Παραλληλισμος αλγορίθμων Bubble sort, Bucket sort και της μεθόδου του Otsu

Δημοκρίτειο Πανεπιστήμιο Θράκης
Τμήμα ηλεκτρολόγων και ηλεκτρολόγων μηχανικών
Εργαστήριο Αρχιτεκτονικής Υπολογιστών
και Συστημάτων Υψηλών επιδόσεων
Τεχνολογία παράλληλης Επεξεργασίας
Ευάγγελος Λουρμπάκης
evanlour@ee.duth.gr

May 18, 2025

Περιεχόμενα

1 Εισαγωγή

Τη σήμερων ημέρα παρατηρείται μια συνεχή ανοδική τάση στον αριθμό των επεξεργαστών και στον επαγγελματικό τομέα αλλά και στο χώρο του καταναλωτή. Καθώς δεν μπορεί να γίνει αυτόματη ανάθεση των διεργασιών για να αξιοποιηθούν όλοι οι πυρήνες, δημιουγείται η ανάγκη για ανακατασκευή της πλειονότητας των προγραμμάτων με σκοπό να αξιοποιηθεί πλήρως όλη η διαθέσιμη υπολογιστική δύναμη. Επιδιώκεται να προσδιοριστεί το ποσοστό επιτάχυνσης των αλγορίθμων *B*ubble sort και *B*ucket sort, με τη χρήση του αλγορίθμου *M*erge sort για την εσωτερική αναδιάταξη των δεξαμενών του αλγορίθμου. Επίσης θα εξεταστεί η επιτάχυνση της μεθόδου του Otsu. Για την παραλληλοποίηση των παρακάτω αλγορίθμων θα χρησιμοποιηθεί η βιβιοθήκη [2] OpenMP στη γλώσσα προγραμματισμού C.

2 Ανάλυση λειτουργίας των αλγορίθμων

2.1 Ο αλγόριθμος Bubble sort

Ο αλγόριθμος *B*ubble sort είναι από τους πιό απλούς και διαδεδομένους αλγορίθμους που υπάρχουν. Ο αλγόριθμος ξεκινάει απο την αρχή του πίνακα και συγκρίνει διαδοχικά όλους τους αριθμούς για να δεί εάν βρίσκεται στη θέση που πρέπει. Μετα συνεχίζει στο δεύτερο στοιχείο του πίνακα κοκ. Όπως είναι προφανές, ο αλγόριθμος εμφανίζει υψηλή πολυπλοκότητα χρόνου και συγκεκριμένα εμφανίζει $O(N^2)$ πολυπλοκότητα.

Algorithm 1 Ο αλγόριθμος Bubble sort

```
i ← 0
N ← array size
array[N]
for i ← 0 to N − 1 do
    for j ← 0 to N − i − 1 do
        if array[j] > array[j + 1] then
            swap(array[j], array[j + 1])
```

Ενα ακόμη πρόβλημα που προκύπτει είναι πως λόγω της φύσης του αλγορίθμου είναι αρκετά δύσκολο να παραλληλοποιηθεί κατάλληλα χωρίς να χάσει αρκετή απόδοση λόγω του χρόνου συγχρονισμού. Για να αποφεχθούν οι καθυστερήσεις λόγω συγχρονισμού, θα χρησιμοποιηθεί μια παραλλαγή του αλγορίθμου *B*ubble sort: Τον αλγόριθμο *O*dd even sort.

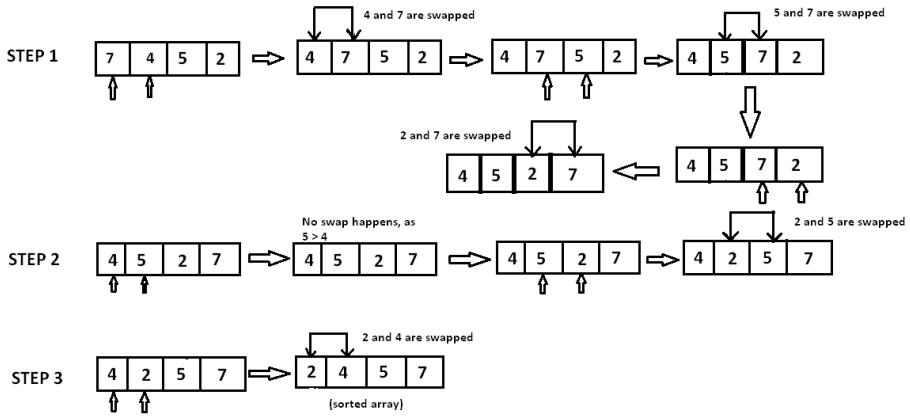

Figure 1: Ο τρόπος λειτουργίας του Bubble sort [3].

2.2 Ο αλγόριθμος Odd even sort

Ο αλγόριθμος *O*dd even sort είναι μια ευνοική παραλλαγή του *B*ubble sort κατά την οποία ελέγχεται εναλλάξ το κάθε στοιχείο του πίνακα με το δεξιά του και με το αριστερά του. Ενώ σε πολυπλοκότητα παρουσιάζει ίδιο θεωρητικό χρόνο εκτέλεσης, λόγω του ότι ελέγχεται το κάθε στοιχείο με το διπλανό του είναι πολύ πιο εύκολο και αποδοτικό να παραλληλοποιηθεί η διαδικασία: Ν πυρήνες ελέγχουν ανά 2 τα στοιχεία στα αριστερά τους και έπειτα αυτά στα δεξιά τους. Επειδή κάθε φορά για κάθε νήμα τα στοιχεία θα είναι ανεξάρτητα των υπολοίπων, δεν χάνουμε χρόνο για συγχρονισμό, το οποίο μας επιτρέπει να παρατηρήσουμε σημαντικές βελτιώσεις στο χρόνο εκτέλεσης.

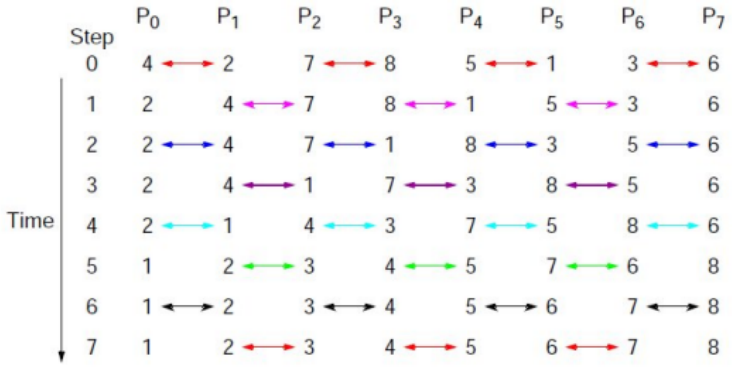

Figure 2: Ο τρόπος λειτουργίας του Odd even sort [7].

Algorithm 2 Ο αλγόριθμος Odd even sort

```
i ← 0
j ← 0
N ← array size
array[N]
while array is not sorted do
    for i ← 1 to N − 1 with jumpsize: 2 do
        if array[i] > array[i + 1] then
            swap(array[i], array[i + 1])
    for j ← 1 to N − 1 with jumpsize: 2 do
        if array[j] > array[j + 1] then
            swap(array[j], array[j + 1])
```

Είναι προφανές πως η μετατροπή η οποία υλοποιήθηκε στον αλγόριθμο επιτρέπει την αποδοτική αξιοποίηση περισσότερων πυρήνων του επεξεργαστή, ακόμη και GPU [4]

2.3 Ο αλγόριθμος Bucket sort

Ο αλγόριθμος *Bucket sort* διαχωρίζει τα στοιχεία του πίνακα σε N δεξαμενές. Έπειτα, στη κάθε δεξαμενή ανακατατάσει τα στοιχεία και στο τέλος αποθηκεύει με τη σειρά τα στοιχεία των containers στον πίνακα εξόδου. Εσωτερικά του container μπορούν να επιλεχθούν διάφοροι αλγόριθμοι. Επιλέχθηκε να χρησιμοποιηθεί ο αλγόριθμος *Merge sort* διότι παρουσιάζει λογαριθμική χρονική πολυπλοκότητα στο χρόνο. Η διαδικασία παραλληλοποίησης είναι σχετικά απλή: Γίνεται διαμοιρασμός των αριθμών σε N threads και τοποθετούνται τα στοιχεία στις ανάλογες τοπικές δεξαμενές. Έπειτα αρκεί να ενωθούν οι ανάλογες δεξαμενές μεταξύ τους και έπειτα να συνδεθούν σε σειρά για να δημιμουργηθεί τελικά ο πίνακας εξόδου. Έτσι αποφεύγεται η εκτεταμένη χρήση εντολών συγχρονισμού μεταξύ των νημάτων και επιτρέπει την πλήρη αξιοποίηση του διαθέσιμου υλικού.

Algorithm 3 Ο αλγόριθμος Bucket sort

```
N ← array size
M ← number of containers
array[N]
containers[number of containers]
i ← 0
j ← 0
for i ← 0 to N − 1 do
    Send array[i] To the corresponding container
for j ← 0 to M do
    merge sort (containers[i])
array ←Concatenate(containers)
```

Figure 3: Ο τρόπος λειτουργίας του Bucket sort [5].

2.4 Ο αλγόριθμος Merge sort

Για την εσωτερική ανακατανομή της κάθε δεξαμενής έχει επιλεχθεί να χρησιμοποιηθεί ο αλγόριθμος *merge sort*. Ο *merge sort* είναι ιδανικός γιατί είναι ευσταθής αλγόριθμος, το οποίο είναι προαπαιτούμενο για τον *bucket sort* και επίσης παρουσιάζει γρήγορους χρόνους εκτέλεσης. Αξίζει να σημειωθεί πως στα πειραματικά δεδομένα επιλέχθηκε να μην υλοποιηθεί παράλληλα διότι αρχικά είναι γνωστή η επιτάχυνση του αλγορίθμου όταν αξιοποιούνται παράλληλες μέθοδοι [8], καθώς επίσης και για φυσιολογικούς αριθμούς δεξαμενών το overhead θα υποβάθμιζε σημαντικά την οποιαδήποτε επιτάχυνση λόγω παραλληλοποίησης. Ο *merge sort* εχει την εξής αρχή λειτουργίας: Διασπά τον αρχικό μεγάλο πίνακα σε ζεύγη των 2, τα ανακατατάσει εσωτερικά και μετά κατασκευάζει σταδιακά τον υπόλοιπο πίνακα παίρνοντας κάθε φορά τον μικρότερο διαθέσιμο αριθμό. Με αυτόν τον τρόπο επιτυγχάνει ταχύτατους χρόνους εκτέλεσης.

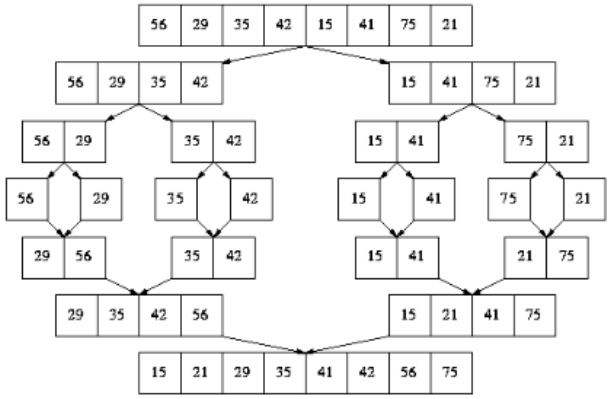

Figure 4: Ο τρόπος λειτουργίας του merge sort [5].

Algorithm 4 Ο αλγόριθμος Merge sort

```
MergeSortArray(array, size)
if size ≤ 1 then
    return
half ← size / 2
left ← new array of size half
right ← new array of size (size - half)
for i = 0 to half − 1 do
    left[i] ← arr[i]
for i = 0 to size − half − 1 do
    right[i] ← arr[half + i]
MergeSortArray(left, half)
MergeSortArray(right, size − half)
leftPointer ← 0
rightPointer ← 0
while leftPointer < half and rightPointer < size − half do
    if left[leftPointer] < right[rightPointer] then
        array[leftPointer + rightPointer] ← leftPointer
        leftPointer + +
    else
        array[leftPointer + rightPointer] ← rightPointer
        rightPointer + +
while leftPointer < half do
    array[leftPointer + rightPointer] ← leftPointer
    leftPointer + +
while rightPointer < size − half do
    array[leftPointer + rightPointer] ← rightPointer
    rightPointer + +
```

3 Otsu's thresholding method

3.1 Η σημασία του thresholding στην φηψιακή επεξεργασία

Στην επεξεργασία εικόνας thresholding ονομάζεται η διαδικασία δυαδικοποίησης μιας ασπρόμαυρης εικόνας με κάποιο ανώτατο όριο. Η διαδικασία αυτή ορίζει την εικόνα κατάλληλη για περαιτέρω αναλύσεις, όπως για παράδειγμα τη τροφοδότησή της σε κάποιο σύχρονο πρόγραμμα μετάφρασης εικόνας σε κείμενο. Είναι απαραίτητο η διαδικασία να είναι αποδοτική και γρήγορη καθώς η επεξεργασία εικόνας είναι εξαιρετικά ακριβή υπολογιστικά, καθώς και πολύ ευαίσθητη σε θόρυβο και δυσμορφίες στην εικόνα εισόδου.

3.2 Ο τρόπος λειτουργίας του αλγορίθμου του Otsu

Συγκεκριμένα ο αλγόριθμος του Otsu χρησιμοποιεί τον παρακάτω τύπο για να μπορέσει να επιλέξει το κατάλληλο threshold για τη δυαδικοποίηση της εικόνας.

$$\sigma_w^2 = \omega_0(t)\sigma_0^2(t) + \omega_1(t)\sigma_1^2(t) \tag{1}$$

Οπου ω_0 και ω_1 είναι τα εκάστοτε βάρη των 2 κλάσεων που καθορίζει το threshold και υπολογίζονται ως εξής:

$$\omega_0(t) = \sum_{i=0}^{t-1} p(i), \quad \omega_1(t) = \sum_{i=t}^{L-1} p(i) \tag{2}$$

Με t να είναι το threshold και το L να είναι ο αριθμός των δεξαμενών που χρησιμοποιούνται. Συνήθως είναι 255. Επιδιώκεται να επιτευχθεί η μέγιστη διακύμανση μεταξύ των 2 κλάσεων που δημιουργούνται από τη τιμή threshold.

$$\sigma_b^2 = \sigma^2 - \sigma_\omega^2(t) = \omega_0(t)(\mu_0 - \mu_T)^2 + \omega_1(t)(\mu_1 - \mu_T)^2 \Rightarrow$$
$$\Rightarrow \sigma_b^2 = \omega_0(t)\omega_1(t)[\mu_0(t) - \mu_1(t)]^2$$

Οι τιμές μ_0, μ_1, μ_T είναι οι μέσες τιμές των κλάσεων και υπολογίζονται ώς εξής:

$$\mu_0 = \frac{\sum_{i=0}^{t-1} ip(i)}{\omega_0(t)} \quad \mu_1 = \frac{\sum_{i=t}^{L-1} ip(i)}{\omega_1(t)} \quad \mu_T = \sum_{i=0}^{L-1} ip(i) \tag{3}$$

3.3 Ο αλγόριθμος του Otsu από τη μεριά του προγραμματιστή

Η πιο διαδεδομένη μέθοδος thresholding που υπάρχει είναι η μέθοδος του Otsu. Σε αυτή τη μέθοδο αρχικά γίνεται εύρεση του ιστογράμματος της εικόνας. Έπειτα, χωρίζει την εικόνα σε 2 κλάσεις: Την κλάση με όλες τις τιμές κάτω από ενα προκαθορισμένο όριο και τις υπόλοιπες. Τότε, γίνεται έυρεση του αριθμού ο οποίος θα ελαχιστοποιήσει τη διαφορά μεταξύ των 2 κλάσεων.

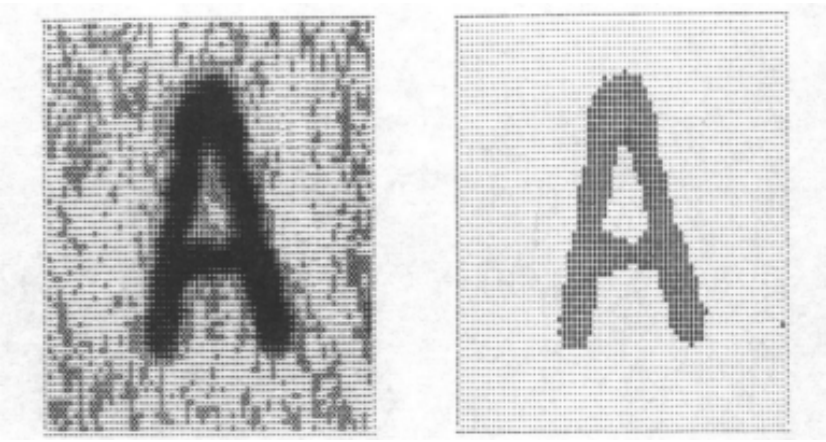

Figure 5: Παράδειγμα λειτουργίας της μεθόδου του Otsu[6].

Algorithm 5 Ο αλγόριθμος του Otsu

```
procedure OtsuThreshold(image)
    Compute histogram of image
    Compute total number of pixels
    for t = 0 to 255 do
        Compute background weight ω_B(t)
        Compute foreground weight ω_F(t)
        Compute background mean μ_B(t)
        Compute foreground mean μ_F(t)
        Compute between-class variance σ_b^2(t)
    return threshold t that maximizes σ_b^2(t)
```

4 Απόδοση αλγορίθμων με τη βιβιοθήκη OpenMP

4.1 Το περιβάλλον εκτέλεσης

Ο κώδικας εκτέλεσης γράφτηκε στη γλώσσα προγραμματισμού C και η παραλληλοποίηση των αποτελεσμάτων υλοποιήθηκε με τη βοήθεια της βιβλιοθήκης OpenMP [2]. Το υλικό εκτέλεσης αποτελείται από ένα σύστημα xeon με 64 πυρήνες και 128GB RAM. Κατά την εκτέλεση και εξόρυξη πειραματικών δεδομένων από τον κώδικα δεν εκτελούνταν καμία άλλη διεργασία για να υπάρξει ισοτιμία στο περιβάλλον εκτέλεσης. Επιπρόσθετα, η χρονομέτρηση έγινε εσωτερικά στον κώδικα μέσω της εντολής *omp_get_wtime()* και μετρήθηκε μόνο ο χρόνος εκτέλεσης του αλγορίθμου. Επιπρόσθετα, αξίζει να αναφερθεί πως έγινε σύγκριση των επιπέδων βελτιστοποίησης του προγράμματος από τον compiler. Τα levels O1, O2 και O3 παρουσίασαν πολύ κοντινούς χρόνους εκτέλεσης.

4.2 Τα αποτελέσματα για τον Bubble sort με OpenMP

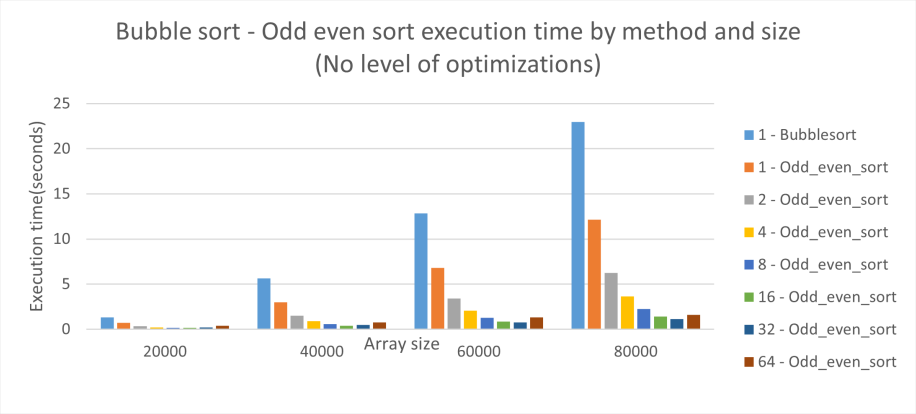

Figure 6: Οι χρόνοι εκτέλεσης Bubble sort και Odd even sort χωρίς optimization levels.

Στα Figure 6 και 7 απεικονίζονται οι χρόνοι εκτέλεσης των αλγορίθμων μαζί με τα threads τα οποία αξιοποίησαν ανάλογα με το μέγεθος του πίνακα. Η ειδοποιός διαφορά είναι η βελτιστοποίηση του προγράμματος κατά τη κατασκευή του προγράμματος. Όπως αναγράφεται και στις περιγραφές το Figure 6 δεν αξιοποιεί βελτιστοποιήσεις από τον compiler, ενώ τι Figure 7 αξιοποιεί επιπέδου 2. Διακρίνονται πολλά ενδιαφέροντα αποτελέσματα: Για πολύ μικρά μεγέθη πινάκων Η παραλληλοποίηση δεν προσδίδει καθόλου speedup, το οποίο είναι λογικό λόγω του overhead που προστίθεται με την εισαγωγή των επιπλέων συναρτήσεων για την αξιοποίηση των παραπάνω πόρων του συστήματος. Επιπρόσθετα, ο αλγόριθμος *odd even sort* παρουσιάζει χαμηλότερους εκτέλεσης ακόμη και όταν χρησιμοποιούμε μόνο ένα thread, το οποίο σημαίνει πως ο γενικός χρόνος εκτέλεσης είναι χαμηλότερος, άρα ο αλγόριθμος είναι αποδοτικοτερος. Σε μεγαλύτερα μεγέθη πινάκων παρατηρούμε πως τα επιπλέον threads προσφέρουν αρκετά μεγάλο speedup και δεν επηρεάζονται από το επιπλέον overhead που εισάγεται κατά την παραλληλοποίηση. Τέλος, παρατηρείται πως ο χρόνος βελτιώνεται σημαντικά σε μικρότερους αριθμούς νημάτων με χρήση βελτιστοποίησης O2. Σε μεγαλύτερους αριθμούς threads η διαφορά είναι πολύ μικρή.

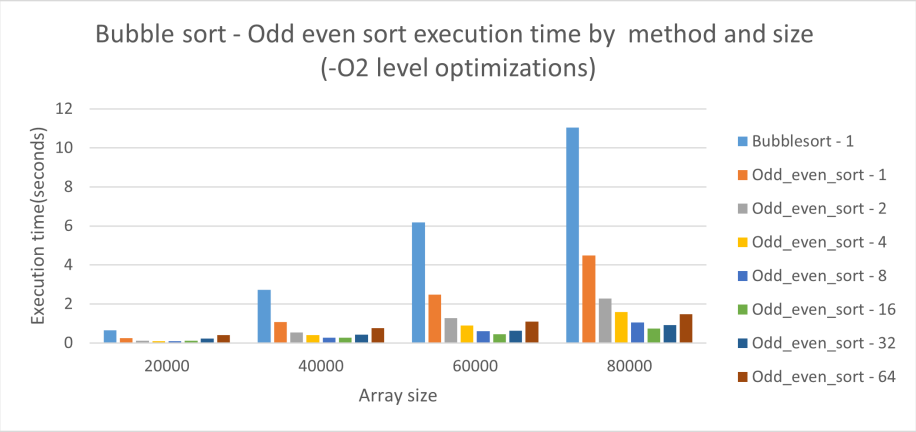

Figure 7: Οι χρόνοι εκτέλεσης Bubble sort και Odd even sort με O2 optimizations.

Figure 8: Οι τιμές επιτάχυνσης και απόδοσης του Odd even sort.

4.3 Τα αποτελέσματα για τον Bucket sort με OpenMP

Για την αξιολόγηση του αλγορίθμου πέραν του χρόνου εκτέλεσης, του μεγέθους του πίνακα και τον αριθμό των νημάτων θα γίνει αξιολόγηση της ταχύτητας και με βάση τον αριθμό των δεξαμενών. Παρατηρώντας το Figure 8 διακρίνεται πως για μεγαλύτερο αριθμό δεξαμενών έχουμε μια μικρή μείωση του χρόνου εκτέλεσης. Επίσης, είναι εμφανής η διαφορά μεταξύ τον αριθμό των νημάτων και υπάρχει βελτίωση έως και για μέγεθος πίνακα 15,000,000. Τέλος, στο figure 9 παρατηρείται μια μεγάλη σχετικά μείωση στο χρόνο εκτέλεσης του αλγορίθμου για χρήση βελτιστοποίησης επιπέδου 2, υποδεικνύοντας πως υπάρχει περιθώριο βελτίωσης.

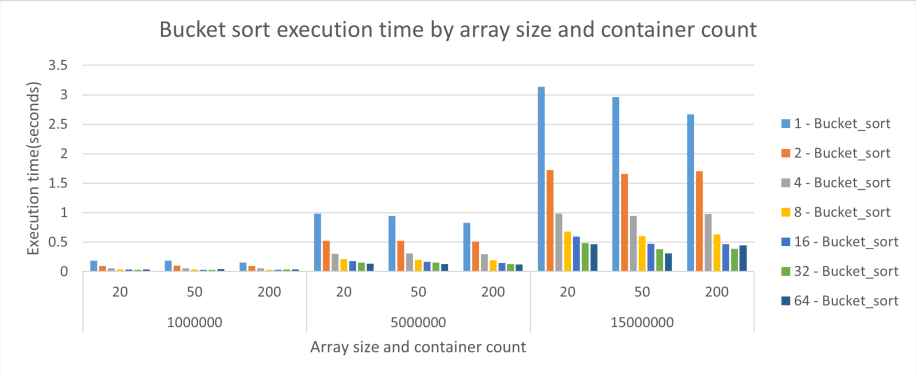

Figure 9: Οι χρόνοι εκτέλεσης για τους Bucket sort και Merge sort για διάφορα μεγέθη δεξαμενών (καμία βελτιστοποίηση).

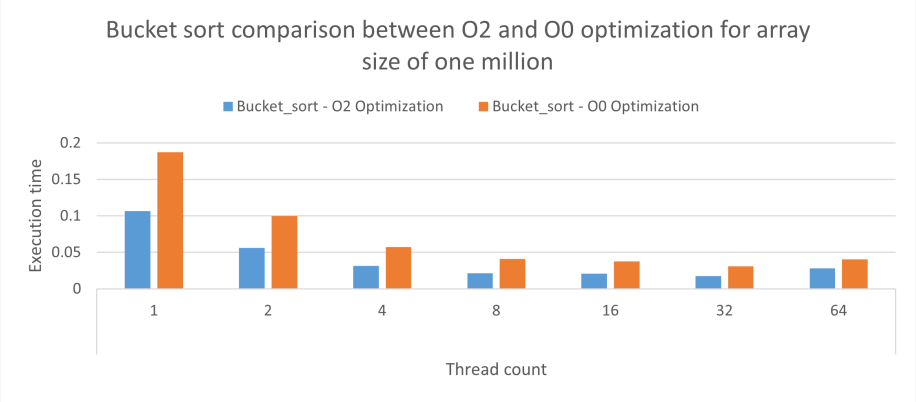

Figure 10: Η διαφορά του χρόνου εκτέλεσης του bucket sort με και χωρίς compiler optimization (20 δεξαμενές).

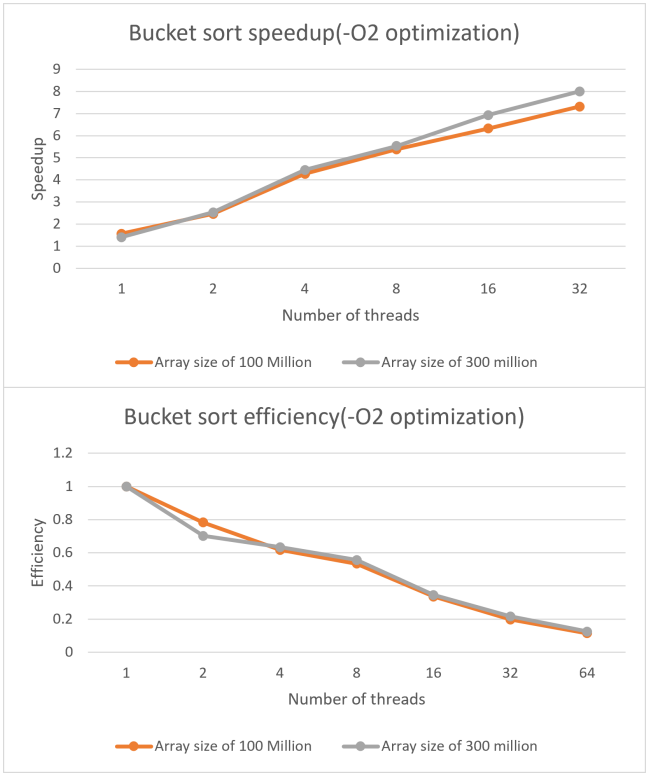

Figure 11: Οι τιμές επιτάχυνσης και απόδοσης του Bucket sort.

4.4 Τα αποτελέσματα για τη μέθοδο του Otsu με OpenMP

Για την μέθοδο του Otsu επιλέχθηκε για να είναι εμφανές η αποδοτικότητα της παραλληλοποίησης του αλγορίθμου η χρήση εικόνας με μέγεθος 30000x20000. Και πάλι, όπως φαίνεται και απο το Figure 10, χρειάζεται ακόμη μεγαλύτερη εικόνα για να αξιοποιηθούν περισσότερα νήματα. Επίσης, όπως και αναφέρθηκε παραπάνω, επιλέχθηκε να απεικονιστεί η διαφορά μεταξύ των διάφορων επιπέδων optimization, η όπως συμπεραίνεται από τα πειραματικά στοιχεία η πανομοιότυπη απόδοση των επιπέδων. Παρ'ολα αυτά, παρατηρείται ο υποδιπλασιασμός του χρόνου εκτέλεσης και για πολυπληθές νήματα και όχι μόνο για ένα νήμα, το οποίο είναι το πιο συνηθισμένο αποτέλεσμα. Τέλος, κρίνεται επιτακτικής ανάγκης να αναφερθεί πως με τη χρήση του επιπέδου βελτιστοποίησης 2 ο αλγόριθμος παρουσιάζει πανομοιότυπους χρόνους και αποτελέσματα με αυτούς της μεθόδου της βιβλιοθήκης OpenCV [1].

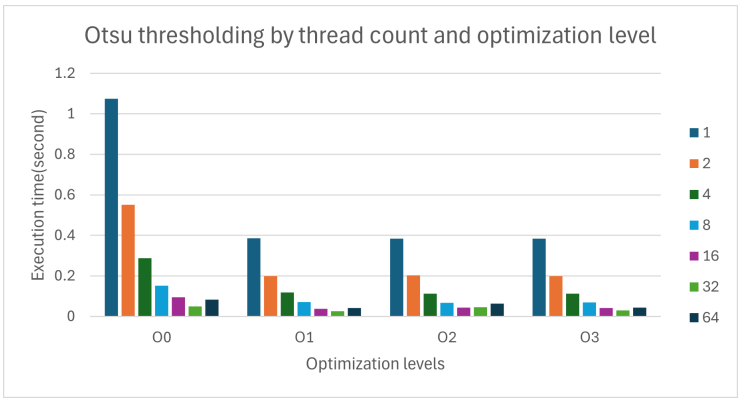

Figure 12: Ο χρόνος εκτέλεσης της μεθόδου του Otsu.

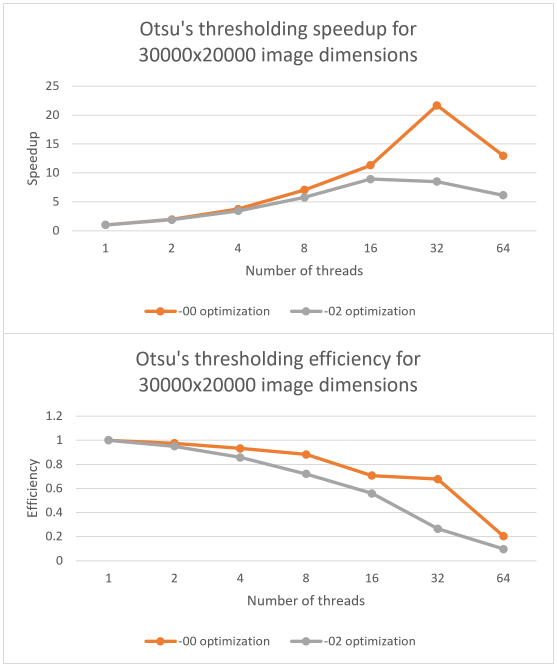

Figure 13: Οι τιμές επιτάχυνσης και απόδοσης της μεθόδου του Otsu.

5 Παραλληλοποίηση αλγορίθμων με CUDA

5.1 Το υλικό σύγκρισης

Για να παραλληλοποιηθούν οι αλγόριθμοι με CUDA χρειάστηκε να γίνουν ορισμένες αλλαγές στο κομμάτι του κώδικα. Συγκεκριμένα έπρεπε να ληφθεί υπόψιν πως αρχικά δεν παρουσιάζεται τόσο μεγάλο overhead για τη κλήση συναρτήσεων αλλά παρουσιάζονται περιορισμοί λόγω της καθυστέρησης των αντικειμένων στη μνήμη της GPU. Τέλος, χρήζει επιτακτικής ανάγκης να σημειωθεί πως εφόσον χρησιμοποιηθούν πολλαπλά blocks με τη GPU είναι δύσκολος ο συγχρονισμός τους. Λαμβάνοντας υπόψιν αυτά, κατασκευάστηκαν οι αλγόριθμοι με τις γλώσσες προγραμματισμού CUDA C. Για το υλικό σύγκρισης χρησιμοποιούνται οι κάρτες γραφικών GTX950, GTX1650(mobile) και RTX3080. Για σύγκριση χρησιμοποιούνται 64 Threads από τον επεξεργαστή Xeon.

5.2 Τα αποτελέσματα για τον Odd even Sort με CUDA

Για τον Odd even sort κατασκευάστηκαν 2 εκδοχές: Μια που αξιοποιεί συνεργατικά kernels και μια που δεν τα αξιοποιεί. Η ειδοποιός διαφορά τους είναι πως η έκδοση με τα συνεργατικά kernel μπορεί να αξιοποιήσει συγχρονισμό μεταξύ των kernel, αποφεύγοντας την εκκίνηση πολλαπλών kernel, μειώνοντας κατά πολύ την απόδοση, και έχει έως μειονέκτημα πως μπορεί να χρησιμοποιηθεί μόνο έως έναν ορισμένο αριθμό πίνακα. Η διαφορά αυτή φαίνεται ξεκάθαρα στα δεδομένα παρακάτω. Λόγου χάρη, για την RTX3080 που επιτρέπει τη χρήση των cooperative kernels έως και για περίπου 70.000 μέγεθος πίνακα, παρουσιάζεται μεγάλη διαφορά στους χρόνους εκτέλεσης μεταξύ των 60.000 και 100.000 στοιχείων.

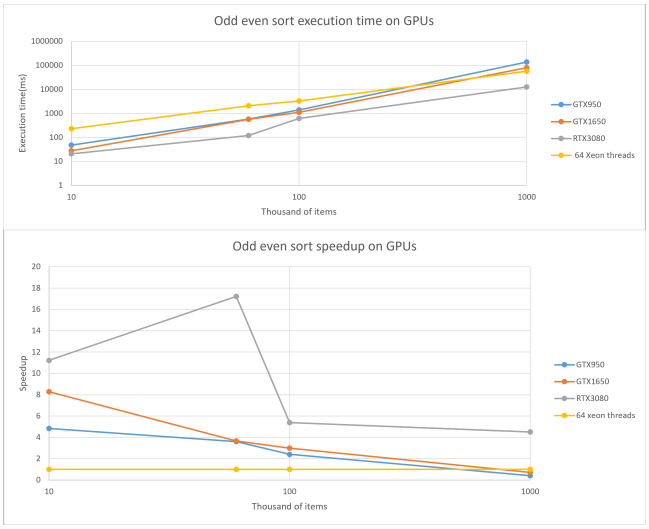

Figure 14: Οι τιμές εκτέλεσης και επιτάχυνσης της μεθόδου Odd even sort στη GPU.

5.3 Τα αποτελέσματα για τον Bucket sort με CUDA

Για την αναδιάταξη των δεξαμενών χρησιμοποιήθηκαν οι ενσωματωμένες τεχνικές sorting Radix Sort και Thrust Sort. Για τα πειραματικά δεδομένα επιλέχθηκε η Radix Sort διότι παρουσίασε καλύτερες ταχύτητες εκτέλεσης. Επίσης πρέπει να σημειωθεί πως λόγω ασυμβατότητας των driver δεν εξετάστηκαν οι GTX950 και RTX3080, διότι δε μπορούσαν να αξιοποιήσουν τις παραπάνω ενσωματωμένες μεθόδους. Παρ'όλα αυτά, καθώς ο Bucket Sort είναι πολύ πιο ευνοικός για κάρτες γραφικών εμφανίζει πολύ ταχύτερους χρόνους από την έκδοση της OpenMP.

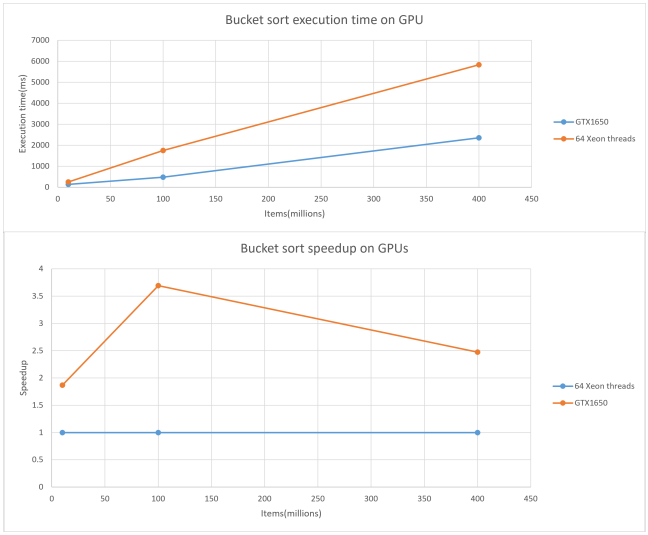

Figure 15: Οι τιμές εκτέλεσης και επιτάχυνσης της μεθόδου Bucket sort στη GPU.

5.4 Τα αποτελέσματα για τον αλγόριθμο του Otsu

Για τον αλγόριθμο του Otsu δεν περιμένουμε καμία επιτάχυνση κυρίως λόγω της μεγάλης καθυστέρησης που εμφανίζεται για την αντιγραφή και μεταφορά των δεδομένων στη GPU. Πράγματι αυτό δείχνουν και τα δεδομένα. Παρ'όλα αυτά, αναμφίβολα μπορεί να βελτιωθεί η απόδοσή του τροποποιώντας τον για να αξιοποιεί πιο ευνοϊκά τις δυνατότητες της GPU και να αποφευχθούν τα σειριακά κομμάτια και οι εντολές atomic.

Ο αλγόριθμος του Otsu για εικόνα 25000x17500			
GTX 950	GTX1650	RTX3080	64 Xeon Threads
374ms	385ms	198ms	35ms

6 Συμπεράσματα

Σύμφωνα με τα πειραματικά δεδομένα αλλά και το θεωρητικό υπόβαθρο είναι εμφανές πώς με την αξιοποίηση σύγχρονων μεθόδων, βιβλιοθηκών και εργαλειών μπορούμε με μεγάλη επιτυχία να επιταγχύνουμε σημαντικά την αποδοση των αλγορίθμων *Bubble sort*, *Bucket sort* και της μεθόδου του *Otsu* και να επιτύγχουμε χρόνους εκτέλεσης αδύνατους για σειριακές εκτελέσεις κώδικα. Παράλληλα, αξίζει να σημειωθεί πώς ο προγραμματιστής παίζει ακόμη μεγάλο μέρος στη κατασκευή και την απόδοση του προγράμματος, καθώς είναι υπεύθυνος για την βέλτιστη αξιοποίηση και εφόσον χρειαστεί, όπως στην περίπτωση του αλγορίθμου *Bubble sort* να τροποποιήσει τη μαθηματική μέθοδο με σκοπό να ευνοεί την παράλληλη επεξεργασία. Τέλος, είναι σημαντικό να σημειωθεί πως η σειριακή εκτέλεση έχει ακόμη ρόλο στη κατασκευή προγραμμάτων σε τομείς που δεν παρουσιάζουν ιδιαίτερη ανάγκη για χρήση έντονων πόρων αλλά και σε τομείς οι οποίοι δεν μπορούν να παραλληλοποιηθούν αποτελεσματικά.

References

[1] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[2] Rohit Chandra et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[3] HackerEarth. *Bubble Sort Algorithm Tutorial*. URL: `https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/tutorial`.

[4] Yernar Kumashev. *GPGPU Verification: Correctness of Odd-Even Transposition Sort Algorithm*. URL: `https://essay.utwente.nl/80585/1/Final_Research_Paper.pdf`.

[5] Rashita Mehta. *Bucket Sort Algorithm*. URL: `https://www.scaler.in/bucket-sort-algorithm/`.

[6] NOBUYUKI OTSU. *A Threshold Selection Method from Gray-Level Histograms*. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4310076`.

[7] Ricardo Rocha and Fernando Silva. *Parallel Sorting Algorithms*. URL: `https://www.dcc.fc.up.pt/~ricroc/aulas/1516/cp/apontamentos/slides_sorting.pdf`.

[8] Christopher Zelenka. *Parallel Merge Sort*. URL: `https://www.sjsu.edu/people/robert.chun/courses/cs159/s3/T.pdf`.