Τελική Εργασία 325

Name and surname: Γρηγόρης Γρηγοριάδης
Department: *Computer Science*
University:University of Cyprus

Location: Nicosia, Cyprus
Username and ID: ggrego04,

Abstract

Ο σκοπός αυτού του κειμένου, είναι η περιγραφή και ανάλυση όλων των εγασιών που υλοποιήθηκαν στο μάθημα ΕΠΛ 325. Επίσης θα συμπεριληφθεί και μια συγκριτική αξιολόγηση μεταξύ όλων των παράλληλων λύσεων και υλοποιήσεων που κάναμε κατά τη διάρκεια του εξαμήνου, για το πρόβλημα Any Char Range Count, με τη βοήθεια γραφικών παραστάσεων.

I. Εισαγωγη

Η παραλληλοποίηση σε ένα πρόγραμμα, μας επιφέρει μια πιο αποδοτική λύση στο πρόβλημά μας. Το πρόγραμμα γίνεται πού πιο γρήγορο με την χρήση παράλληλων μεθόδων κάθε μια από τις οποίες προσφέρει τα δικά της διαφορετικά στοιχεία κατά την υλοποίηση. Το εξαμηνιαίο μας πρόβλημα, μας ζητούσε να παίρνουμε ένα εύρος χαρακτήρων από το a μέχρι το z και να μετρούμε πόσες φορές εμφανίζεται κάθε χαρακτήρας μέσα σε ένα πίνακα από τυχαίους χαρακτήρες οι οποίοι προκύπτουν με την βοήθεια του αριθμού ταυτότητάς μας.

II. Τροπος Εργασιας Μας

A. *Σύνταξη κώδικα*

Το πρώτο βήμα σε κάθε άσκηση είναι η σύνταξη του κώδικα. Σε αυτό το βήμα το πρώτο πράγμα που κάναμε ήταν να μελετήσουμε τις διαλέξεις για να κατανοήσουμε την τεχνική που θα χρησημοποιούσαμε στην άσκηση. Μετά από εξονυχιστική μελέτη, γράψαμε τον κώδικα και ελέγξαμε ότι λειτουργεί σωστά.

B. *Εκτέλεση κώδικα*

Πριν την εκτέλεση αποφασίσαμε ότι θα έχουμε ένα ενιαίο πρόγραμμα στο οποίο σε κάθε άσκηση θα προσθέταμε πάνω τον καινούργιο μας κώδικα, έτσι μετά το compilation (το οποίο είναι gcc -mavx2 -lpthread -fopenmp -O3 AVX2_file.c), για να εκτελεστεί το πρόγραμμα, η ακόλουθη γραμμή θα έπρεπε να είχε εκτελεστεί, ./a.out TableSizeInMB charFrom charTo NumberOfThreads Method, όπου το TableSizeInMB αντιπροσωπεύει το μέγεθος του πίνακα στον οποίο θα περιέχονται όλοι οι χαρακτήρες, τα charFrom και charTo αντιπροσωπεύουν το σύνολο των χαρακτήρων που θα ψάχνουμε στον πίνακά μας, το NumberOfThreads αντιπροσωπεύει τον αριθμό threads που θα χρησιμοποιήσουμε και το Method είναι η μέθοδος που θα χρησιμοποιήθεί στην εκτέλεση. Η είσοδος της μεθόδου είναι ένας αριθμός από το 0 μέχρι το 8. Το 0 είναι η AVX2 μέθοδος, το 1 το pthreads, το 2 είναι το pthreads με dynamic allocation, το 3 είναι το pthreads σε συνδιασμό με AVX2 εντολές, το 4 είναι το pthreads σε συνδιασμό με AVX2 εντολές και dynamic allocation, το 5 είναι το OpenMP, το 6 είναι το OpenMP με dynamic allocation, το 7 είναι το OpenMP σε συνδιασμό με AVX2 εντολές και το 8 είναι το OpenMP σε συνδιασμό με AVX2 εντολές και dynamic allocation. Μετά τη σύνταξη του κώδικά μας, δημιουργήσαμε ένα script το οποίο θα μας βοηθούσε να τρέξουμε το πρόγραμμά μας τουλάχιστον 12 φορές για κάθε μέγεθος πίνακα (64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB) και για κάθε σύνολο χαρακτήρων, δηλαδή a-b, a-c, a-d…a-z.

Να σημειωθεί ότι η υλοποίηση με CUDA έγινε σε διαφορετικό αρχείο γιατί για να λειτουργήσει χρειάζεται αρχείο με extension .cu και compilation nvcc -O3 numberCountCUDA_Unified_solution.cu.

C. *Συλλογή μετρήσεων*

Αφού, τρέξαμε το πρόγραμμά μας για κάθε δυνατό συνδιασμό 12 φορές, αποθηκεύαμε τις μετρήσεις του χρόνου (και του perf stat – ανάλογα με το τι ζητούσε η άσκηση) σε .txt αρχεία. Να αναφερθεί επίσης ότι το πρόγραμμα εκτελέστηκε με τον ίδιο τρόπο και για τη σειριακή λύση του προβλήματος η οποία δινόταν από την άσκηση.

D. *Επεξεργασία μετρήσεων*

Μετά τη συλλογή των μετρήσεων περνούσαμε τα αποτελέσματά μας μέσα σε ένα spreadsheet στην Excel. Κάναμε 12 πίνακες οι οποίοι είχαν όλες τις μετρήσεις μας για όλα τα σύνολα χαρακτήρων και πέρναμε τους μέσους όρους του κάθε πίνακα. Ακολούθως κάναμε ένα καινούργιο πίνακα όπου πέρναμε τον μέσο όρο όλων των πιο πάνω πινάκων αφαιρώντας από το άθροισμα των μετρήσεων την πιο ψηλή μέτρηση και την πιο χαμηλή και διαιρούσαμε δια 10, έτσι είχαμε όσο το δυνατό πιο υψηλή ακρίβεια. Αυτό υλοποιήθηκε για την σειριακή υλοποίηση και για την παράλληλη. Ακολούθως, με τη βοήθεια του Amdahl’s Law υπολογίζαμε την αναμενόμενη επιτάχυνση του προγράμματός μας και έπειτα διαιρώντας τον μέσο όρο του χρόνου της σειριακής υλοποίησης με το μέσο όρο του χρόνου

της παράλληλης υλοποίησης πέρναμε την καταμετρημένη μας επιτάχυνση. Στο τέλος, υπολογίζαμε το Efficiency διαιρώντας τη θεωρητική επιτάχυνση με την καταμετρημένη.

Ε. Παρουσίαση αποτελεσμάτων

Μετά τον υπολογισμό των πιο πάνω τιμών,κατασκευάσαμε γραφική στην οποία απεικονίζαμε τις μετρήσεις του χρόνου εκτέλεσης της σειριακής υλοποίησης μαζί με τις μετρήσεις της παράλληλης υλοποίησης. Ακολούθως κάναμε γραφικές στις οποίες απεικονίζαμε την αναμενόμενη επιτάχυνση μαζί με την καταμετρημένη και το Efficiency σε σχέση με το μέγεθος του πίνακα. Τέλος κάναμε μια μεγάλη γραφική στην οποία απεικονίζαμε όλες τις επιταχύνσεις που πετύχαμε σε κάθε άσκηση για σκοπούς σύγκρισης.

III. ΑΝΑΠΤΥΞΗ ΚΑΙ ΥΛΟΠΟΙΗΣΗ ΠΑΡΑΛΛΗΛΗΣ ΛΥΣΗΣ

Σε αυτό το σημείο, θα περιγράψουμε τον κώδικα και την σκέψη μας που υλοποιήσαμε σε κάθε άσκηση ξεχωριστά.

Α. Υλοποίηση με AVX2 εντολές

Στην άσκηση αυτή μας ζητήθηκε να υλοποιήσουμε το πρόβλημα που αναφέραμε στην εισαγωγή, χρησιμοποιώντας εντολές AVX2. Έτσι φτιάξαμε μια μέθοδο η οποία παίρνει σαν παράμετρους το μέγεθος του πίνακα με όλους τους τυχαίους χαρακτήρες, το μέγεθος του, τον πίνακα στον οποίο θα αποθηκεύσουμε πόσες φορές εμφανίζεται ο κάθε χαρακτήρας, τον εναρκτήριο χαρακτήρα και τον τελικό χαρακτήρα του εύρους που θα μετρήσουμε.

Αρχικά ξεκινήσαμε ένα loop το οποίο περνούσε ανάμεσα από το εύρος των χαρακτήρων που θέλαμε. Βασικά παίρναμε ένα ένα τους χαρακτήρες και ελέγχαμε με τους χαρακτήρες στον πίνακα. Ακολούθως εκτελείται η εντολή_mm256_set1_epi8(c) η οποία μεατρέπει τον χαρακτήρα c στην μορφή __m256i που είναι ένα 256 bit vector που περιλαμβάνει αριθμούς (integers). Μετά ξεκινήσαμε ένα loop το οποίο πηγαίνει από το 0 μέχρι το μέγεθος του πίνακα διά 32. Σε αυτό το loop ορίζουμε και μηδενίζουμε το τοπικό counter sum καθώς επίσης και το 256i counter μας. Μέσα στο loop έχουμε ένα inner loop που θα πηγαίνει από το 0 μέχρι το 128 και θα αυξάνει την μεταβλητή του outer loop κατά ένα κάθε φορά. Μέσα στο loop φορτώσαμε αρχικά τον χαρακτήρα του πίνακα μετατρέποντας τον σε μορφή __m256i και με την εντολή _mm256_cmpeq_epi8 (x,y) τον συγκρίναμε με τον χαρακτήρα που ψάχνουμε. Η εντολή αυτή όμως επιστρέφει -1 αν οι δύο αριθμοί είναι ίσοι, έτσι εκτελούμε μια άλλη εντολή για να βεβαιωθούμε ότι ο αριθμός που επιστράφηκε δεν είναι -1, την εντολή _mm256_and_si256 (z, _mm256_set1_epi8(1)) η οποία κάνει and τους αριθμούς 1 και τον αριθμό που επιστράφηκε από την cmpeq. Ουσιαστικά η εντολή and ελέγχει μεταξύ δύο αριθμών, αν είναι και οι δύο 1 τότε επιστρέφει 1 και αυτό προστήθεται στο counter μας.

Μετά το τέλος του loop εκτελούνται πρώτα οι εντολές __m128i part1 = mm256_extracti128_si256 (s,0) και __m128i part2 = _mm256_extracti128_si256 (s,1). Ο ρόλος αυτών των εντολών είναι να κάνουν extract από τον 256 bit αριθμό τα δύο μέρη του σε 128 bits, με άλλα λόγια το 256 διαιρείται διά δύο και μένουν δύο κομμάτια των 128. Ακολούθως εκτελούνται οι εντολές __m256i p1 = _mm256_cvtepi8_epi16 (part1) και __m256i p2 = _mm256_cvtepi8_epi16 (part2) οι οποίες μας βοηθούν να κάνουμε extend τα δύο μέρη που κάναμε extract από 8 bits σε 16. Μετά κάνουμε horizontally add τέσσερις φορές ως εξής: __m256i result = _mm256_hadd_epi16 (p1,p2) result = mm256_hadd_epi16 (result,result) result = _mm256_hadd_epi16 (result,result) result = _mm256_hadd_epi16 (result,result). Αυτό γίνεται τέσσερις φορές γιατι κάθε φορά προσθέτει ζεύγη των 4 bits έτσι δύο ζεύγη των 16 bits χρειάζονται να προστεθούν τέσσερις φορές. Τέλος, προσθέτουμε τα δύο μέλη του 32 bit result μέσα στο τοπικό sum το οποίο μετά το τέλος του loop θα προστεθεί με τη σειρά του στην αντίστοιχη θέση στον πίνακα counters. Αυτή η διαδικασία επαναλαμβάνεται για κάθε χαρακτήρα που περιλαμβάνεται στο εύρος των χαρακτήρων που δηλώσαμε.

Β. Υλοποίηση με pthreads

Στην άσκηση αυτή μας ζητήθηκε να υλοποιήσουμε το πρόβλημα που αναφέραμε στην εισαγωγή, χρησιμοποιώντας pthreads. Έτσι φτιάξαμε μια μέθοδο η οποία παίρνει σαν παράμετρους το μέγεθος του πίνακα με όλους τους τυχαίους χαρακτήρες, το μέγεθος του, τον πίνακα στον οποίο θα αποθηκεύσουμε πόσες φορές εμφανίζεται ο κάθε χαρακτήρας, τον εναρκτήριο χαρακτήρα, τον τελικό χαρακτήρα που θα μετρήσουμε και τον αριθμό των threads που θα χρησιμοποιήσουμε. Στη συνέχεια δηλώσαμε ένα mutex, το οποίο θα μας βοηθούσε στο να είμαστε βέβαιοι ότι μόνο ένα thread έχει πρόσβαση σε μια κονή μεταβλητή κάθε φορά.

Σε αυτή την υλοποίηση σαν πρώτη ενέργειά μας δημιουργούμε ένα struct το οποίο θα περνούμε μέσα στον worker μας και θα περιέχει όλα τα απαραίτητα στοιχεία για να μπορεί ο worker να εργαστεί αναλόγος για κάθε thread. Αυτό το struct περιλαμβάνει, τον αριθμό ταυτότητας του thread, το μέγεθος στο οποίο θα εργαστεί το κάθε ένα μαζί με το κομμάτι του πίνακα που θα ελέγξει, τους δύο χαρακτήρες που αποτελούν το εύρος των χαρακτήρων που θα μετρήσουμε και τον πίνακα στον οποίο θα αποθηκεύσουμε το πόσες φορές εμφανίζεται ο κάθε ένας τους.

Το επόμενο στάδιο της υλοποίησής μας είναι η κατασκευή των pthreads. Δημιουργήσαμε ένα πίνακα που περιείχε τα threads μας, ώς επίσης και το struct του κάθε thread. Ακολούθως, χρησιμοποιήσαμε την εντολή pthread_create(&threads[i],&attr,any_Char_Range_Count_pthreads_worker,(void *)&args[i]) η οποία παίρνει σαν παράμετρους, το thread, τα attributes, που στην περίπτωσή μας και γενικά συνήθως είναι NULL, το όνομα του worker function που θα καλεστεί και το struct του κάθε thread. Με τη λήξη της εργασίας του worker εκτελούμε την εντολή pthread_join(threads[i], &status) η οποία χρησιμεύει για να περιμένουμε όλα τα threads να τερματίσουν.

Όσο αφορά το τι συμβαίνει μέσα στον worker, υπάρχουν τέσσερα είδη εργασίας για κάθε ένα από τα ζητήματα της άσκησης. Θα αναφερθούμε πρώτα στο σειριακή έκδοση των

pthreads, όπου το σειριακό κομμάτι της άσκησης εκτελείται παράλληλα σε κάθε thread ξεχωριστά με μέγεθος πίνακα το αρχικό μας μέγεθος διά τον αριθμό των threads, ούτως ώστε όλος ο φόρτος εργασίας να είναι ισοκατανεμημένος σε κάθε thread. Για να γίνει αυτό δημιουργήσαμε ένα πίνακα counter ο οποίος θα ήταν ένας τοπικός μετρητής του κάθε χαρακτήρα στον πίνακα. Μετά ξεκινήσαμε να διαπερνούμε το κομμάτι πίνακα που δώσαμε στο thread και ελέγχαμε αν ο χαρακτήρας που βρήσκαμε άνηκε στο εύρος χαρακτήρων που θέλαμε να ψάξουμε και προσθέταμε μια μονάδα στον τοπικό πίνακα που δημιουργήσαμε, στη θέση array[i] - 'a', δηλαδή αν βρήκαμε τον χαρακτήρα b στον πίνακα τότε b – a = 1, άρα στην θέση 1 του τοπικού counter θα προστεθεί μια μονάδα. Όταν τελειώσει αυτή η διαδικασία τότε κάνουμε lock το mutex που είχαμε δηλώσει στην αρχή του προγράμματός μας ούτως ώστε μόνο ένα thread να μπορεί ναενεργήσει πάνω σε μια κοινή μεταβλητή (στην περίπτωσή μας ο πίνακας counters που περιέχει τις μετρήσεις των εμφανίσεων του κάθε χαρακτήρα), μεταφέραμε τα δεδομένα του τοπικού counter στον shared πίνακα counters και στο τέλος κάναμε unlock το mutex για να μπορεί κάποιο άλλο thread να έχει πρόσβαση στον κοινό πίνακα και ακολούθως κάναμε exit το pthread.

Η ίδια διαδικασία επαναλήφθηκε για τα pthreads με εντολές AVX2 με την διαφορά ότι ο worker μας αντί να εκτελέσει το σειριακό κομμάτι του κώδικά μας, εκτέλεσε την υλοποίησή μας με εντολές AVX2 όπως την περιγράψαμε στο μέρος Α.

Για την υλοποίηση της σειριακής εκδοχής του pthreads με dynamic allocation ακολουθήσαμε τα ίδια βήματα με το non-dynamic με διαφορά τα δεδομένα στο struct και το worker. Στο struct για πίνακα βάζαμε όλο τον αρχικό πίνακα με τους χαρακτήρες και όχι κομμάτι του οπως δίναμε στο non-dynamic και σαν μέγεθος πίνακα το αρχικό επίσης. Αυτό γινόταν επειδή στο δυναμικό καταμερισμό εργασίας δεν έχουμε σταθερό μέγεθος πίνακα και το υπολογίζουμε δυναμικά σε κάθε thread. Για να κάνουμε την ζωή μας πιο εύκολη δημιουργήσαμε μια μέθοδο getChunkStart η οποία μας βοηθούσε να υπολογισουμε από πιο σημείο του πίνακα θα ξεκινούσε το thread το ψάξημό του. Σε αυτή την μέθοδο στέλναμε το struct που δίναμε και στον worker. Χρησιμοποιούσαμε μια global μεταβλητή στην οποία προσθέταμε το στατικό chunk size που θέλαμε και ορίσαμε στην αρχή του προγράμματός μας. Αν η τιμή της global αυτής μεταβλητής ήταν μικρότερη από το μέγεθος του πίνακά μας, δηλαδή δεν φτάσαμε στο τέλος του, τότε επέστρεφε πίσω το σημείο που θα ξεκινούσε το chunk στον πίνακα. Αν το υπερέβαινε, τότε έστελνε πίσω -1 για να τερματιστεί το thread.

Ο worker μας ξεκινούσε ένα 'infinite' loop το οποίο θα σταματούσε όταν τελείωνε το thread. Σε αυτό το loop καλούνταν η μέθοδος getChunkStart για να δούμε από που θα ξεκινούσε/συνεχίζε το thread, αφότου κάναμε lock το mutex επειδή η global μεταβλητή που χρησιμοποιούσαμε στη μέθοδο μπορούσε να γίνει accessed από οποιδήποτε thread. Μετά το κάλεσμα της μεθόδου getChunkStart κάναμε unlock mutex. Πριν να ξεκινήσει η διαδικασία ελέγχου στον πίνακα, ελέγξαμε πρώτα ότι η τιμή που πήραμε από την getChunkStart σύν το σταθερό chunk size που δηλώσαμε στην αρχή του προγράμματος, δεν υπερβαίνουν το μέγεθος του πίνακα. Αν το υπερβαίνουν τότε η διαδικασία ελέγχου του πίνακα θα γίνει από το chunk start μέχρι το τέλος του πίνακα. Αν είναι μικρότερο από το μέγεθος του πίνακα τότε ο έλεγχος του πίνακα θα γίνει από το chunk start μέχρι το chunk start + constant chunk size. Όταν η getChunkStart επιστρέψει -1 το thread τερματίζεται.

Η ίδια διαδικασία επαναλήφθηκε για τα pthreads με εντολές AVX2 και dynamic allocation, με την διαφορά ότι ο worker μας αντί να εκτελέσει το σειριακό κομμάτι του κώδικά μας, εκτέλεσε την υλοποίησή μας με εντολές AVX2 όπως την περιγράψαμε στο μέρος Α.

C. Υλοποίηση με OpenMP

Στην άσκηση αυτή μας ζητήθηκε να υλοποιήσουμε το πρόβλημα που αναφέραμε στην εισαγωγή, χρησιμοποιώντας εντολές OpenMP. Έτσι φτιάξαμε μια μέθοδο η οποία παίρνει σαν παράμετρους τον πίνακα με όλους τους τυχαίους χαρακτήρες, το μέγεθος του, τον πίνακα στον οποίο θα αποθηκεύσουμε πόσες φορές εμφανίζεται ο κάθε χαρακτήρας, τον εναρκτήριο χαρακτήρα, τον τελικό χαρακτήρα που θα μετρήσουμε και τον αριθμό των threads που θα χρησιμοποιήσουμε. Στη συνέχεια δηλώσαμε ένα mutex, το οποίο θα μας βοηθούσε στο να είμαστε βέβαιοι ότι μόνο ένα thread έχει πρόσβαση σε μια κονή μεταβλητή κάθε φορά.

Μέσα στην μέθοδο αυτή ξεκινούμε το παράλληλο κομμάτι με την εντολή #pragma omp parallel num_threads(numThreads) private(i){} μέσα στην οποία θα γραφτεί ο υπόλοιπος κώδικας. Ουσιαστικά σε αυτό το κομμάτι περνούμε τον αριθμό των threads που θα χρησιμοποιήσουμε και ένα private i (δηλαδή μια μεταβλητή η οποία θα χρησιμοποιείται μόνο μέσα στο παράλληλο κομμάτι). Ακολούθως, δηλώσαμε ένα τοπικό πίνακα ο οποίος θα ενεργούσε σαν το counter μας για αυτό το thread. Στη συνέχεια, ακολούθησε η εντολή #pragma omp for schedule(static) η οποία δείχνει πως θα ακολουθήσει ένα for loop με στατικό μέγεθος πίνακα. Μετά, μέσα στο loop εκτελέστηκε ο κώδικας του σειριακού μέρους του προγράμματός μας και χρησιμοποιήθηκε σαν counter ο τοπικός πίνακας. Μετά από το loop αυτό, ακολούθησε το critical μέρος του κώδικα το οποίο καλείται με την εντολή #pragma omp critical και ελέγχει ότι μόνο ένα thread λειτουργεί κάθε φορά μέσα σε αυτό. Μέσα στο critical σημείο προσθέσαμε τα δεδομένα του τοπικού πίνακα στον κοινό πίνακα των threads.

Όσο αφορά την υλοποίηση με dynamic allocation, ακολουθήθηκαν όλα τα πιο πάνω βήματα εκτός από το σημείο που καλέσαμε το παράλληλο for loop. Η εντολή που χρησιμοποιήθηκε για δυναμικό καταμερισμό εργασίας ήταν η #pragma omp for schedule(dynamic, size/numThreads) , με άλλα λόγια αντί να προσθέσουμε τη λέξη static, προσθέσαμε την λέξη dynamic μαζί με το μέγεθος του πίνακα που θα χρησιμοποιούσε το κάθε thread. Στην περίπτωσή μας επιλέξαμε να χρησιμοποιήσουμε σαν μέγεθος το αρχικό μέγεθος του πίνακα διά τον αριθμό των threads ούτως ώστε να έχουμε ίσο καταμερισμό εργασίας.

Οι δύο υλοποιήσεις με εντολές AVX2 ακολούθησαν όλα τα πιο πάνω βήματα για στατικό και δυναμικό καταμερισμό εργασίας αντίστοιχα με την διαφορά ότι ο κώδικας που εκτελέστηκε στο παράλληλο for loop ήταν ο κώδικας που χρησιμοποιήσαμε στην πρώτη μας άσκηση και περιγράψαμε πιο πάνω.

D. *Υλοποίηση με CUDA*

Η άσκηση αυτή αποτελεί εξαίρεση στον τρόπο εκτέλεσης για τον λόγο ότι για να εκτελεστεί μια CUDA εντολή χρειάζεται αρχείο με .cu extension. Το compilation είναι nvcc -O3 file.cu και για να εκτελεστεί πρέπει να ακολουθήσει η γραμμή a.out ArraySize from to.

Φτιάξαμε μια μέθοδο η οποία παίρνει σαν παράμετρους τον πίνακα με όλους τους τυχαίους χαρακτήρες, το μέγεθος του, τον πίνακα στον οποίο θα αποθηκεύσουμε πόσες φορές εμφανίζεται ο κάθε χαρακτήρας, τον εναρκτήριο χαρακτήρα, τον τελικό χαρακτήρα που θα μετρήσουμε, τον αριθμό των blocks που θα χρησιμοποιήσουμε και τον αριθμό των threads ανά block.

Πρώτα πρώτα υπολογίζουμε τον συνολικό αριθμό από threads που θα χρησιμοποιήσουμε, ο οποίος είναι ίσος με blocks*threads per block, μετά δημιουργούμε δύο πίνακες. Ο πρώτος πίνακας θα είναι το τοπικό counter και ο δεύτερος θα περιέχει όλες τις πληροφορίες που θα είναι χρήσιμες για τον kernel μας. Ο πρώτος πίνακας θα είναι περίπου κοινός μεταξύ των blocks, το μέγεθός του θα είναι ίσο με 26*blocks, δηλαδή όσα είναι όλα τα γράμματα επί τον αριθμό των blocks. Ο τρόπος αυτός επιλέχθηκε γιατί δεν ήταν εφικτό να περάσουμε δισδιάστατο πίνακα μέσα στον kernel όπως ήταν και η σκέψη μας αρχικά, έτσι το σημείο στο οποίο θα ενεργούσε το κάθε block θα ήταν από το 26*blockID μεχρι το 26*blockID + 26. Ο δεύτερος πίνακας που φτιάξαμε λέγετε Info και έχει μέγεθος 4 όσες και οι πληροφορίες που θα χρειαστεί ο kernel. Οι πληροφορίες αυτές θα είναι το μέγεθος του πίνακα, ο συνολικός αριθμός από threads, ο αρχικός χαρακτήρας που θα ψάξουμε και ο τελικός. Ακολούθως οι πίνακες έγιναν malloc με τις εντολές cudaMallocManaged(&Counters_CUDA, sizeof(unsigned int)*blocks*26) και cudaMallocManaged(&Info, sizeof(unsigned int)*4) για να μπορούν να χρησιμοποιήθουν από τον kernel. Οι επόμενες εντολές dim3 dimBlock(threads) και dim3 dimGrid(blocks) εκτελέστηκαν για να δημιουργηθεί το grid στο οποίο θα εργαστεί ο kernel. Στη συνέχεια καλέστηκε η μέθοδος του kernel με την εντολή numCountKernel<<<dimGrid, dimBlock>>>(theArray,Counters_CUDA,Info) που ουσιαστικά δίνουμε στον kernel τις διαστάσεις του grid και τις παραμέτρους (τον πίνακα με τους χαρακτήρες, τον τοπικό πίνακα counters και τον πίνακα info). Μετά τον kernel χρησιμοποοιήθηκε η εντολή cudaThreadSynchronize() η οποία περιμένει όλα τα threads να τελειώσουν. Και τέλος, προσθέτουμε τα δεδομένα του πίνακα counters στον τελικό πίνακα μας.

Η μέθοδος kernel παίρνει σαν παραμέτρους ότι αναφέραμε και πιο πάνω και ακολούθως παίρνει το threadID του κάθε thread μέσα στο block. Αυτό το threadID θα χρησιμοποιηθεί σαν σημείο εκκίνησης ελέγχου του πίνακα με τους χαρακτήρες και κάθε φορά θα αυξάνεται με τον συνολικό αριθμό των threads. Μετά ακολουθεί ο έλεγχος του χαρακτήρα και προστήθεται στον τοπικό πίνακα με την εντολή atomicAdd(&dCounters[dArray[i]-'a'+26*blockID],1). Δηλαδή 26*blockID είναι η θέση του πίνακα από την οποία ξεκινά το counter του block και dArray[i]-'a' είναι η θέση του counter του κάθε χαρακτήρα.

IV. ΜΕΤΡΗΣΕΙΣ

Σε αυτό το σημείο, θα περιγράψουμε τον τρόπο με τον οποίο πήραμε τις μετρήσεις μας όσον αφορά την θεωρητική και καταμετρημένη επιτάχυνση και θα συγκρίνουμε τις επιταχύνσεις των παράλληλων υλοποιήσεων μας με τη βοήθεια γραφικών παραστάσεων. Ώς πρός τον τρόπο υπολογισμού των δύο επιταχύνσεων θα χρησιμοποιηθεί η υλοποίηση με εντολές AVX2.

A. *Θεωρητική Επιτάχυνση*

Σε πρώτο στάδιο κάναμε compile το πρόγραμμά μας με την εντολή gcc -mavx2 -O3 AVX2_Any_Char_Range_Count_std.c –S. Η πιο πάνω εντολή μετατρέπει τον κώδικά μας σε κώδικα assembly. Ο λόγος που το κάναμε αυτό ήταν για να μπορέσουμε να μετρήσουμε όλα τα instructions του προγράμματός μας.

Το σειριακό μέρος του προγράμματός μας καλεί 2 μέρη, το πρώτο είναι το initialization του πίνακα, ο οποίος γεμίζει με αριθμούς σειριακά και μετά καλείται ο υπολογισμός του sum. Για την παράλληλη μας υλοποίηση καλούμε επίσης 2 μέρη, το initialization του πίνακά μας και τον υπολογισμό του sum με παράλληλη επεξεργασία.

Για τον υπολογισμό του θεωρητικού speedup θα χρησιμοποιήσουμε τον τύπο:

*Runtime = (Instructions/Program) * (Cycles/Instruction) * (Seconds/Cycle).*

Όπου, το Runtime, συμβολίζει το θεωρητικό speedup, το Seconds/Cycle και το Cycles/Instruction είναι ίσο με 1.

Μετά το compilation με -S, μετρήσαμε τις γραμμές κώδικα στην assembly έκδοσή του για την σειριακή μέθοδο και βρήκαμε ότι αρχικά εκτελούνται 12 εντολές και μετά 9 μέσα στο loop το οποίο εκτελείται n φορές. Επαναλάβαμε το ίδιο και για τη AVX2 μέθοδο και βρήκαμε ότι αρχικά εκτελούνται 5 εντολές μετά μέσα στο πρώτο μας loop εκτελούνται 10 εντολές, ακολούθως μέσα στο επόμενο loop (n/(32*128)) εκτελούνται 45 εντολές και στο τελευταίο inner loop εκτελούνται 20 εντολές 128 φορές. Άρα η πράξη που θα ακολουθήσουμε για Θεωρητικό speed up είναι αυτή:

5+(to-from)(10+(n/(32*128)*(45+128*20)))*

Το αποτέλεσμα για AVX2 εντολές ήταν ίσο με 2. Άρα και η θεωρητική επιτάχυνση είναι ίση με 2 για εύρος χαρακτήρων από a μέχρι z.

Με τον ίδιο τρόπο συνεχίσαμε και στις υπόλοιπες ασκήσεις.

B. *Καταμετρημένη Επιτάχυνση*

Για να υπολογισουμε την καταμετρημένη επιτάχυνση εκτελέσαμε το πρόγραμμα 12 φορές για τη σειριακή μέθοδο και για την παράλληλη με τον τρόπο που αναφέραμε στον τρόπο εργασίας μας. Ακολούθως πήραμε μέσο όρο για σειριακό και παράλληλο πρόγραμμα για κάθε μέγεθος πίνακα ξεχωριστά και

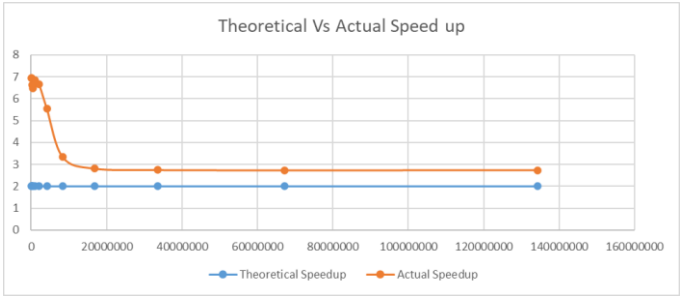
διαιρέσαμε τον χρόνο εκτέλεσης του σειριακού με τον χρόνο εκτέλεσης του παράλληλου. Για το δικό μας παράδειγμα με τις εντολές AVX2 η καταμετρημένη επιτάχυνση για εύρος χαρακτήρων από a μέχρι z είναι 2.74 ενώ η θεωρητική 2.

C. *Αποτελεσματικότητα (Efficiency)*

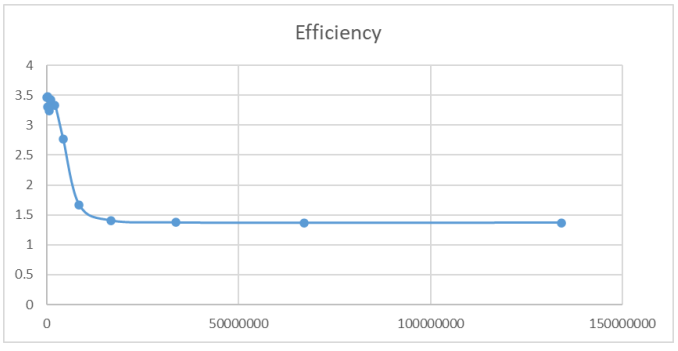
Για να υπολογισουμε την αποτελεσματικότητα του παράλληλου κώδικά μας απλά διαιρέσαμε την καταμετρημένη επιτάχυνση με την θεωρητική. Για το πρόγραμμα με AVX2 εντολές βρήκαμε περίπου 1.4, αυτό σημαίνει ότι το Efficiency του προγράμματος μας είναι 140%, το οποίο είναι πολύ μεγάλο. Άρα το πρόγραμμά μας είναι efficient. Γενικά ότι efficiency περισσότερο από 50% βρούμε είναι καλό.

D. *Γραφικές Παραστάσεις*

Μετά από τους πιο πάνω υπολογισμούς ακολούθησε η αναπαράστασή τους πάνω σε γραφική παράσταση, συγκρίνοντας έτσι την θεωρητική με την καταμετρημένη επιτάχυνση. Φαίνεται το παράδειγμα με εντολές AVX2 πιο κάτω:



Επίσης χρησιμοποιήσαμε γραφικές για την αναπαράσταση της αποτελεσματικότητάς ως εξής:



V. ΣΥΓΚΡΙΣΕΙΣ ΜΕΤΑΞΥ ΠΑΡΑΛΛΗΛΩΝ ΛΥΣΕΩΝ

A. *Σύγκριση ως προς τις επιταχύνσεις*

Σε αυτό το σημείο, θα συγκρίνουμε τις παράλληλες λύσεις μας ως πρός την δυσκολία υλοποίησης. Στο section appendix φαίνεται η γραφική παράσταση που περιέχει τις μετρήσεις της επιτάχυνσης όλων των ασκήσεων.

Παρατηρούμε ότι η επιτάχυνση μας στην άσκηση με εντολές AVX2 ξεκινά σε πολύ υψηλό σημείο και σιγά σιγά πέφτει πρός τα αναμενόμενα επίπεδα, δηλαδή πρός την θεωρητική μας επιτάχυνση.

Στην επόμενη άσκηση, στην οποία ασχοληθήκαμε με pthreads, υλοποιήθηκαν 4 μεθόδοι. Στη μια ο κώδικας που υλοποιήθηκε ως pthreads ήταν ο σειριακός με προκαθορισμένο αριθμό από threads, στη $2^{η}$ ο κώδικας που υλοποιήθηκε ως pthreads ήταν ο σειριακός και πάλι, με τη διαφορά ότι ο το μέγεθος του κομματιού του πίνακα που έλεγχε το κάθε thread δίνονταν σε αυτό δυναμικά σαν chunks. Στο $3^{ο}$ ζήτημα, ο κώδικας που υλοποιήθηκε ως pthreads ήταν ο AVX2 από την πρώτη εργασία και στο $4^{ο}$ ζήτημα, ο κώδικας που υλοποιήθηκε ως pthreads ήταν ο AVX2 με δυναμική κατανομή μεγέθους πίνακα σε κάθε thread. Η θεωρητική μας ήταν 4 και φαίνεται μαζί με τις άλλες τέσσερις στην πιο πάνω γραφική. Παρατηρούμε ότι όλες οι επιταχύνσεις μας ξεκινάνε χαμηλά. Η καταμετρημένη επιτάχυνση pthreads με σειριακό και η καταμετρημένη επιτάχυνση pthreads με AVX2, όσο μεγαλώνει το μέγεθος του πίνακά μας, ανεβαίνουν στα αναμενόμενα επίπεδα, σε αντίθεση με τις δυναμικές τους υλοποιήσεις που βρίσκονται πολύ πιο χαμηλά.

Στην Τρίτη εργασία μας ζητήθηκε να υλοποιήσουμε το πρόβλημα με τη χρήση OpenMP. Και αυτή η άσκηση με τη σειρά της ζητούσε τέσσερις υλοποιήσεις. Στη μια ο κώδικας που υλοποιήθηκε ως OMP ήταν ο σειριακός με προκαθορισμένο αριθμό από threads, στη 2η ο κώδικας που υλοποιήθηκε ως OMP ήταν ο σειριακός και πάλι, με τη διαφορά ότι ο το μέγεθος του κομματιού του πίνακα που έλεγχε το κάθε thread δίνονταν σε αυτό δυναμικά. Στο 3ο ζήτημα, ο κώδικας που υλοποιήθηκε ως OMP ήταν ο AVX2 από την πρώτη εργασία και στο 4ο ζήτημα, ο κώδικας που υλοποιήθηκε ως OMP ήταν ο AVX2 με δυναμική κατανομή μεγέθους πίνακα σε κάθε thread. Η θεωρητική μας επιτάχυνση για τις υλοποιήσεις με το σειριακό κώδικα ήταν 4, ενώ για τις υλοποιήσεις με τον AVX2 κώδικα η θεωρητική επιτάχυνση ήταν 8. Όλες οι πιο πάνω επιταχύνσεις φαίνονται στην γραφική μας. Παρατηρούμε ότι οι καταμετρημένες επιταχύνσεις των υλοποιήσεων OMP με το σειριακό τρόπο, ξεκινούν χαμηλά αλλά σιγά σιγά πλησιάζουν στην αναμενόμενη επιτάχυνσή τους. Η καταμετρημένη επιτάχυνση της OMP υλοποίησης με AVX2 κώδικα και δυναμικό καταμερισμό μεγέθους πίνακα παραμένει πιο χαμηλά από την αναμενόμενη της, ενώ η καταμετρημένη επιτάχυνση της OMP υλοποίησης με AVX2 κώδικα χωρίς δυναμική κατανομή φόρτου, παρόλο που ξεκίνησε χαμηλά, με την αύξηση του μεγέθους του πίνακα ήρθε στα αναμενόμενα επίπεδα.

Στην τέταρτη άσκηση, μας ζητήθηκε να υλοποιήσουμε το πρόβλημα με τη χήση εντολών CUDA. Παρατηρούμε ότι η επιτάχυνση που πήραμε από την υλοποίηση με CUDA είναι η

πιο ψηλή. Αυτό σημαίνει ότι η υλοποίηση μας αυτή, είναι η πιο αποδοτική.

B. Σύγκριση ως προς τη δυσκολία υλοποίησης

Από την πιο εύκολη υλοποίηση στην πιο δύσκολη κατά την άποψή μου η σειρά είναι η εξής: OpenMP, pthreads, CUDA, AVX2.

Η OpenMP μου φάνηκε πιο εύκολη στην υλοποίηση από όλες τις άλλες γιατι στον παράλληλο κώδικα απλά πρόσθεταμε 3 καινούργιες γραμμές κώδικα και επίσης η αλλαγή από στατικό καταμερισμό εργασίας σε δυναμικό γινόταν αλλάζοντας μόνο μια μεταβλητή (από static σε dynamic μαζί με μέγεθος πίνακα).

Η υλοποίηση με pthreads μου φάνηκε κι αυτή με τη σειρά της εύκολη γιατί απλά χρειαζόταν να δημιουργήσεις ένα struct το οποίο θα είχε τα σημαντικά στοιχεία που θα χρησιμοποιούσε το κάθε thread, να δημιουργήσεις και να καλέσεις κάθε thread με το worker του και τέλος να κάνεις join τα threads.

Η υλοποίηση με CUDA δεν μου φάνηκε ούτε αυτή πάρα πολή δύσκολη, χρειάζεται απλά προσοχή όταν καλείται ο kernel, δημιουργούμε grid με βάση threads και blocks, κάνουμε cudamalloc στους πίνακες μας. Δεν στέλνουμε δισδιάστατο πίνακα στον kernel και τέλος μετά το synchronization, κάνουμε πάντα cudaFree τους πίνακες που χρησιμοποιήσαμε.

Η υλοποίηση με εντολές AVX2 για μένα ήταν η πιο δύσκολη στην υλοποίηση γιατί έπρεπε να προσέχω πολύ το μέγεθος στο οποίο θα κινούνται τα loops. Επίσης η λογική με την οποία ενεργούσαν οι εντολές AVX2 ήταν περίπλοκη και την βρήκα δύσκολη στην κατανόηση και τέλος υπήρχαν πολλές και διάφορες περίπλοκες εντολές με διαφορετικά μεγέθη σε bits.

C. Σύγκριση ως προς τον καταμερισμό εργασίας

Γενικά όποιαδήποτε υλοποίηση με δυναμικό allocation είχε καλύτερο καταμερισμό εργασίας επειδή στην περίπτωση αυτή κανένα νήμα δεν έμενε χωρίς δουλειά σε κανένα χρονικό σημείο. Αυτό έρχεται σε αντίθεση με το στατικό allocation, στο οποίο όταν ένα thread τελείωνε την εργασία του τότε περίμενε χωρίς να κάνει τίποτα. Στο δυναμικό δηλαδή όταν ένα νήμα τελείωνε τότε το σύστημα του παραχωρούσε απευθείας το επόμενο κομμάτι πίνακα που θα έλεγχε. Στο δυναμικό όμως είχαμε overhead που δεν είχαμε στο σειριακό.

REFERENCES

Χρησιμοποιήθηκε το conference template του IEEE.

APPENDIX