executed.sum	8.46M	**12.1M**	+43.0%

Πίνακας 1: *Σύγκριση των εκδόσεων V1 και V2.*

Πέραν του ότι βλέπουμε ξεκάθαρα ότι ο χρόνος της συνάρτησης είναι μεγαλύτερος δικαιολογώντας τις μετρήσεις, παρατηρούμε πως:

- Οι **αναγνώσεις/εγγραφές από/προς την global μνήμη** ναι μεν έγιναν ντετερμινιστικές (ratio 4 ακριβώς, όσες και οι προσβάσεις που αναλογούν στο thread), αλλά η έκδοση V1 είχε ήδη πολύ καλή συμπεριφορά, καθώς είχε γίνει προσπάθεια ώστε η πρόσβαση στη μνήμη να είναι όσο μειωμένη γίνεται. Μάλιστα το read ήταν και καλύτερο. Έτσι ουσιαστικά δεν μπορέσαμε να κερδίσουμε κάτι από εδώ.
- Εκτός από την παραπάνω πρόσβαση στη μνήμη, **προστέθηκε η πρόσβαση στη shared μνήμη** η οποία είναι σε σημαντικό ποσοστό ($525k$ αναγνώσεις και $314k$ εγγραφές).
- **Ο αριθμός των κύκλων αλλά και των εντολών** που εκτελούνται από τα τους streaming multiprocessors (SMs) έχει εκτοξευτεί κατά **25%** και **43%** αντίστοιχα. Αυτό θεωρούμε ότι συμβαίνει πάλι γιατί η έκδοση V1 είχε αρκετά προσεγμένο κώδικα με σκοπό τη μείωση των εντολών που εκτελούνται. Έτσι η έστω και μικρή αλλαγή για να γίνουν οι αντιγραφές από και προς την global αλλά και η προσαρμογή των διευθύνσεων αποδεικνύεται σημαντική.

Με βάση τα παραπάνω αποτελέσματα συμπεραίνουμε πως για να βελτιωθεί η έκδοσή μας χρειάζεται:

- Να καταφέρουμε με *κάποιο τρόπο* να **μειώσουμε τις εντολές**.
- Να κάνουμε την **εκτέλεση των εντολών** που αφορά τις ανταλλαγές στο σώμα της *inBitonicStep()* **ακόμα πιο γρήγορη**.
- Να αναζητήσουμε κάποιου **άλλου είδους βελτιστοποίηση** που δεν έχει να κάνει με το shared memory optimization.

Αν και καταβάλαμε φιλότιμη προσπάθεια να μειώσουμε τις εντολές, τελικά αυτή η κατεύθυνση δεν μας οδήγησε πουθενά. Αντίθετά για τα άλλα δύο σημεία προσπαθήσαμε κάποιες λύσεις.

3.1. Ελαχιστοποίηση των αρχικών *inBlockStep()* κλήσεων

Μια βελτιστοποίηση που σκεφτήκαμε και δεν έχει να κάνει με το shared memory optimization είναι ο αριθμός των *inBlockStep()* στην αρχή του αλγόριθμου. Αν προσέξει κανείς το σχήμα 4 εύκολα θα παρατηρήσει πως στην αρχή του αλγόριθμου ο διπλός βρόχος περιέχει βήματα που χωράνε εξολοκλήρου στο block. Παρόλα αυτά γίνονται ξεχωριστές κλήσεις στην *inBlockStep()* χωρίς να μεσολαβεί κάποια κλήση στην *interBlockStep()*. Μπορούμε έτσι να **συνενώσουμε όλες αυτές τις πρώτες ξεχωριστές κλήσεις της *inBlockStep()* σε μία συνάρτηση** η οποία δεν χρειάζεται να επιστρέφει παρά μόνο όταν τα βήματα του βρόχου φτάσουν να χρειαστεί να καλέσουν την πρώτη *interBlockStep()*. Η συνάρτηση αυτή είναι η *prephace()*.

Πριν αποδεχτούμε βέβαια αυτή τη βελτιστοποίηση έπρεπε πρώτα να την μετρήσουμε. Η έκδοση στην οποία δοκιμάστηκε αυτή η προσέγγιση είναι η RC2 και ο αναγνώστης μπορεί να τη βρει στο αποθετήριο της εργασίας.

Ο πίνακας παρακάτω δείχνει τα αποτελέσματα που πήραμε από το profiling εργαλείο ncu, για prephase που αντικαθιστά 28 κλήσεις *inBlockStep()*, αλλά και κάποιους ενδεικτικούς χρόνους εκτέλεσης από τη συστοιχία:

Metric	**InBlockStep()**	**prephase()**
gpu time duration.sum	184 μs	1.06 ms
average sectors per request-mem global ld.ratio	3.47	3.91
average sectors per request-mem global st.ratio	4.71	5.48
smsp inst executed.avg	132k	770k
smsp inst executed.sum	8.46M	49.3M
Sorting Time	**No-Opt (RC1)**	**Prephase-Opt (RC2)**
V1Q27 - A100	156 [msec]	153 [msec]
V1Q30 - A100	1580 [msec]	1566 [msec]

Πίνακας 2: *Σύγκριση των εκδόσεων InBlockStep() - RC1 και prephase() - RC2.*

Από τον παραπάνω πίνακα συμπεραίνουμε πως η αλλαγή βελτιώνει τόσο τη συμπεριφορά όσο και τους χρόνους εκτέλεσης. Στο αποθετήριο της εργασίας υπάρχουν όλες οι μετρήσεις και τα αποτελέσματα του profiling τόσο για την έκδοση RC1 (πριν τη βελτιστοποίηση) όσο και για την RC2 (μετά τη βελτιστοποίηση).

3.2. Ελαχιστοποίηση του χρόνου εκτέλεσης της *inBlockStep()*

Παρατηρώντας τον αλγόριθμο της έκδοσης V2 μπορεί κάποιος να παρατηρήσει ότι η πρόσβαση στη global μνήμη ακολουθεί τα εξής. Το κάθε thread:

1. Διαβάζει δύο θέσεις μνήμης(πίνακα) από τη global
2. Γράφει δύο θέσεις πίνακα στη shared
3. Διαβάζει δύο θέσεις από τη shared
4. Γράφει με μία πιθανότητα δύο θέσεις στη shared
5. Διαβάζει δύο θέσεις από τη shared
6. Γράφει δύο θέσεις στη global

Συνολικά global: 2RD/2WR και shared: 4RD/(2 ή 4)WR. Αν κοιτάξει κάποιος τα αποτελέσματα από το profiling στον πίνακα 1 όπου οι αναγνώσεις/εγγραφές στη shared είναι RD:525k/WR:314k θα δει ότι επιβεβαιώνεται αυτός ο συλλογισμός.

Η ιδέα αφορά ουσιαστικά το βήμα 3. Αντί να διαβάζουμε από τη θέση του shared πίνακα που μόλις γράψαμε, **θα μπορούσαμε να κρατάμε την τιμή του τρέχοντος thread που διαβάσαμε από την global, σε ένα register**. Έπειτα η σύγκριση με τον κάθε γείτονα θα γίνεται μεταξύ register και shared και μόνο όταν γίνεται ανταλλαγή, η τιμή του register θα επαναγράφεται στη shared. Αυτό το trick μας **μειώνει τον αριθμό των προσβάσεων στη shared σε: 3RD/(2 ή 4)WR** κάτι που αναμένεται να μειώσει τις αναγνώσεις από **525*k*** σε **393*k*** . Απαιτεί βέβαια προσοχή γιατί δημιουργούμε πλεονασμό δεδομένων μεταξύ του register και της θέσης στον πίνακα share που αυτός αντιστοιχεί.

Με προσεκτική αλλαγή στην *exchange()* ώστε να επιστρέφει bool για το αν έγινε ανταλλαγή, δεν προστίθεται κάποιο branch στη ροή ελέγχου κάτι που μας επιτρέπει απλώς να προσθέσουμε δύο εντολές για ανάγνωση και εγγραφή στη register μεταβλητή.

Δυστυχώς όμως δεν καταφέραμε ποτέ να πετύχουμε μια λειτουργική υλοποίηση. Είτε η μεταβλητή που θα έπρεπε να είναι στους registers ήταν στη local μνήμη, είτε δημιουργούνταν κάποιο mismatch μεταξύ της μεταβλητής του register και της αντίστοιχης θέσης στον shared πίνακα. Ο αναγνώστης μπορεί να βρει ένα draft στο branch RC3 του αποθετηρίου αν θέλει να πειραματιστεί[2]!

[2]Βέβαια δεν μπορώ να σκεφτώ ένα καλό λόγο για να το κάνει κάποιος αυτό στον εαυτό του.

4. Τελικη εκδοση

Έχοντας λοιπόν προηγηθεί η παραπάνω ανάλυση, για την τελική έκδοση της εφαρμογής επιλέξαμε απλά να κρατήσουμε την βελτιστοποίηση με το prephase. Η επαλήθευση του αλγόριθμου έγινε μέσω ενός validator που ενσωματώθηκε στο εκτελέσιμο και μπορεί να καλεστεί από το command line argument `--validation`, ο οποίος στο τέλος της ταξινόμησης ελέγχει αν ο πίνακας είναι ορθά ταξινομημένος.

Στον κατάλογο hpc, βρίσκονται scripts που δημιουργούν τα batch scripts για τη συστοιχία και τα οποία χρησιμοποιήθηκαν για τις τελικές μετρήσεις. Ο αναγνώστης μπορεί να δοκιμάσει τον αλγόριθμο στη συστοιχία για παράδειγμα στην ampere, δίνοντας από το root directory:

```
$ module load gcc/9.2.0 cuda/11.1.0
$ make -j hpc-build
$ cd hpc && ./makeSlurmScripts.sh
$ cd ..  && ./hpc/submitJobs.sh hpc ampere
```

Ο αναγνώστης μπορεί επίσης τοπικά να μεταγλωττίσει και να εκτελέσει τον αλγόριθμο, δίνοντας από το root directory:

```
$ make -j hpc-build
$ ./out/v1/bitonicCUDA -v --validation --perf 5 -q 24
$ # or
$ ./out/v2/bitonicCUDA -v --validation --perf 5 -b 512 -q 24
```

Το οποίο για πίνακα μεγέθους q=24 θα εκτελέσει την ταξινόμηση 5 φορές, θα ελέγξει την εγκυρότητά της και θα εκτυπώσει τον ενδιάμεσο χρόνο, χρησιμοποιώντας στην πρώτη εντολή την έκδοση v1 και στη δεύτερη την έκδοση v2 αλλά και block size 512 threads. Δίνοντας το όρισμα `-h` ο χρήστης μπορεί να δει όλες τι επιλογές εκτέλεσης και μια μικρή τεκμηρίωση ώστε να πειραματιστεί.

Παρακάτω παραθέτουμε τα αποτελέσματα στους τελικούς χρόνους από τη συστοιχία ampere, gpu(tesla), αλλά και από μια κάρτα GTX1650, όπου κρατήσαμε τον ενδιάμεσο χρόνο από 7 εκτελέσεις.

		Q20	Q22	Q24	Q26	Q28	Q30
V0 - ampere	Total	8733 usec	59 msec	56 msec	229 msec	976 msec	3971 msec
	Sort	3134 usec	6074 usec	29 msec	120 msec	554 msec	2453 msec
V0 - tesla	Total	6559 usec	23 msec	101 msec	442 msec	1986 msec	9071 msec
	Sort	5419 usec	19 msec	85 msec	385 msec	1762 msec	8034 msec
V0 - gtx1650	Total	16 msec	71 msec	327 msec	1493 msec	6782 msec	N/A
	Sort	15 msec	66 msec	305 msec	1408 msec	6445 msec	N/A
V1 - ampere	Total	2676 usec	10 msec	34 msec	134 msec	664 msec	2938 msec
	Sort	1417 usec	3699 usec	17 msec	71 msec	351 msec	1556 msec
V1 - tesla	Total	4559 usec	18 msec	80 msec	358 msec	1618 msec	7326 msec
	Sort	3418 usec	14 msec	64 msec	300 msec	1394 msec	6411 msec
V1 - gtx1650	Total	8271 usec	37 msec	161 msec	774 msec	3701 msec	N/A
	Sort	6666 usec	31 msec	139 msec	689 msec	3358 msec	N/A
V2 - ampere	Total	2883 usec	12 msec	34 msec	135 msec	749 msec	3063 msec
	Sort	1574 usec	4134 usec	19 msec	76 msec	372 msec	1650 msec
V2 - tesla	Total	4407 usec	17 msec	75 msec	338 msec	1533 msec	6932 msec
	Sort	3264 usec	13 msec	60 msec	280 msec	1308 msec	6048 msec
V2 - gtx1650	Total	8985 usec	38 msec	169 msec	810 msec	3864 msec	N/A
	Sort	7294 usec	33 msec	148 msec	726 msec	3527 msec	N/A

Πίνακας 3: *Τελικές μετρήσεις στη συστοιχία.*