HPC - Project

Function Approximation using KNN

Η Εργασία εκπονήθηκε από τους:

- Σφήκας Θεόδωρος, 1072550
- Τσούλος Βασίλειος, 1072605

❖ Εισαγωγή:

System Specifications:

Το πρόγραμμα για την παράδοση των τελικών αποτελεσμάτων εκτελέστηκε μέσω remote connection στο υπολογιστικό σύστημα που προσφέρεται στα πλαίσια του μαθήματος σε συνδιασμό με τον προσωπικό μας υπολογιστή. Συνοπτικά τα χαρακτηριστικά των συστημάτων αναφέρονται στην συνέχεια.

» AMD Ryzen 5600x :	» Intel Core i7-3770 :	» Nvidia GPU Tesla K40c :
# of CPU Cores → 6	# of CPU Cores → 4	# of Cuda Cores → 2880
# of Threads → 12	# of Threads → 8	# of SMXs → 15
Boost Clock → 4.6 GHz	Boost Clock → Up to 3.9 GHz	Compute Capability → 3.5
Base Freq → 3.7 GHz	Base Freq → 3.4 GHz	Base Freq → 745 MHz
L1 Cache → 64 KB *(per core)*	L1 Cache → 128 KB *(per core)*	L1 Cache → 128 KB *(per SMX)*
L2 Cache → 512 KB *(per core)*	L2 Cache → 1 MB *(per core)*	L2 Cache → 1536 KB *(shared)*
L3 Cache→ 32 MB *(shared)*	L3 Cache → 8 MB *(shared)*	*DRAM* → 12GB

Οδηγίες εκτέλεσης:

Μέσα στο παραδοτέο εμπεριέχεται ένας υπό-φάκελος για κάθε διαφορετική υλοποίηση που πραγματοποιήσαμε. Αναλυτικά εμπεριέχεται κώδικας με την χρήση παραλληλισμού σε OpenMP, MPI, Cuda και OpenACC πέρα από την βελτιστοποιημένη σειριακή υλοποίηση. Για το compilation process προσφέρονται Makefiles με τις αντίστοιχες ρυθμίσεις. Τέλος εμπεριέχεται για κάθε περίπτωση ένα bash script file με μία παραδειγματική εκτέλεση. Κρίνουμε σημαντικό να αναφέρουμε πως ανάλογα με τις δυνατότητες του κάθε επεξεργαστή ορισμένα compilation flags μπορεί να μην υποστηρίζονται.

Παραδοχές:

1) Με στόχο την καλύτερη κατανόηση των μεταβολών που πραγματοποιήθηκαν στο κομμάτι του κώδικα επιλέξαμε να αφαιρέσουμε από τα παραδοτέα αρχεία βοηθητικές συναρτήσεις που δεν χρησιμοποιήθηκαν.
2) Λόγω αδυναμίας εκτέλεσης *AVX* εντολών στον επεξεργαστή που μας δίνεται access, παρά την θεωρητική υποστήριξη των αντίστοιχων features και intrinsics, επιλέξαμε να αναδείξουμε τα αποτελέσματα και τις βελτιστοποιήσεις που επιτεύχθηκαν με την χρήση των παραπάνω μέσω του προσωπικού μας επεξεργαστή μόνο στις απολύτως απαραίτητες εκτελέσεις. *[Error Code: 1082344 illegal instruction].* Με στόχο την μέτρηση ορθών στατιστικών και συγκρίσεων ανεξαρτήτως του μηχανήματος χρησιμοποιείται η μέτρηση της εκτέλεσης του σειριακού κώδικα στον αντίστοιχο επεξεργαστή. (Κάναμε προσπάθεια να compile-άρουμε τον κώδικα με την χρήση του *-mavx* flag το οποίο από την εκτέλεση της *lscpu* φαίνεται να υποστηρίζεται, ωστόσο οδηγηθήκαμε στο ίδιο μήνυμα λάθους.)
3) Εφόσον στον δικό μας υπολογιστή δεν διαθέτουμε cuda capability, οι εκτελέσεις και οι συγκρίσεις αποτελεσμάτων που απαιτούν την χρήση GPU έγιναν αποκλειστικά μέσω του remote access μηχανήματος.

4) Συμπληρωματικά των προηγούμενων παραδοχών θέλουμε να τονίσουμε, πως για τις μετρήσεις των *speed up* και του *efficiency* που ακολουθούν χρησιμοποιείται κάθε αλγοριθμική βελτιστοποίηση που πραγματοποιήθηκε στον σειριακό κώδικα. Υπολογίζουμε την βελτιστοποίηση της παραλληλοποίησης που επιτεύχθηκε και όχι την χρονοβελτίωση συγκριτικά με τον αρχικό κώδικα.
5) Με στόχο την μέτρηση των αποτελεσμάτων αποφασίσαμε να χρησιμοποιήσουμε καθολικά *1048576 training elements* και *1024 query elements*. Έχουν πραγματοποιηθεί παντού δοκιμές για το scalability του κώδικα και η μέτρηση των παραπάνω τιμών αρχικά φαίνονταν επαρκείς για την εξαγωγή ουσιαστικών αποτελεσμάτων, καθώς και οι χρόνοι αρκετά μεγάλοι ώστε να μην επηρεάζονται ιδιαίτερα από την τυχαιότητα κάθε εκτέλεσης.

❖ Serial Code Optimizations

Οφείλουμε να αναφέρουμε πως λόγω χρονικών πιέσεων επικεντρωθήκαμε σε πολύ βασικές βελτιστοποιήσεις της διαδικασίας που θα οδηγούσαν σε μετρήσιμη χρονοβελτίωση. Δυστυχώς δεν προλάβαμε δηλαδή να μεταβάλλουμε τον τρόπου παραγωγής δεδομένων σε binary αρχείο ή το παράλληλο διάβασμα αυτού, ούτε επιλέξαμε να μεταβάλλουμε αισθητά την δομή της Knn διαδικασίας. Συγκεκριμένα αναλύοντας την αλγοριθμική "διαδρομή" και μέσω *naive profiling* διαπιστώνεται πως το μεγαλύτερο workload βρίσκεται στον υπολογισμό του estimation της τιμής σε κάθε query element, συμπεριλαμβάνοντας την διαδικασία εύρεσης των αποστάσεων του με όλα τα training elements. Στην συνέχεια ελέγχονται οι προαναφερθείς αποστάσεις με στόχο την εύρεση ορισμένου αριθμού *(NNBS)* ελαχίστων. Σε κάθε επανάληψη για τον έλεγχο κάθε training element με την μέγιστη απόσταση που έχει συγκρατηθεί μεταξύ των μικρότερων *NNBS* επανεξετάζαμε την μέγιστη τιμή. Η παραπάνω διαδικασία ήταν αχρείαστη σε μεγάλη πλειοψηφία επαναλήψεων καθώς απαιτείται το κάλεσμα της συνάρτησης *" compute_max_position() "* μόνο σε περίπτωση μεταβολής των *NNBS* μικρότερων αποστάσεων που συγκρατούμε. Η αλλαγή μίας γραμμής κώδικα μετέφερε *(προσεγγιστικά)* τον χρόνο εκτέλεσης του σειριακού κώδικα από 40 δευτερόλεπτα στα 11, στον υπολογιστή που κάνουμε remote access. Η χρονοβελτίωση ήταν τόσο μεγάλη που θεωρήσαμε πως δεν απαιτούταν περαιτέρω μεταβολή της αλγοριθμικής διαδικασίας πέρα από το να αφαιρέσουμε το sorting των NNBS κοντινότερων training elements καθώς δεν χρησιμοποιούμε weighted sums. Επιπρόσθετα για την μείωση του χρόνου του προαναφερθέντος workload, μετατρέψαμε την συνάρτηση της compute distance ώστε να εκμεταλλεύεται SIMD κώδικα. Ωστόσο όπως αναφέραμε και στις παραδοχές παρουσιάστηκε αδυναμία εκτέλεσης του συγκεκριμένου κώδικα στο υπολογιστικό σύστημα που μας παραχωρείται. Προκειμένου να αναδείξουμε τα συγκεκριμένα αποτελέσματα στην συνέχεια θα συνοδευτούν μετρήσεις και εκτελέσεις που πάρθηκαν από τον επεξεργαστή Ryzen 5 5600x.

```
// Brute force -> find the knn
for (i = 0; i < TRAINELEMS; i++) {
    new_d = compute_euclidean_distance(xquery, &xtr[i*prob_dim], prob_dim);
    if (new_d < max_d) {
        nn_x[max_i] = i;
        nn_d[max_i] = new_d;
        max_d = compute_max_position(nn_d, knn, &max_i);
    }
}
```

❖ OpenMp

Με στόχο την παραλληλοποίηση του βελτιστοποιημένου σειριακού κώδικα μέσω threading θα μπορούσαμε να ακολουθήσουμε δύο μεθοδολογίες. Αρχικά είτε να παραλληλοποίηση την εύρεση των ελάχιστων αποστάσεων μεταξύ των training elements και του κάθε query element, είτε να παραλληλοποιήσουμε ολόκληρη την διαδικασία μεταξύ των query elements. Καθώς δεν υπάρχει καμία συσχέτιση ή dependency μεταξύ των εκτελέσεων της διαδικασίας για κάθε query elements επιλέξαμε να παραλληλοποιήσουμε το μεγαλύτερο workload της ολικής διαδικασίας. Ωστόσο είναι σημαντικό να αναφερθεί πως σε περίπτωση που κάνουμε μία "εκπαίδευση" ενός μοντέλου συνήθως μετά ζητάμε ένα regression πάνω σε ένα μόνο query element. Υπό αυτή την υπόθεση η παραλληλοποίηση θα έπρεπε να πραγματοποιηθεί εσωτερικά στην εύρεση των ελάχιστων αποστάσεων και των γειτονικότερων training elements. Δοκιμάζοντας και τις δύο μεθόδους, η διαδικασία που ακολουθήθηκε απέδιδε περισσότερο "ελπιδοφόρα" αποτελέσματα. Σε περίπτωση ενός server με την σύνδεση πολλών μηχανημάτων οι δύο παραπάνω μεθοδολογίες θα συνδιάζονταν με nested παραλληλοποίηση με threads στην εύρεση των knns και με την χρήση του MPI θα μοιράζαμε τα query elements μεταξύ των nodes.
Όπως εξηγήσαμε και στον σειριακό κώδικα παρέχονται τόσο υλοποιήσεις με χρήση SIMD εντολών όσο και χωρίς για την εκτέλεση του κώδικα μας στο σύστημα που μας δίνεται πρόσβαση.

❖ MPI

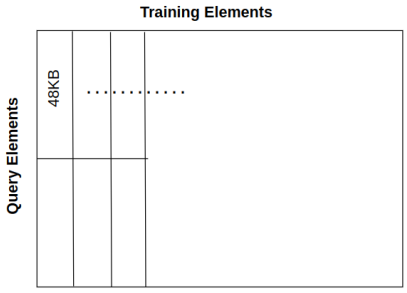
Συνυπολογίζοντας τις παρατηρήσεις που σημειώθηκαν για το OpenMP, χρησιμοποιήσαμε το MPI για να διαμοιράσουμε μεταξύ διεργασιών τον υπολογισμό στα query Elements. Ωστόσο σε αντίθεση με το OpenMP, δεν κρίνουμε την χρήση του mpi για την διαμοιρασμένη - παράλληλη εύρεση των αποστάσεων μεταξύ των training Elements και κάθε query Element ως καλή πρακτική. Σε κάθε βήμα θα απαιτούταν μεγάλος όγκος μεταφοράς δεδομένων μεταξύ των διεργασιών. Η καλύτερη αλγοριθμική επιλογή θα ήταν κάθε διεργασία να έβρισκε τους δικούς της ελάχιστους NNBS γείτονες και έπειτα από επικοινωνία να βρίσκαμε τους ολικούς ελάχιστους για κάθε query element. Ωστόσο η απαίτηση των μεταφορών και των συγχρονισμών, ακόμα και σε αυτήν την περίπτωση θα δημιουργεί σημαντικές επαναληπτικές καθυστερήσεις οι οποίες υποθέτουμε πως θα οδηγούν σε χειρότερα αποτελέσματα.

❖ Cuda

Για αναλύσουμε τις επιλογές που πραγματοποιήσαμε στην προσπάθεια μας να για παραλληλισμό σε κάρτες γραφικών οφείλουμε αρχικά να συνοψίσουμε τις διαφορές στο μοντέλο προγραμματισμού μας.
Αρχικά όπως μπορεί να παρατηρηθεί και από τις πληροφορίες στο *system specifications* οι κάρτες γραφικών έχουν πολύ χαμηλότερα ρολόγια λειτουργίας

συγκριτικά με τις CPU. Ωστόσο προσφέρουν την δυνατότητα μαζικού παραλληλισμού με ορισμένες προυποθέσεις και περιορισμούς. Τα χαμηλότερα ρολόγια μας εμποδίζουν από το να θεωρήσουμε πρακτικό τον υπολογισμό και διαμοιρασμό των query elements μεταξύ των cuda threads. Υπολογισμοί που απαιτούν memory dependencies, race conditions και κατά συνέπεια γρήγορη sequential εκτέλεση σηματοδοτούν περιοχές κώδικα που δεν μπορούν να παραλληλοποιηθούν σε κάρτες γραφικών. Επιπροσθέτως κώδικας με πολλά if statements, memory jumps και branching οδηγεί σε thread divergence μέσα σε κάθε warp και σε απόλυτες περιπτώσεις μπορεί να οδηγήσει και σε συνολικά σειριακή εκτέλεση κώδικα ακόμα και στην κάρτα γραφικών. Η ιεραρχία μνήμης διαφέρει σημαντικά με αυτή που έχουμε συνηθίσει στις CPU, δεν γίνεται caching μέσω cache lines μεταξύ global data, είναι πολύ εύκολο να κάνουμε under utilize το memory bus bandwidth ανά κύκλο ρολογιού απαιτώντας συνεχώς μικρό πλήθος δεδομένων και μειώνοντας σημαντικά την απόδοση.
Δεδομένα που επαναχρησιμοποιούνται σε υπολογισμούς πρέπει να φορντίσουμε να αποθηκεύονται σε shared memory, σε κάθε streaming multiprocessor, ενώ μπορούμε πιθανώς να εκμεταλλευτούμε read only constant memory ή textures αν παρουσιάζεται αλγοριθμικό locality στην ζήτηση δεδομένων.
Στην περίπτωση του κώδικα μας πρέπει να επισημαίνουμε πως παραλληλοποιούμε την εύρεση της απόστασης όλων των training elements σε κάθε query element. Το brute forcing των αποστάσεων αρχικά δεν απαιτεί κανένα data dependency μεταξύ των threads και έπειτα η σύγκριση των συνιστωσών των δύο elements δεν απαιτεί μεγάλη υπολογιστική ισχύ. Υπό το συγκεκριμένο πρίσμα αρχικά μπορούμε να χρησιμοποιήσουμε έναν κύκλο μηχανής στην κάρτα ώστε μερικά threads να φορτώσουν στην *shared μνήμη* το query element για fast access σε κάθε ύστερο υπολογισμό. Κάθε thread στην συνέχεια ακολουθεί μία κατά βάση *I/O bound* διαδικασία καθώς πρέπει να δαπανίσει κύκλους ώστε να φορτώνει μία συνιστώσα του training element και ύστερα να την συγκρίνει με την αντίστοιχη shared τιμή της συνιστώσας του *Query Element*. Αν και τα *training elements* είναι *read only*, δεν μπορούμε να τα αποθηκεύσουμε ως constant λόγω του μεγάλου μεγέθους τους, ούτε έχει νόημα να τα αποθηκεύουμε στην shared μνήμη καθώς κάθε memory cell γίνεται accessed μόνο μια φορά ανά query. Τα παραπάνω αποτελούν την συλλογιστική πορεία που ακολουθήθηκε και οδήγησε στην υλοποίηση που παρουσιάζουμε. Στοχεύουμε στον ταχύτερο υπολογισμό του regression σε κάθε query element ξεχωριστά. Δυστυχώς δεν προλάβαμε να ακολουθήσουμε και να αναπτύξουμε και μια συνδιαστική μεθοδολογία ωστόσο θα την περιγράψουμε στην συνέχεια μερικές ιδέες που θεωρούμε πως θα οδηγούσαν σε βελτιστοποίηση της παραλληλίας.
Αρχικά προσπαθήσαμε να εισάγουμε μία δεύτερη πηγή παραλληλισμού με βάση τα *query elements* από την CPU. Με τον συνδιασμό του *openmp*, και *cuda* streams προσπαθήσαμε να δημιουργήσουμε ένα διαφορετικό πυρήνα / stream σε κάθε *thread* και να μοιράσουμε τα *query elements* σε *threads*. Όπως πιθανότατα γίνεται αντιληπτό η προσπάθεια δεν οδήγησε πουθενά. Χρησιμοποιώντας το 100% της κάρτας γραφικών τα *streams* εκτελούνταν σειριακά στην κάρτα παράγοντας τους ίδιους ουσιαστικά χρόνους με την προηγούμενη υλοποίηση. Για τον λόγο αυτό δεν θεωρούμε πως ο κώδικας καν άξιζε να συμπεριληφθεί στην αναφορά μας. Μια ακόμα αποτυχημένη προσπάθεια πραγματοποιήθηκε με τον ίδιο συνδιασμό

τεχνολογιών προσπαθώντας ωστόσο να παραλληλοποιήσουμε σε CPU την εύρεση των *NNBS* μικρότερων αποστάσεων που υπολογίζονται μέσα από την GPU. Και πάλι οι χρόνοι που σημειώθηκαν ήταν παρόμοιοι των αρχικών καθώς ο χρόνος εύρεσης των *NBBS nearest neighbors* ήταν ελάχιστος συγκριτικά με την υπόλοιπη διαδικασία, άρα και η παραλληλοποίηση του δεν παρέδωσε κάποια ιδιαίτερη βελτιστοποίηση. Το overhead των *OPENMP system calls* ήταν πιθανώς αρκετό για να ζυγίσει το speedup που παίρναμε.
Τα παραπάνω αν και ανελπιδοφόρα, αναφέρονται ως βάση των optimizations που θεωρούμε πως θα έχουν κάποια συντριπτική χρονοβελτίωση στον κώδικα μας. Προσπαθώντας αρχικά να επιταχύνουμε την εύρεση των κοντινότερων γειτόνων σε κάθε query elements θα μπορούσαμε να δεσμεύσουμε ένα μητρώο μεγέθους *[query elements X training elements]* στο οποίο να αποθηκεύσουμε την αποστάσεις μεταξύ κάθε query και κάθε training element σε αντίστοιχες θέσεις. Έπειτα με την χρήση ενός δεύτερου kernel να χρησιμοποιήσουμε κάθε *cuda thread* ώστε να βρει τις ελάχιστες *NNBS* τιμές σε κάθε row του μητρώου που προαναφέραμε. Επιπροσθέτως πρέπει να αναφερθούμε στην περίπτωση του να εκτελούμε την εύρεση των αποστάσεων πολλών *query elements* και πολλών *training elements* ταυτόχρονα. Οπτικοποιώντας το πρόβλημα εκ νέου ως ένα μητρώο σχηματιζόμενο από τα training και τα query data μπορούμε να οδηγηθούμε σε μία προσπάθεια blocking ώστε να εκμεταλλευτούμε τα στοιχεία που επαναχρησιμοποιούνται συνεχώς. Παρατηρούμε πως το μεγαλύτερο κόστος ακόμα και στην GPU είναι οι μεταφορές δεδομένων. Πρέπει να εκμεταλλευτούμε την shared memory όσο το δυνατόν καλύτερα.



Θα δημιουργήσουμε ένα *grid* από *threadblocks* βασισμένα στο μέγιστο μέγεθος της Shared μνήμης. Σε *cuda compute capabilities 3.0* το μέγιστο μέγεθος είναι τα *48KB*, ενώ ο μέγιστος αριθμός από threads 1024. Επιπροσθέτως μπορούμε να σημειώσουμε πως μπορούμε να εκμεταλλευτούμε πιθανώς το μικρό μέγεθος των query elements ώστε να φορτωθούν σε constant memory (μάλλον μεγέθους *64KB*) για *faster read only access* πρωτού τα αποθηκεύσουμε στην *shared* ή εάν η απόδοση είναι αρκετά μεγάλη θα μπορούσαμε να γλιτώσουμε την μεταφορά τους εξολοκλήρου. Ωστόσο σύμφωνα με την έρευνα μας πιθανότατα θα χρειαστεί, πέρα από το γεγονός πως είναι ως μέθοδος είναι problem specific. Με βάση τα παραπάνω μπορούμε να χρησιμοποιήσουμε πολλαπλά *threads* για να μεταφέρουμε *cumulative μνήμη* ταυτόχρονα στην *shared memory* μέχρι να την γεμίσουμε. Στο σημείο αυτό βοηθάει ιδιαίτερα η χρήση των *floats* όχι μόνο στις ταχύτητες υπολογισμών αλλά και στο μέγεθος των μητρώων. Ανάλογα με τις διαστάσεις των δεδομένων θα μπορούσαμε να χρησιμοποιήσουμε έως και 1024 *threads* ώστε να μεταφέρουμε και να επεξεργαστούμε δεδομένα αλλά καθώς κάθε *block* γίνεται dedicated σε ένα *Streaming multiprocessor* μπορεί να διευκολύνει η χρήση λιγότερων threads ανά *block* που να εκτελούν περισσότερους κύκλους μεταφορών. Οφείλουμε να παραδεχτούμε πως δεν μπορούμε να προβλέψουμε το trade off και οι επιλογές των διαστάσεων των blocks πρέπει να γίνουν πειραματικά με μία δομή αντίστοιχη με του παραδείγματος του σχεδίου μας. Ως εκτούτου κάθε *block* θα υπολογίζει την μερική απόσταση μεταξύ των

training elements και μερικών *query elements* αποθηκεύοντας τα αποτελέσματα σε μία κατάλληλη δομή του προαναφερθέντος μητρώου αποστάσεων σε αντίστοιχες θέσεις. Επαναχρησιμοποιούμε πολλές φορές για κάθε *query element* που έχουμε φορτώσει σε κάθε block τα αντίστοιχα re accessed *training elements*. Τέλος μπορούμε να δοκιμάσουμε να αποθηκεύσουμε εκ νέου τα *training elements* ως *read only textures 1D ή 2D*, προσπαθώντας να εκμεταλλευτούμε το *read only nature* και *cache-ability* του προβλήματος και το *locality* των δεδομένων. Πειραματιστήκαμε με την χρήση *texture μνήμης* χωρίς ιδιαίτερη χρονοβελτίωση στον κώδικα που παραδώσουμε καθώς δεν εκμεταλλεύεται ιδιαίτερα καλά το locality και υπάρχουν πολλές deprecated αναφορές πάνω σε συγκεκριμένες μεθοδολίες. Αν υπάρχει χρονοβελτίωση με την συγκεκριμένη τεχνική υποθέτουμε πως θα είναι με την χρήση της *L2 cache* της GPU σε περίπτωση που έχουν επαναφορτωθεί σε διαφορετικό *Streaming multiprocessor* συγκεκριμένα δεδομένα.
Δυστυχώς δεν πρόλαβα να τελειώσω την δεύτερη διαδικασία που αναφέρουμε και παραδώσαμε ολοκληρωμένα την απλουστευμένη και πρώτη ιδέα που είχαμε, αλλά αν είναι δυνατό θα ήθελα να ζητήσω την διατήρηση της πρόσβασης μου στο μηχάνημα με την κάρτα γραφικών για δοκιμές και περαιτέρω πειραματισμό.

❖ OpenACC

Έχοντας αναλύσει επαρκώς θεωρούμε στην αναφορά μας για την λογική προγραμματισμού στην Cuda, θέλαμε να αναφέραμε με ο accelerator του *openacc* μας "κρύβει" πολλές από τις εσωτερικές του διαδικασίες. Χρησιμοποιήθηκε ο nvc++ compiler καθώς δεν υπήρχε access σε pgi και pgc++. Με στόχο να επιτύχουμε την shared memory που αναφέραμε παραπάνω στα query elements χρησιμοποιήσαμε είτε *firstprivate* clause είτε το *#pragma cache()* directive. Το σοβαρότερο λάθος κατά τον προγραμματισμό σε OpenACC είναι το data management. Για να σιγουρευτούμε τον ελάχιστο αριθμό μεταφορών χρησιμοποιήσαμε build in συναρτήσεις που παρομοιάζουν αυτές της Cuda για data allocation και transfer. Ύστερα από το *#pragma acc parallel* προσομοιάζεται η διαδικασία του *#pragma omp parallel*. Δημιουργείται μια παράλληλη περιοχή την οποία "εκτελούν" redundantly όλα τα *gangs*, ενώ με την χρήση της *#pragma acc loop* γίνεται διαμοιρασμός του workload σε αυτά. Καθορίζουμε στατικά το μέγεθος του *threadblock* προσομοιάζοντας τον κώδικα που πραγματοποιήσαμε με την χρήση του CUDA.

❖ Παρουσίαση αποτελεσμάτων :

Τα χρονικά στιγμιότυπα εκτελέσεων αναδεικνύουν τους τις χρόνους για διαφορετικό αριθμό από threads ή processes μετρημένα σε milliseconds.

OMP	*TRA = 1048576*		*QUE=1024*				*SIMD*	
Ryzen 5 5600x	*Serial*	*Serial SIMD*	*OMP*	*OMP SIMD*	*Speedup*	*Efficiency*	*Speedup*	*Efficiency*
1	6610.51	3988.28	6744.94	4070.16	0.980	0.980	0.9798	0.9798
2			3429.68	2565.32	1.927	0.963	1.5546	0.7773
4			1897.37	1288.27	3.484	0.871	3.0958	0.7739
6			1228.04	995.85	5.382	0.897	4.0048	0.6674
8			1252.27	714.42	5.278	0.659	5.5824	0.6978

MPI	*TRA = 1048576*		*QUE=1024*				*SIMD*	
Ryzen 5 5600x	*Serial*	*Serial SIMD*	*MPI*	*MPI SIMD*	*Speedup*	*Efficiency*	*Speedup*	*Efficiency*
1	6513.95	3781.52	6456.058	3826.68	1.0089	1.008	0.9881	0.9881
2			3374.35	2848.93	1.930	0.9652	1.3273	0.6636
4			2974.93	2992.97	2.1895	0.5473	1.2634	0.3158
6			3186.81	3165.42	2.0440	0.3406	1.1946	0.1991
8			3210.28	3135.73	2.0290	0.2536	1.2059	0.1507

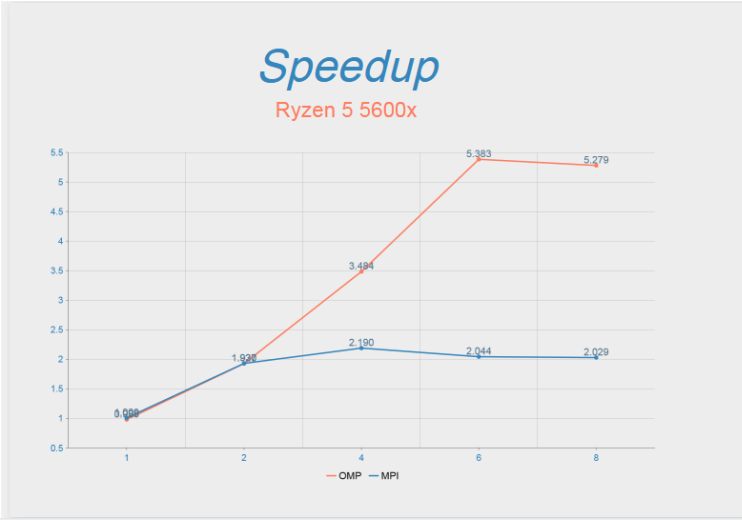
Θα θέλαμε να αναφέρουμε εκ νέου πως παραθέτουμε τα αποτέλεσμα στον προσωπικό μας επεξεργαστή καθώς δεν καταφέραμε να εκτελέσουμε κώδικα βασισμένο σε avx intrinsics στο μηχάνημα που μας δίνεται πρόσβαση.

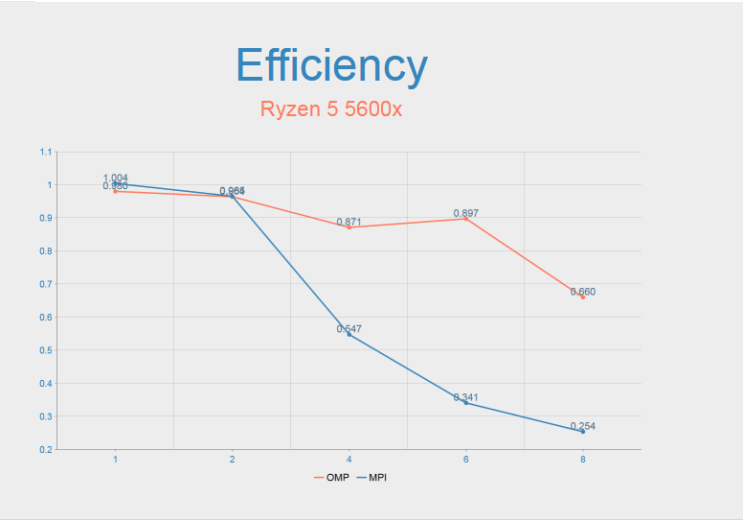
OMP	*TRA = 1048576*	*QUE=1024*		
Intel i7 3770K	*Serial*	*OMP*	*Speedup*	*Efficiency*
1	10450.27	10495.67	0.9956	0.9956
2		7259.896	1.4394	0.7197
4		5706.136	1.8314	0.4578
6		3863.809	2.7046	0.4507
8		2750.771	3.7990	0.4748

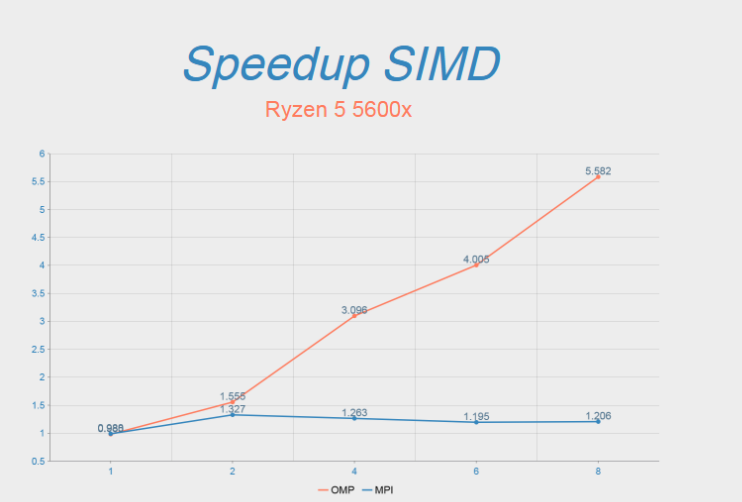
MPI	*TRA = 1048576*	*QUE=1024*		
Intel i7 3770K	*Serial*	*MPI*	*Speedup*	*Efficiency*
1	10554.12	10869.19	0.9710	0.9710
2		7351.85	1.4355	0.7177
4		7160.40	1.4739	0.3684
6		5419.13	1.9475	0.3245
8		7324.24	1.4409	0.1801

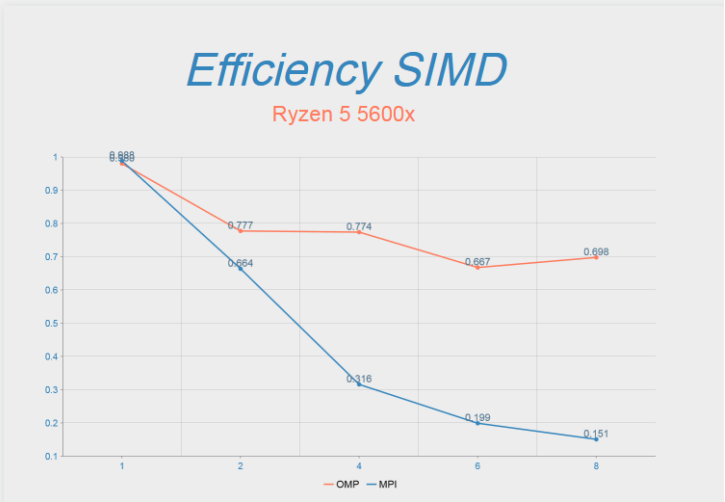
	Serial	Cuda	*ACC*
Total Time (ms)	10554.123	3983.453	1950.025
Average time Per Query	10.306762	3.890091	1.904321
Speedup ratio		2.649492	5.412301

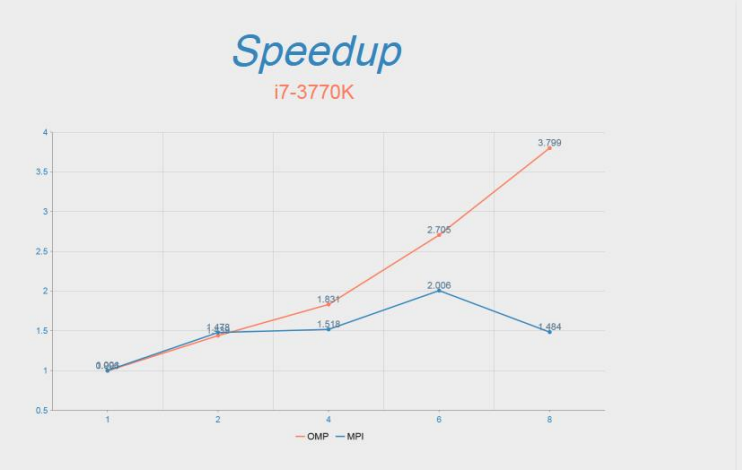
❖ Οπτικοποίηση αποτελεσμάτων :

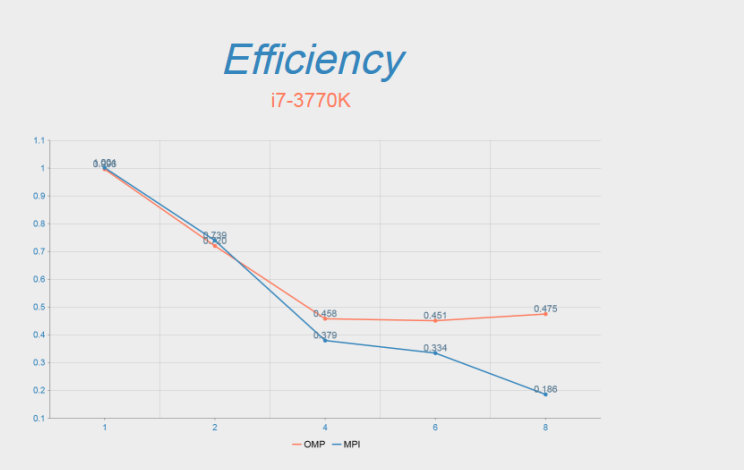




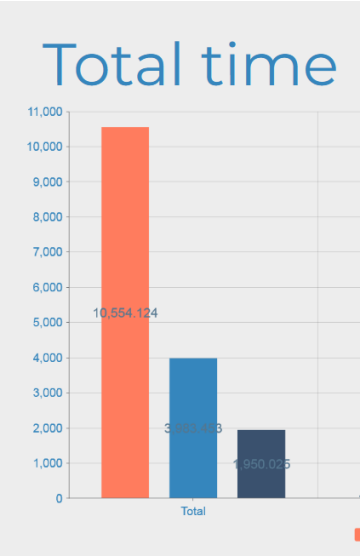








❖ Σχολιασμός αποτελεσμάτων :



Σύγκριση μεταξύ σειριακής εκτέλεσης, cuda, openacc

Αρχικά θέλουμε να αναφερθούμε στις διαφορές που παρατηρούνται μεταξύ των υλοποιήσεων με την χρήση του OpenMP και MPI. Τα αποτελέσματα που πάρθηκαν μας δημιούργησαν τρομερή εντύπωση που οδήγησε σε αναζήτηση της καθυστέρησης που παρατηρείται στον κώδικα του MPI. Έχοντας γράψει τον κώδικα οι διαδικασίες που ακολουθήθηκαν και η μεθοδολογία ήταν ταυτόσημες, ο διαμοιρασμός μεταξύ των query elements είναι ο ίδιος και στις δύο διαδικασίες ενώ δεν χρειάζεται σχεδόν καμία επικοινωνία μεταξύ των διεργασιών του MPI. Μάλιστα το πόσο πολύ τα αποτελέσματα ταυτίζονται για δύο διεργασίες ή δύο processes δημιουργεί ακόμα μεγαλύτερη περιέργεια. Αναλύσαμε εξονυχιστικά τον κώδικα και τα αποτελέσματα δεν μας έδωσαν καμία σαφή απάντηση. Θα παρατηρήσετε πως ο κώδικας που παραδώθηκε για το MPI αν γίνει compile με την χρήση του -DDEBUG παράγει διαγνωστικά μηνύματα που κρατήσαμε υποδειγματικά. Το μόνο που περιέχουν οι συγκεκριμένες μεταβολές είναι μετρητές χρόνου και δεν επηρεάζουν καθόλου την διαδικασία εκτέλεσης του προγράμματος. Η επιπλέον ασάφεια δημιουργείται καθώς με την εκτέλεση του κώδικα με την χρήση του προαναφερθέντος flag, ο κώδικας παρομοιάζει ταυτόσημο σχεδόν speedup με τις εκτελέσεις του OpenMP, συγκρίνοντας το με τον σειριακό κώδικα που παράγεται με εκτέλεση του ίδιου mpi κώδικα για μια διεργασία. Στο προσωπικό μας ωστόσο μηχάνημα οι χρόνοι φαίνονται να είναι 6 φορές πιο αργοί.

```
for (int i=0; i<TRAINELEMS; i++){
    if( computed_distance[i] < max_d){
        nn_x[max_i] = i;
        nn_d[max_i] = computed_distance[i];
    }
    max_d = compute_max_pos( nn_d, knn, &max_i);
}
```

Επιπροσθέτως αν αντί για την χρήση του -DDEBUG flag αντιστρέψουμε την διαδικασία που είχαμε κάνει για optimization στον σειριακό κώδικα και πάλι το scalability του κώδικα με processes φαίνεται να προσεγγίζει εξαιρετικά τα γραφήματα του OpenMP, με πολύ πιο αργές εκτελέσεις φυσικά. (Για 4 processes το MPI φαίνεται να παράγει speedup 3.98) Είναι σημαντικό να σημειωθεί πως σε όλες τις εκτελέσεις που αναφέρουμε τα διαγνωστικά μηνύματα για το APE and MSE ταυτίζονται, οι κώδικες εκτελούνται ορθά. Επιλέξαμε να δείξουμε τις πιο γρήγορες εκτελέσεις του mpi ακόμα και αν αυτές δεν έχουν καλό scalability. Δεν μπορώ να το εξηγήσω με κάποια βάση πέρα από την διαίσθηση. Όσο γρηγορότερη φαίνεται να είναι η εκτέλεση, τόσο χειρότερα φαίνεται το MPI να κάνει scale, βασισμένοι στην διαφοροποίηση που παρατηρείται όταν χρησιμοποιούμε και simd instructions σε συνδιασμό με το MPI. Αυξάνοντας ωστόσο το μέγεθος του πρβλήματος σε 8192 query elements δεν φάνηκε να αλλάζει την συμπεριφορά των εκτελέσεων άρα υποθέτουμε πως δεν μετράμε λανθασμένα κάποιο overhead από την βιβλιοθήκη καθώς οι χρόνοι γίνονται γρηγορότεροι των 2 ms.
Η πτώση στις γραφικές τόσο του MPI όσο και του OpenMP είναι φυσικά αναμενόμενες καθώς πλησιάζουμε τον μέγιστο αριθμό των cores.

Αναλύοντας και τα αποτελέσματα που καταγράφουμε με τις παραλληλοποιήσεις στην GPU, ακόμα και η “naive” προσέγγιση που πραγματοποιήθηκε φαίνεται να παρουσιάζει σημαντικά αποτλέσματα στην χρονοβελτίωση σε αρχική εικόνα, ωστόσο ταυτίζονται με την εκτέλεση του openmp για 6 threads. Παρά την μεγάλη προσπάθεια μας πάνω στο πρόγραμμα με την χρήση CUDA, δεν καταφέραμε να ξεπεράσουμε τους χρόνους που παράγει με πολύ μικρή προσπάθεια και λίγη προσοχή το OpenACC. Η τόσο μεγάλη απόκλιση στο speedup αποδεικνύει πως, ακόμα και η συγκεκριμένη μεθοδολογία που ακολουθήσαμε για τον παραλληλισμό των αποστάσεων απέχει υπερβολικά από το να πετύχει μέγιστο optimality. Συγκριτικά με τον αρχικό κώδικα που μας παραδώθηκε θεωρούμε πως έχει επιτευχθεί μία σημαντική βελτίωση κρίνοντας αποκλειστικά τους χρόνους εκτέλεσης, ωστόσο υπάρχουν πολύ μεγάλα περιθώρια βελτίωσης και ίσως ακόμα και πιθανές αστοχίες που θα μπορούσαν να οδηγήσουν σε τεράστια βελτίωση των αποτελεσμάτων για τις εκτελέσεις του MPI και CUDA.

Ευχαριστούμε πολύ για την κατανόηση και την υπομονή σας μέσα στο ακαδημαικό εξάμηνο.
Με εκτίμηση,
Η ομάδα.