ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη απόδοσης τελεστών βάσεων δεδομένων σε ετερογενείς αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης Α. Λαμπράκης

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π

Αθήνα, Οκτώβριος 2019

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη απόδοσης τελεστών βάσεων δεδομένων σε ετερογενείς αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης Α. Λαμπράκης

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 16η Οκτωβρίου 2019.

.....	.....	.....
Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π	Γεώργιος Γκούμας Επ. Καθηγητής Ε.Μ.Π	Δημήτριος Τσουμάκος Αν. Καθηγητής Ιονίου Πανεπιστημίου

Αθήνα, Οκτώβριος 2019

.....
Παναγιώτης Α. Λαμπράκης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Σκοπός της παρούσας διπλωματικής εργασίας είναι η μελέτη ετερογενών αρχιτεκτονικών, εστιάζοντας στο πεδίο εφαρμογής των βάσεων δεδομένων. Οι αρχιτεκτονικές τις οποίες πραγματεύεται η παρούσα διπλωματική είναι αυτή της απλής κεντρικής μονάδας επεξεργασίας(CPU), της μονάδας επεξεργασίας με δυνατότητα χρήσης εντολών «μονής εντολής-πολλαπλών δεδομένων»(SIMD) καθώς επίσης και της αρχιτεκτονικής «μονής εντολής - πολλαπλών νημάτων»(SIMT), η οποία αναφέρεται στην μονάδα επεξεργασίας γραφικών(GPU). Τα ερωτήματα προς τη βάση δεδομένων που θα εξετάσουμε είναι αυτά της εύρεσης ενός στοιχείου (Scan), της απαρίθμησης (Count), του αθροίσματος (Sum), του μικρότερου στοιχείου (Min) και του μεγαλύτερου στοιχείου (Max).

Λέξεις κλειδιά: SIMD, SIMT, CUDA, νήματα, βάσεις δεδομένων, συναθροιστικοί τελεστές, εύρεση στοιχείου, τελεστές βάσεων δεδομένων, ετερογενής αρχιτεκτονική

Abstract

The goal of this Diploma Thesis is to study different heterogeneous architectures focusing their appliances in the field of databases. CPU, CPU with SIMD (Single Instruction – Multiple Data) instructions enabled and SIMT (Single Instruction – Multiple Threads) architecture referring to GPU are the three architectures we studied for this purpose. Studied SQL queries are filtering an element among others (Scan), counting the number of entries (Count), calculating the sum of the elements (Sum) and finding the min (Min) and the max (Max) value among the elements.

Key words: SIMD, SIMT, CUDA, threads, databases, aggregation functions, filtering, heterogeneous computing

Ευχαριστίες

Με την ευκαιρία ολοκλήρωσης της διπλωματικής μου εργασίας, θα ήθελα να ευχαριστήσω τον κ. Ιωάννη Κωνσταντίνου για την βοήθεια που μου παρείχε όποτε την χρειάστηκα.

Ακόμα πιο πολύ,θα ήθελα να ευχαριστήσω την οικογένειά μου, η οποία είναι πάντα στο πλευρό μου, σε κάθε μου βήμα, καθώς επίσης και όλα εκείνα τα άτομα που μου συμπαραστάθηκαν καθ' όλη τη διάρκεια αυτού του ταξιδιού.

Αφιερώνεται,
Στην οικογένειά μου και σε
αυτούς που με στηρίζουν...

Περιεχόμενα

Κεφάλαιο 1 – Εισαγωγή

1.1 Ανομοιογένεια (Heterogeneity)

Η ανομοιογενής υπολογιστική (heterogeneity computing) αναφέρεται σε συστήματα τα οποία χρησιμοποιούν περισσότερους του ενός είδους επεξεργαστές ή πυρήνες. [1] Ο κύριος επεξεργαστής και οι υπόλοιποι συνεπεξεργαστές έχουνε διαφορετική αρχιτεκτονική συνόλου εντολών (Instruction Set Architecture). Συνήθως οι τύποι επεξεργαστών που χρησιμοιποιούνται στις ετερογενείς αρχιτεκτονικές, κατά πλειοψηφία, είναι η κεντρική και η γραφική μονάδα επεξεργασίας, CPU (Central Processing Unit) και GPU (Graphics processing Unit) αντίστοιχα. Η CPU βελτιστοποιεί την μετρική του latency ενώ η GPU βελτιστοποιεί την μετρική του throughput.

Ο βαθμός ανομοιογένειας παρουσιάζει μία σταδιακή αύξηση στα σύγχρονα υπολογιστικά συστήματα, όσο η περαιτέρω κλιμάκωση των τεχνολογιών κατασκευής επιτρέπει στα προηγουμένως διακριτά στοιχεία του συστήματος να ενσωματώνονται σε ψηφίδα (system-on-chip).

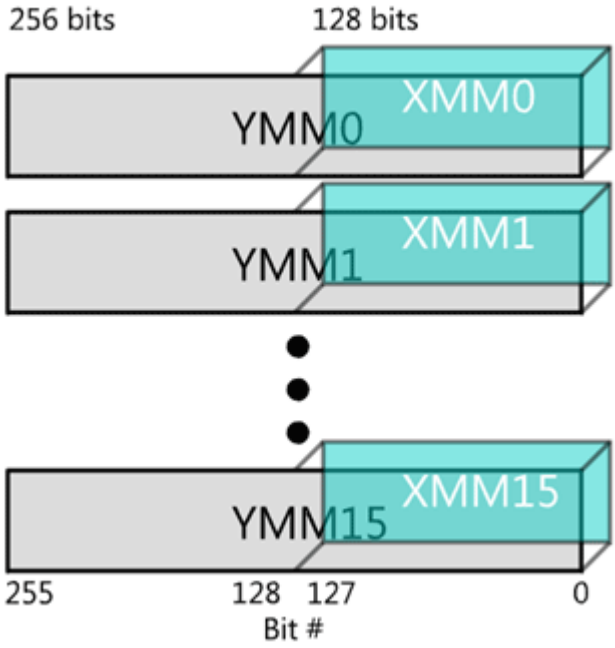
Ο μεγάλος όγκος δεδομένων προς επεξεργασία, η ανάγκη για χαμηλή κατανάλωση ενέργειας κατά την επεξεργασία, καθώς επίσης και η ανάγκη για όσο το δυνατόν μικρότερο χρόνο επεξεργασίας είναι μερικοί λόγοι που οδήγησαν την ανθρωπότητα στην υλοποίηση τέτοιων συστημάτων. Ενδεικτικά πεδία εφαρμογής τέτοιων αρχιτεκτονικών είναι η μηχανική μάθηση (machine learning), η βαθιά μηχανική μάθηση (deep learning), η ρομποτική (robotics), οι βάσεις δεδομένων (databases) και πολλά άλλα.

Το πεδίο με το οποίο θα ασχοληθούμε εμείς είναι αυτό των βάσεων δεδομένων και οι αρχιτεκτονικές τις οποίες θα εξετάσουμε είναι αυτές της CPU και της GPU. Θα μελετήσουμε κάποιες κατηγορίες ερωτημάτων προς τη βάση δεδομένων και θα αξιολογήσουμε τις προς μελέτη αρχιτεκτονικές βάσει απόδοσης.

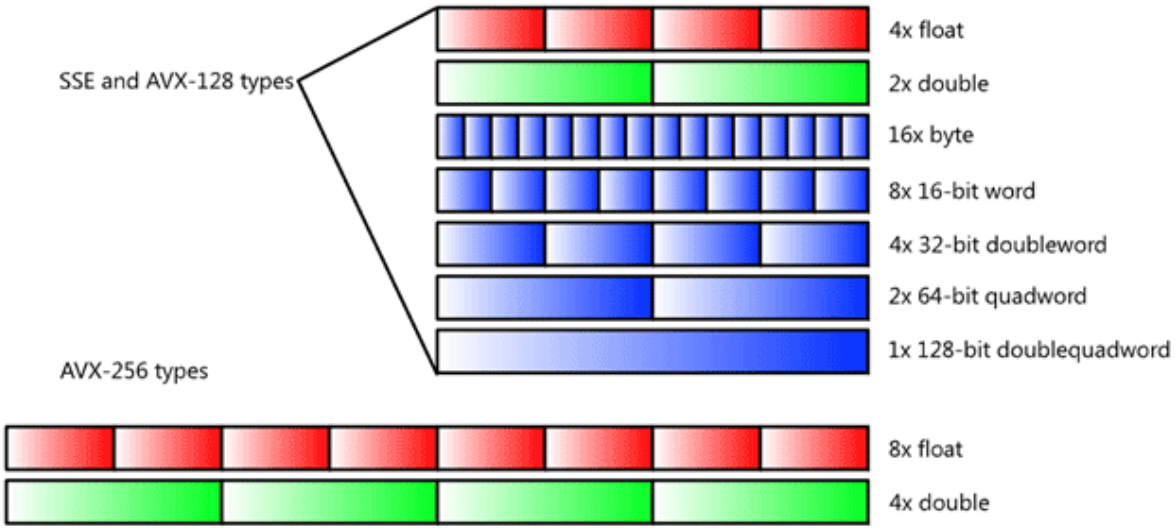
1.2 Αρχιτεκτονική SIMD – Intel AVX

Η αρχιτεκτονική «μονής εντολής-πολλαπλών δεδομένων» (SIMD) μας επιτρέπει να επεξεργαζόμαστε πολλαπλά δεδομένα σε ένα βήμα καταφέρνοντας με αυτόν τον τρόπο να αυξήσουμε το throughput σε πολλές εφαρμογές. Οι εντολές τύπου AVX2 είναι απόγονος των εντολών τύπου AVX. Οι εντολές AVX είναι επεκτάσεις της αρχιτεκτονικής συνόλου εντολών για μικροεπεξεργαστές της AMD και της Intel, οι οποίες προτάθηκαν από την Intel τον Μάρτιο του 2008 και εφαρμόστηκαν στους επεξεργαστές της, γενιάς Sandy Bridge, το πρώτο τέταρτο του έτους 2011 ενώ στο τρίτο τέταρτο του ίδιου έτους την εφαρμογή τους ακολούθησε και η AMD, στους επεξεργαστές γενιάς Bulldozer [2].

Το πρότυπο AVX2 εισήγαγε την επέκταση των εντολών για ακέραιους αριθμούς (κατά κύριο λόγο) καθώς επίσης και λειτουργίες τύπου FMA (fused multiply-accumulate). Το hardware το οποίο υποστηρίζει εντολές AVX2 αποτελείται από δεκαέξι καταχωρητές μεγέθους 256-bit έκαστος, τους YMM0-YMM15, και από τον καταχωρητή ελέγχου /κατάστασης, MXCSR. Οι καταχωρητές YMM0-YMM15 χτίστηκαν πάνω από τους προϋπάρχοντες καταχωρητές XMM0-XMM15 του προτύπου AVX, οι οποίοι αποτελούν το μισό, σε μέγεθος, των πρώτων σύμφωνα με την εικόνα 1.



Εικόνα 1: Καταχωρητές προτύπου AVX2.

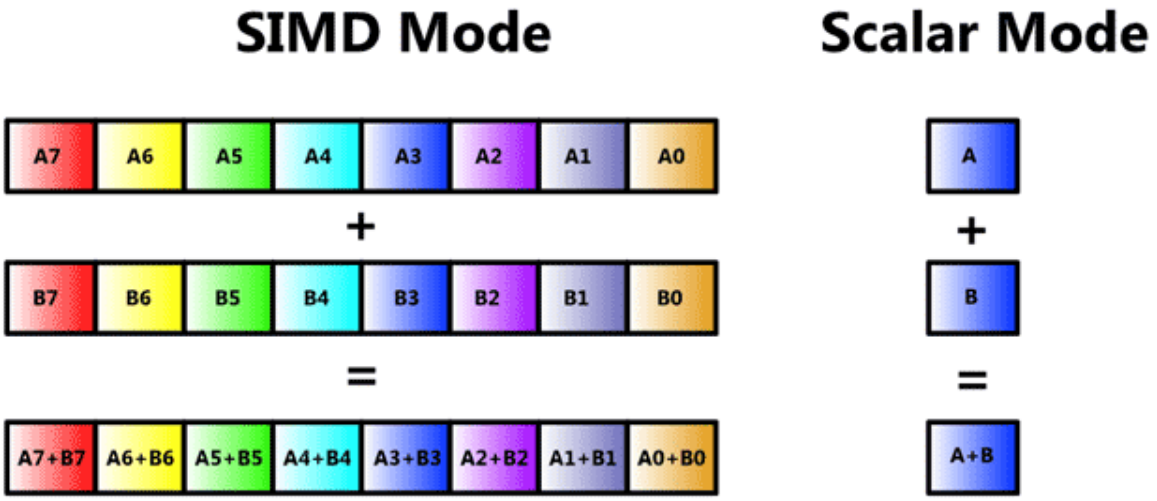


Εικόνα 2: Σύγκριση καταχωρητών προτύπων AVX και AVX2.

Κάθε ένας από τους καταχωρητές YMM μπορεί να περιέχει, για παράδεγμα, οκτώ αριθμούς κινητής υποδιαστολής μονής ακρίβειας (single-precision floating point numbers) μεγέθους 32-bit, τέσσερις αριθμούς κινητής υποδιαστολής διπλής ακρίβειας (double-precision floating point numbers) μεγέθους 64-bit κ.λπ.

Η εικόνα δύο συμπληρώνει τους προηγούμενους ισχυρισμούς κάνοντας μία εκτενέστερη σύγκριση μεταξύ της χωρητικότητας των καταχωρητών για το πρότυπο AVX και AVX2 ανάλογα και με τον τύπο δεδομένων που μπορεί να περιέχουν σε κάθε περίπτωση.

Η υπεροχή των εντολών τύπου SIMD έναντι των κλιμακωτών(scalar) παρουσιάζεται στην εικόνα τρία. Οι scalar εντολές χρησιμοποιούν κάθε φορά μία εγγραφή κάθε καταχωρητή σε αντίθεση με τις SIMD που αυξάνουν την παραλληλία εκτελώντας ταυτόχρονα την ίδια εντολή σε πολλά δεδομένα. Αυτή η διαφορά επιτρέπει και λιγότερες μετακινήσεις δεδομένων για κάποιους αλγορίθμους προσφέροντας έτσι βελτιωμένο throughput.



Εικόνα 3: Σύγκριση ανάμεσα σε scalar εντολές και εντολές τύπου SIMD.

Για τις εντολές του προτύπου Intel SSE, απαιτούνταν η μνήμη να είναι ευθυγραμμισμένη (aligned memory). Το πρότυπο Intel AVX όμως είναι πιο ελαστικό ως προς την ευθυγράμμιση της μνήμης, μία ελαστικότητα όμως που μπορεί να μειώσει την απόδοση εφόσον η μνήμη είναι μη ευθυγραμμισμένη. Λέμε ότι τα δεδομένα είναι ευθυγραμμισμένα στην μνήμη, όταν τα δεδομένα τα οποία θα υποστούν επεξεργασία ως chunk μεγέθους n, είναι αποθηκευμένα σε όριο μνήμης n-byte. Για παράδειγμα, όταν φορτώνουμε 256 bit δεδομένα στον YMM καταχωρητή, αν το data source είναι ευθυγραμμισμένο κατα 256 bit, τότε τα δεδομένα λέγονται ευθυγραμμισμένα [3].

1.3 Αρχιτεκτονική SIMT – GPU

Η αρχιτεκτονική «μονής εντολής – πολλαπλών νημάτων» (SIMT) προτάθηκε για πρώτη φορά από την εταιρία NVIDIA το Νοέμβριο του 2006 και εισήχθει στις κάρτες γραφικών της από τη γενιά Tesla και μετέπειτα. Πρόκειται για ένα μοντέλο εκτέλεσης σύμφωνα με το οποίο πολλαπλά και ανεξάρτητα μεταξύ τους νήματα (threads) εκτελούν ταυτόχρονα την ίδια εντολή [4].

1.3.1 Αρχιτεκτονική GPU

Μία κάρτα γραφικών αποτελείται από δύο κύρια στοιχεία, την καθολική μνήμη (global memory) και τους πολυ-επεξεργαστές (streaming multiprocessors) [5]. Η global memory είναι το ανάλογο της RAM για τον επεξεργαστή και είναι προσβάσιμη τόσο από την CPU όσο και από την GPU. Οι streaming multiprocessors εκτελούν τους υπολογισμούς και καθένας έχει δικές του μονάδες ελέγχου (control units), καταχωρητές (registers), σωληνώσεις εκτέλεσης (execution pipelines), μνήμες τύπου cache και αποτελείται από πολλούς πυρήνες (cores).

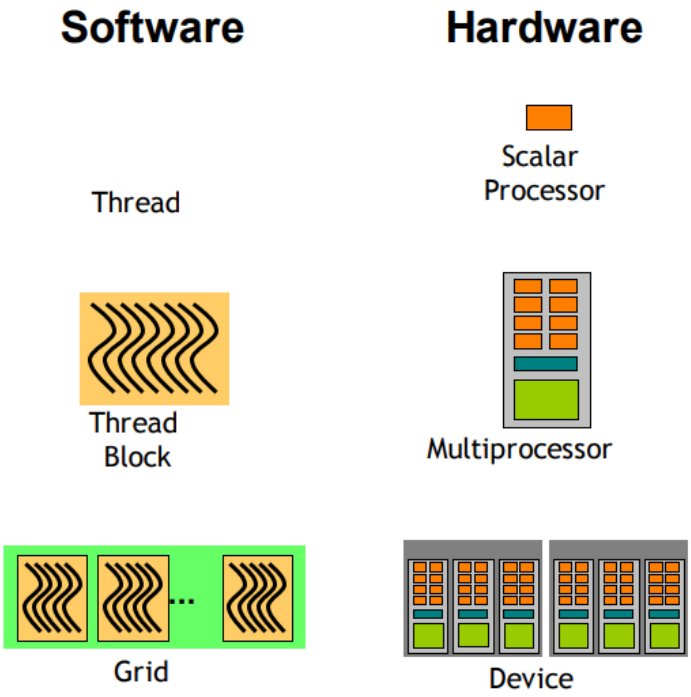
Εικόνα 4: Αναπαράσταση μίας GPU.

Εικόνα 5: Τμήμα ενός streaming multiprocessor.

Εικόνα 6: Εσωτερική αναπαράσταση ενός multiprocessor core.

Κάθε core του multiprocessor αποτελείται από μονάδες ακεραίων (integer units), αριθμών κινητής υποδιαστολής (floating point units), διακλάδωσης (branch units), σύγκρισης και μεταφοράς (move, compare units) και λογικές μονάδες (logic units).

1.3.2 Μοντέλο εκτέλεσης

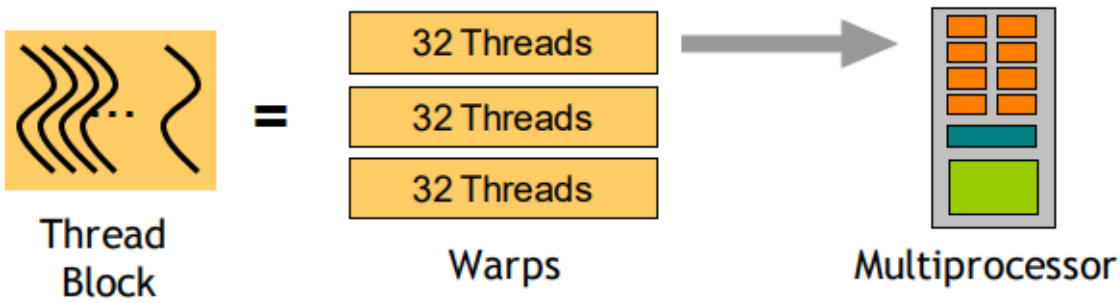


Τα threads εκτελούνται από τους scalar processors.

Τα thread blocks εκτελούνται από τους multiprocessors και δεν μετακινούνται σε άλλους multiprocessors.
Πολλά ταυτόχρονα thread blocks μπορούνε να υπάρχουν σε έναν multiprocessor, περιοριζόμενα από τους πόρους κάθε multiprocessor (shared memory και register file).

Ένας kernel εκτελείται από ένα πλέγμα (grid) thread block.

Κάθε thread block αποτελείται από μικρότερα κομμάτα, το καθένα μεγέθους 32 thread, που ονομάζονται warps. Έτσι, ένα thread block που αποτελείται από 1024 threads συνολικά, εμπεριέχει 32 warps των 32 thread έκαστο. Ένα warp εκτελείται παράλληλα (SIMT) σε κάθε multiprocessor.

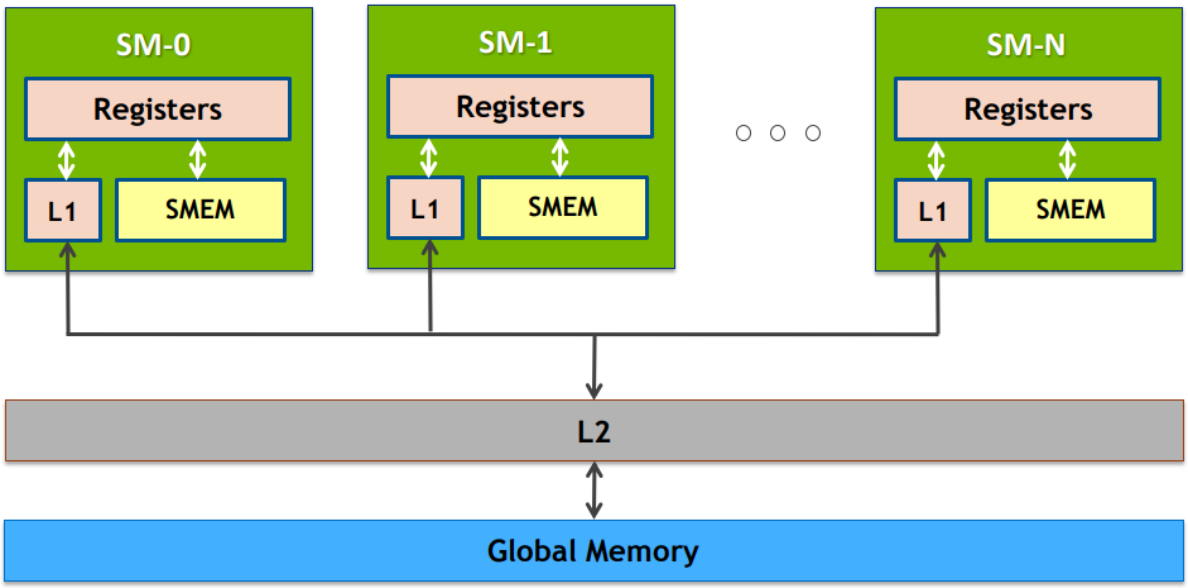


Εικόνα 7: Ανάλυση ενός thread block.

1.3.3 Ιεραρχία μνήμης της κάρτας γραφικών

Μία κάρτα γραφικών αποτελείται από την καθολική μνήμη και την μνήμη cache επιπέδου 2 (Level 2 cache). Μέσα σε κάθε streaming multiprocessor υπάρχει μία μνήμη cache επιπέδου 1 (Level 1 cache), οι καταχωρητές και η κοινή μνήμη (shared memory). Η διαφορά της L1 cache και της SMEM είναι ότι η πρώτη διαχειρίζεται αποκλειστικά από το hardware, ενώ η δεύτερη διαχειρίζεται από τον προγραμματιστή, ο οποίος και είναι

υπεύθυνος για τον συγχρονισμό των προσβάσεων στη μνήμη από τα threads που τρέχουνε παράλληλα.



Εικόνα 8: Ιεραρχία μνήμης της κάρτας γραφικών.

1.3.4 Πιθανοί λόγοι ελάττωσης της απόδοσης

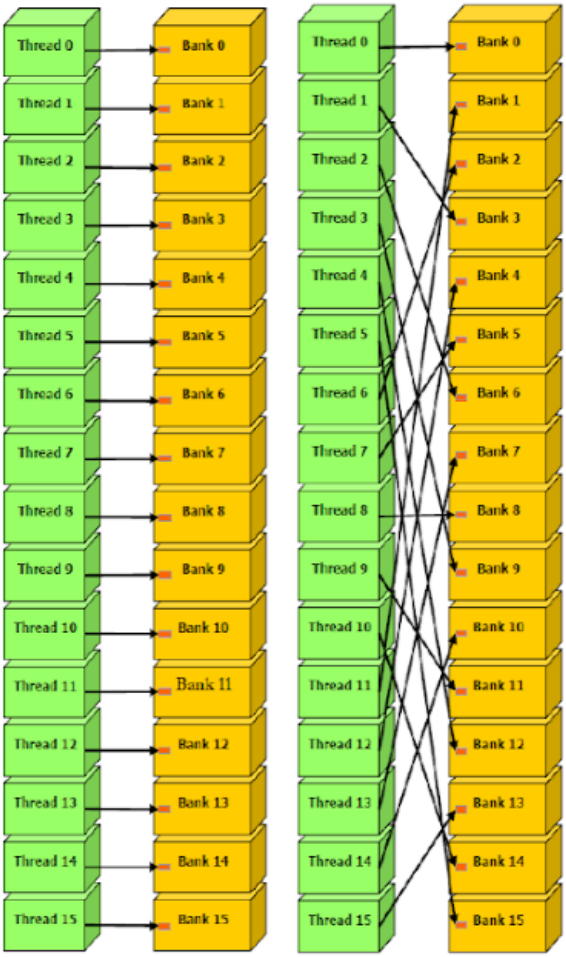
Κατά τον σχεδιασμό προγραμμάτων που θα τρέξουνε σε κάρτα γραφικών, θα πρέπει να δοθεί η απαραίτητη προσοχή ώστε να αποφευχθούν φαινόμενα bank conflict καθώς επίσης και η πρόσβαση στην μνήμη να είναι «συνενωτική» (coalesced memory access).

Η shared memory είναι χωρισμένη σε πολλαπλά banks [6]. Για κάρτες γραφικών με compute capability 2.0 και μετέπειτα, ο αριθμός των banks ισούται με 32. Κάθε bank έχει μέγεθος 4 byte και μπορεί να εξυπηρετήσει ένα αίτημα κάθε φορά. Μία οπτικοποίηση για το πως μοιάζουνε τα banks στην μνήμη φαίνεται στην εικόνα 9.

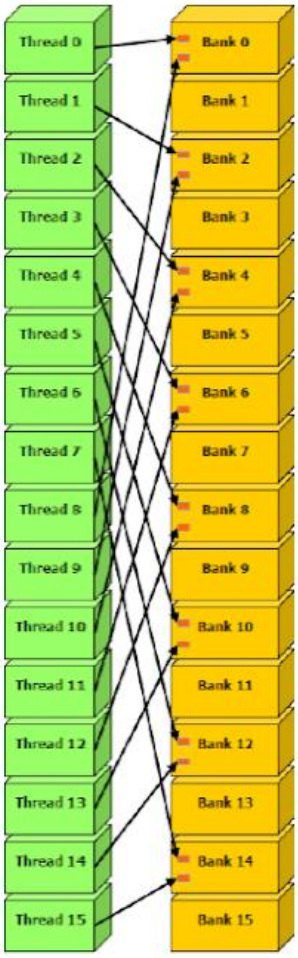
Bank	1	2	3	4	...	32
Address	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	...	28 29 30 31
Address	32 33 34 35	36 37 38 39	40 41 42 43	44 45 46 47	...	60 61 62 63
....	...	...	...	...	...	...

Εικόνα 9: Οπτικοποίηση των banks.

Εάν το μισό warp, δηλαδή 16 threads, προσπελαύνει συνεχόμενες τιμές 32 bit(4 byte) τότε δεν έχουμε bank conflict [7].



Εικόνα 10: Bank conflict free access

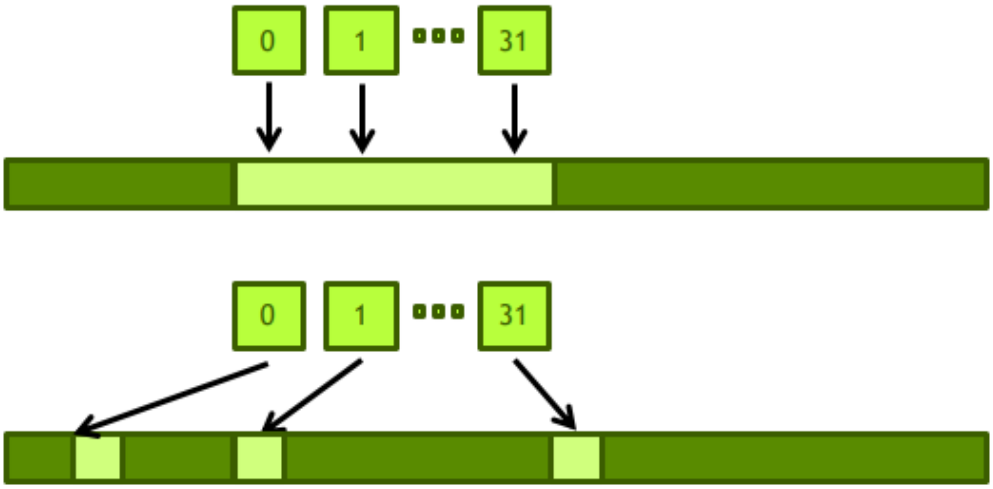


Εικόνα 11: Bank conflict access

Η εικόνα 10 δείχνει δύο τρόπους προσπέλασης της μνήμης, κατά τους οποίους δεν εμφανίζεται το φαινόμενο του bank conflict. Αντίθετα, αυτό γίνεται εμφανές στην εικόνα 11, καθώς υπάρχουνε τουλάχιστον δύο threads μέσα στο μισό warp τα οποία ζητάνε πρόσβαση σε στοιχεία του ίδιου bank ταυτόχρονα. Μία εξαίρεση του ορισμού του bank conflict είναι η περίπτωση όπου όλα τα thread του halfwarp προσπελαύνουν την ίδια θέση μνήμης, δηλαδή το ίδιο bank. Τότε η ζητούμενη τιμή διαβάζεται μόνο μία φορά και έπειτα μεταδίδεται (broadcasts) σε όλα τα threads.

Η πρόσβαση στην καθολική μνήμη (global memory access) γίνεται σε transactions ανά warp και είναι είτε των 32 είτε των 64 είτε των 128 byte. Για λόγους απόδοσης, 32 byte transaction συμβαίνει όταν ένα warp διαβάζει είσοδο της οποίας ο τύπος αναπαράστασης έχει μέγεθος 8 bit (1 byte), 64 byte όταν διαβάζει είσοδο της οποίας ο τύπος αναπαράστασης έχει μέγεθος 16 bit (2 byte) και 128 byte όταν διαβάζει είσοδο της οποίας ο τύπος αναπαράστασης έχει μέγεθος 32 bit (4 byte). [8] Το hardware προσπαθεί να μειώσει όσο το δυνατόν περισσότερο τον αριθμό των transactions. Όταν μία ομάδα 32 συνεχόμενων thread, δηλαδή ένα warp, προσπελαύνει γειτονικές θέσεις μνήμης τότε λέμε ότι η πρόσβαση είναι συνενωτική (coalesced access). Ως αποτέλεσμα, ο αριθμός των transactions μειώνεται και επιτυγχάνουμε μεγάλη χρηστικότητα (high utilization). Το αντίθετο αποτέλεσμα επιφέρει η πρόσβαση διάσπαρτων θέσεων μνήμης από ένα warp, δηλαδή πολλά transactions και χαμηλή χρηστικότητα.

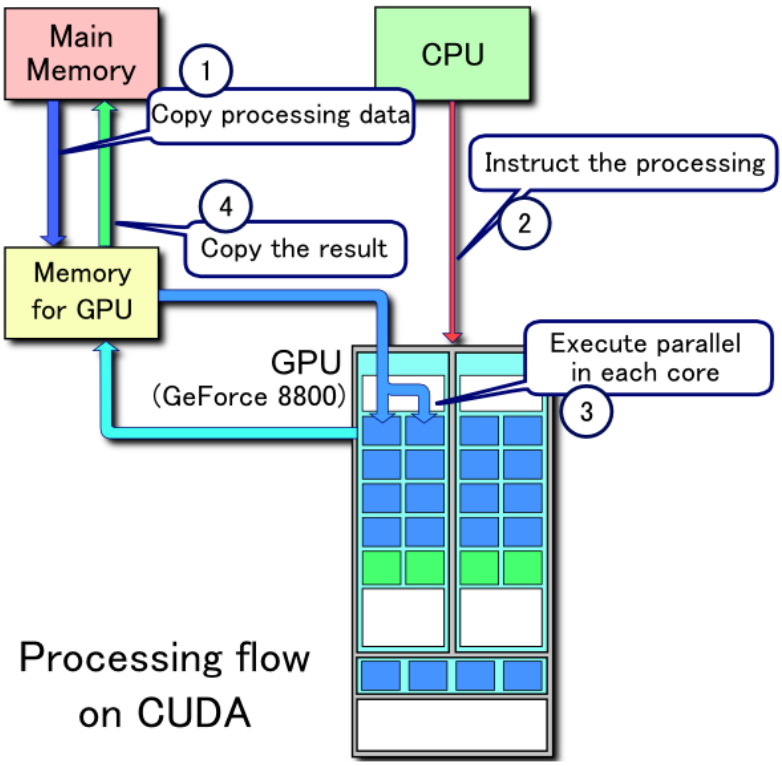
Στο πρώτο μέρος της εικόνας 12 παρουσιάζεται η περίπτωση της συνενωτικής πρόσβασης στη μνήμη, ενώ στο δεύτερο σκέλος της φαίνεται η μη συνενωτική πρόσβαση στη μνήμη.



Εικόνα 12: Συνενωτική και μη συνενωτική πρόσβαση μνήμης.

1.3.5 Ροή εκτέλεσης προγράμματος σε κάρτα γραφικών

Μία τυπική ροή εκτέλεσης ενός προγράμματος σε GPU παρουσιάζεται στην εικόνα 4. Αρχικά αντιγράφουμε τα δεδομένα από την κύρια μνήμη του υπολογιστή στην μνήμη της κάρτας γραφικών. Έπειτα καλούμε το κομμάτι κώδικα που θα εκτελεστεί στην κάρτα γραφικών (kernel). Ο κώδικας εκτελείται παράλληλα στην κάρτα γραφικών και μετά το πέρας της εκτέλεσης το αποτέλεσμα αντιγράφεται από την μνήμη της κάρτας γραφικών στην μνήμη του υπολογιστή.



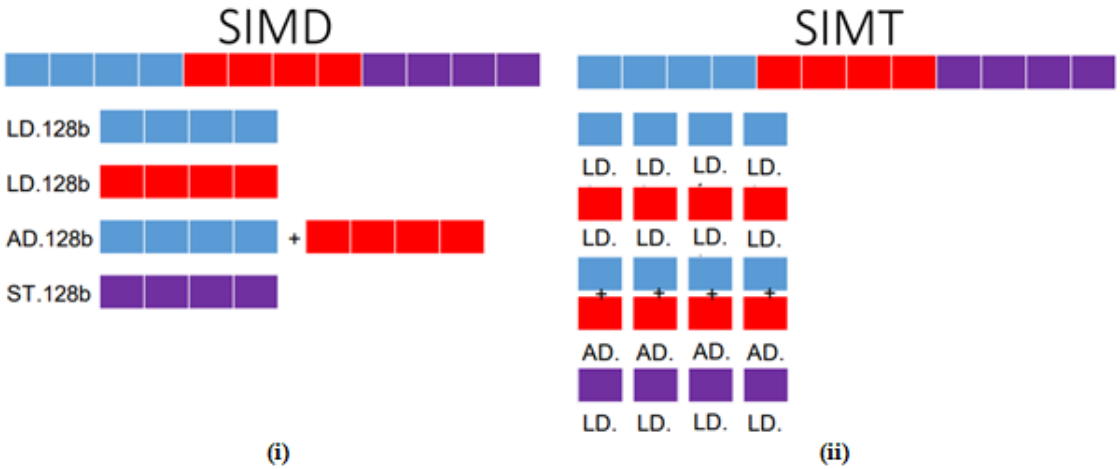
Εικόνα 13: Ροή εκτέλεσης προγράμματος σε κάρτα γραφικών.

1.4 Σύγκριση SIMD - SIMT

1.4.1 Περιορισμοί εκτέλεσης

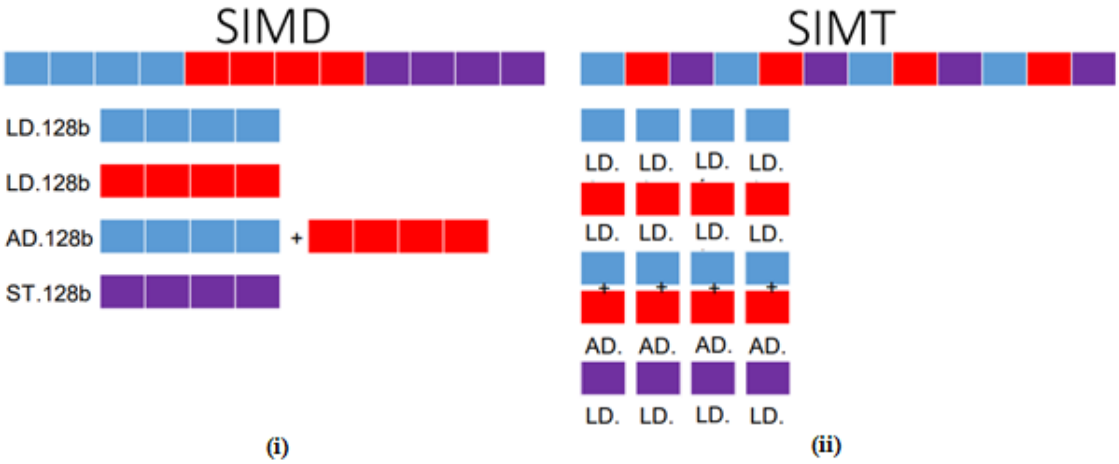
Στο πρότυπο SIMD, οι εντολές διανυσμάτων (vector instructions) εφαρμόζουν την ίδια λειτουργία πάνω σε πολλαπλά δεδομένα, το οποία πρέπει να φορτώνονται και να αποθηκεύονται σε συνεχόμενες θέσεις μνήμης. Οι εντολές αυτές δημιουργούνται είτε από τον προγραμματιστή είτε από τον μεταγλωττιστή (compiler).

Στο πρότυπο SIMT, κλιμακωτές εντολές (scalar instructions) εκτελούνται ταυτόχρονα από πολλαπλά νήματα του υλικού (hardware threads) και τα δεδομένα δεν χρειάζεται να είναι αποθηκευμένα σε συνεχόμενες θέσεις μνήμης. Το ίδιο το υλικό παρέχει την δυνατότητα παράλληλης εκτέλεσης των κλιμακωτών εντολών.



Εικόνα 14: Πρόσθεση δύο αριθμών που βρίσκονται σε συνεχόμενες θέσεις μνήμης, με βάση τις εντολές SIMD (i) και SIMT (ii).

Υπάρχει όμως μία μονόδρομη σχέση ανάμεσα στις εντολές SIMD και SIMT, όπως φαίνεται και στην εικόνα 15. Εάν μία διαδικασία μπορεί να τρέξει με εντολές τύπου SIMD τότε μπορεί και να τρέξει και με εντολές τύπου SIMT, χωρίς όμως να μπορεί να συμβεί και το αντίστροφο. Οι εντολές τύπου SIMT μπορούνε να χειριστούν ευκολότερα το φαινόμενο του dereferencing των δεδομένων [5].

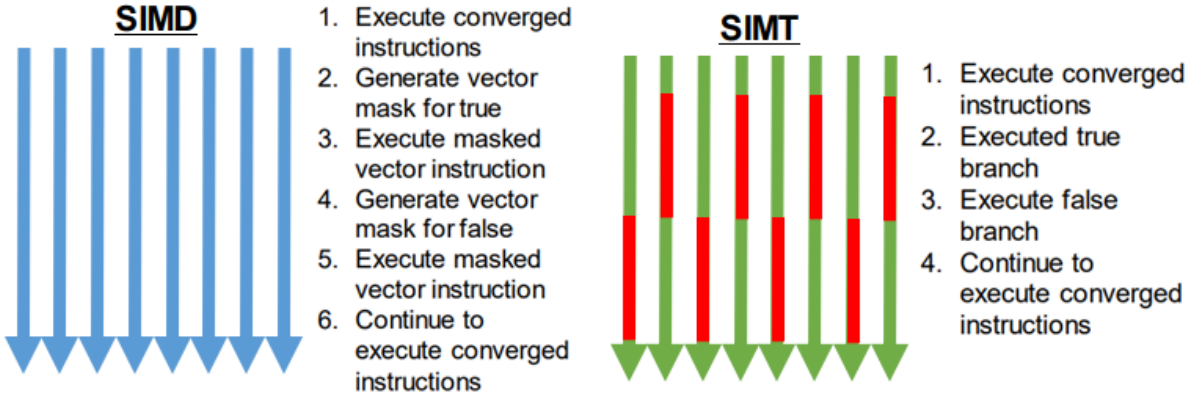


Εικόνα 15: Αδυναμία εκτέλεσης από το SIMD του μοτίβου εκτέλεσης του SIMT.

1.4.2 Διακλαδώσεις σε SIMD και SIMT

Κάθε αρχιτεκτονική διαχειρίζεται με διαφορετικό τρόπο τις διακλαδώσεις στον κώδικα. Στην SIMD αρχιτεκτονική ο compiler χειρίζεται τα branch divergence είτε φτιάχνοντας μάσκες είτε με gather/scatter operations. Αντίθετα, στην SIMT αρχιτεκτονική το υλικό διαχειρίζεται το branch divergence μέσα από predicated εντολές.

Η εικόνα 16 παρουσιάζει τον τρόπο και τα βήματα κατά τα οποία κάθε μία αρχιτεκτονική διαχειρίζεται τις διακλαδώσεις. Για την SIMD αρχιτεκτονική δημιουργείται μια μάσκα για τα taken paths και μία για τα not taken paths και εκτελούνται οι εντολές πάνω στα αντίστοιχα masked vectors. Για την SIMT αρχιτεκτονική, τα threads υπολογίζουν ένα κατηγόρημα και εκτελούν τις εντολές και των δύο διακλαδώσεων. Ο τρόπος με τον οποίο γίνεται η αποτίμηση θα φανεί σε ένα δοθέν παράδειγμα στην συνέχεια.



Εικόνα 16: Διακλαδώσεις σε SIMD και SIMT.

Η ύπαρξη του branch divergence στην SIMT αρχιτεκτονική επιφέρει και μείωση της απόδοσης. Το φαινόμενο αυτό εμφανίζεται όταν τα threads ενός warp εκτελούνε διαφορετικές εντολές, λόγω διαφορετικής αποτίμησης στις συνθήκες διακλάδωσης.

Στο πρώτο σκέλος της εικόνα 17 παρουσιάζεται ένα κομμάτι κώδικα το οποίο προκαλεί branch divergence όταν τα threads ενός warp αποτιμήσουν με διαφορετικό τρόπο την συνθήκη. Ο τρόπος με τον οποίο το χειριζεται αυτό το φαινόμενο ο compiler φαίνεται στο δεύτερο κομμάτι της εικόνας 17. Όλα τα threads θα υπολογίσουνε ένα κατηγόρημα και θα εκτελέσουνε τις αντίστοιχες εντολές βάση του υπολογισθέντος κατηγορήματος [9].

```
if (x<0.0)
  z = x-2.0;
else
  z = sqrt(x);
```

(i)

```
p = (x<0.0);
if (p)  z = x-2.0;      // single instruction
if (!p) z = sqrt(x);
```

(ii)

Εικόνα 17: Χειρισμός του branch divergence.

Η εντολή sqrt(x) θα είχε ως αποτέλεσμα την τιμή NaN αλλά δεν θα εκτελεστεί για την περίπτωση όπου $x < 0$. Αφού όλα τα threads θα εκτελέσουνε και τις δύο διακλαδώσεις, τότε το τελικό κόστος εκτέλεσης θα είναι ίσο με το άθροισμα του κόστους εκτέλεσης των δύο διακλαδώσεων. Στην περίπτωση που τα branch έχουνε πολλές εντολές, τότε ο nvcc compiler εφαρμόζει την τεχνική του warp voting, ώστε να ελέγξει εάν όλα τα thread του warp θα ακολουθήσουνε το ίδιο branch και εάν αυτό είναι αληθές τότε χρησιμοποιεί μόνο

το μονοπάτι που αποτιμάται ως taken από την συνθήκη. Το warp voting κοστίζει αρκετούς κύκλους εντολής, οπότε για απλές διακλαδώσεις χρησιμοποιείται η τεχνική του predication.

1.5 Υπερνημάτωση – Hyper-threading

Ο όρος Υπερνημάτωση (αγγλ. Hyper-threading ή Hyper-threading Technology), αποτελεί την ονομασία που έδωσε η Intel στη τεχνολογία ταυτόχρονης πολυνημάτωσης που ανέπτυξε η ίδια και υλοποίησε για πρώτη φορά το Φεβρουάριο του 2002 στους επεξεργαστές Xeon και Pentium 4. Αργότερα, η συγκεκριμένη τεχνολογία ενσωματώθηκε μεταξύ άλλων και στις σειρές επεξεργαστών Itanium, Atom και Core [10]. Κάθε φυσικός πυρήνας γίνεται αντιληπτός από το λειτουργικό σύστημα ως δύο λογικοί πυρήνες. Η ουσιαστική λειτουργία της υπερνημάτωσης είναι η μείωση του αριθμού των εξαρτημένων εντολών στο δίαυλο μεταφοράς εντολών του επεξεργαστή.

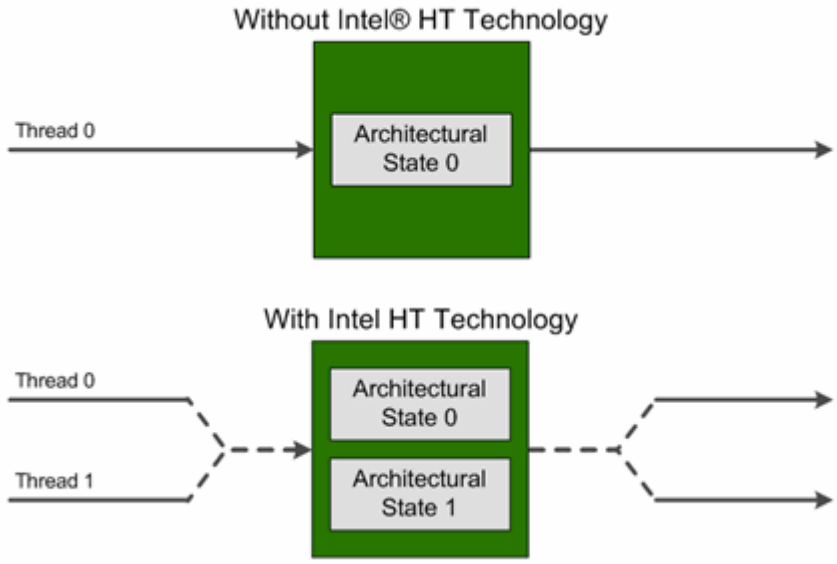
Η υπερνημάτωση απαιτεί από το λειτουργικό σύστημα να μπορεί να υποστηρίζει πολλαπλούς πυρήνες επεξεργασίας και να μπορεί να υποστηρίζει την λειτουργία αυτή. Η μελέτη έγινε με χρήση του λειτουργικού συστήματος Ubuntu 16.04, το οποίο και είναι ικανό να υποστηρίξει τέτοιου είδους λειτουργικότητα. Για το λειτουργικό σύστημα, η τεχνολογία της υπερνημάτωσης είναι αόρατη, με την έννοια ότι το λειτουργικό σύστημα αντιλαμβάνεται δύο «λογικούς» πυρήνες για κάθε φυσικό πυρήνα. Κάθε «λογικός» πυρήνας είναι ένα νήμα εκτέλεσης.

Σύμφωνα με την Intel, η πρώτη υλοποίηση της τεχνολογίας έδινε αύξηση των επιδόσεων του επεξεργαστή από 15%, για dual socket συστήματα, έως και 30%, για single socket συστήματα, με μόλις 5% αύξηση του μεγέθους του τσιπ.

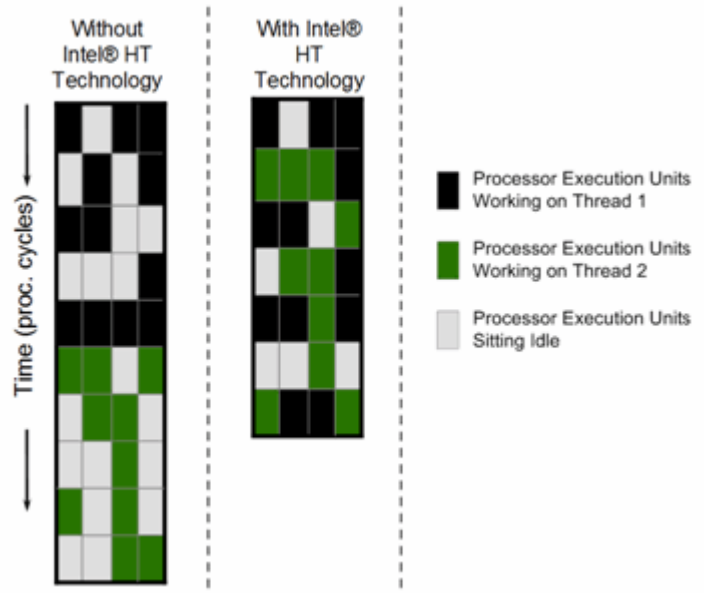
Στην τεχνολογία της υπερνημάτωσης, κάθε νήμα διατηρεί την δική του αρχιτεκτονική κατάσταση (αγγλ. Architectural state) αλλά οι περισσότεροι πόροι του επεξεργαστή, όπως οι μνήμες cache, οι ουρές εντολών (αγγλ. instruction queues), οι μονάδες εκτέλεσης (αγγλ. functional units), οι μονάδες fetch/decode, οι reorder buffers, τα store queues, τα global history arrays, το schedule και το retirement logic κλπ είναι διαμοιρασμένα ανάμεσα στα δύο νήματα του φυσικού επεξεργαστή. [11] Κατά την εκέλεση των νημάτων, κάποιες μονάδες εκτέλεσης του φυσικού πυρήνα παραμένουν αδρανείς για κάποιους κύκλους ρολογιού. Με την λειτουργία της υπερνημάτωσης, οι μονάδες εκτέλεσης μπορούν να επεξεργαστούν ταυτόχρονα διαφορετικές εντολές από δύο νήματα, γεγονός που οδηγεί σε καλύτερη χρησιμοποίηση των μονάδων εκτέλεσης και κατ’ επέκτασιν του επεξεργαστή, χωρίς κάποιες μονάδες εκτέλεσης να παραμένουν αδρανείς.

Η εικόνα 18 απεικονίζει με απλό τρόπο την αρχιτεκτονική της υπερνημάτωσης. Στο πρώτο μισό φαίνεται ένας πυρήνας που δεν υποστηρίζει την τεχνολογία της υπερνημάτωσης ενώ στο δεύτερο μισό απεικονίζεται η αρχιτεκτονική ενός επεξεργαστή που μπορεί και υποστηρίζει αυτήν τεχνολογία.

Στην εικόνα 19 φαίνεται το utilization των functional units για έναν επεξεργαστή με απενεργοποιημένη την λειτουργία hyperthreading (αριστερά) και με ενεργοποιημένη την λειτουργία αυτή (δεξιά). Ο επεξεργαστής αυτός διαθέτει τέσσερις μονάδες εκτέλεσης ανά πυρήνα (οριζόντιος άξονας), οι οποίες μπορούν να επεξεργαστούν ταυτόχρονα εντολές στον ίδιο κύκλο ρολογιού. Στον κατακόρυφο άξονα φαίνονται οι απαιτούμενοι κύκλοι ρολογιού για την διεκπαιρέωση των εργασιών. Για την πρώτη περίπτωση, χωρίς hyperthreading, απαιτούνται δέκα κύκλοι ρολογιού ενώ αντίθετα για την δεύτερη εφτά. Με λίγα λόγια, εάν αυτό το σενάριο αποτελούσε μια πραγματική περίπτωση, τότε θα είχαμε αύξηση της απόδοσης κατά 30%.



Εικόνα 18: Παρουσίαση την τεχνολογίας υπερνημάτωσης.



Εικόνα 19: Core utilization χωρίς και με ενεργοποιημένη την υπερνημάτωση.

Βέβαια, τα πλεονεκτήματα της SMT (Simultaneous Multi-Threading) αρχιτεκτονικής εξαρτώνται και από τον τύπο της εφαρμογής αλλά και από το επίπεδο fine tunning της εκτέλεσης. [12] Ως παράδειγμα έχουμε μία πολύ βελτιστοποιημένη έκδοση πολλαπλασιασμού πινάκων, η οποία και εκμεταλλεύεται πλήρως τους καταχωρητές, την μνήμη cache και πολλαπλά functional units. Η χρησιμοποίηση περαιτέρω νημάτων μπορεί να επιφέρει βλάβη σε αυτήν την καλή χρησιμοποίηση των πόρων με συνέπεια την μείωση της απόδοσης. Αντίθετα, μια μη βελτιστοποιημένη εκδοχή εκτέλεσης πολλαπλασιασμού πινάκων, η οποία και δεν εκμετελλεύεται τους διαθέσιμους πόρους επαρκώς, πιθανώς να εποφελούνταν από την SMT αρχιτεκτονική και την προσθήκη νημάτων στον επεξεργαστή.

Το σύνολο των δεδομένων τα οποία πρέπει να φορτωθούν στους καταχωρητές πριν μία διεργασία συνεχίσει την εκτέλεσή της στην CPU ονομάζεται hardware context. Το hardware context αποτελεί υποσύνολο του execution context της διεργασίας, το οποίο περιέχει όλη την απαραίτητη πληροφορία για την εκτέλεση της διεργασίας.

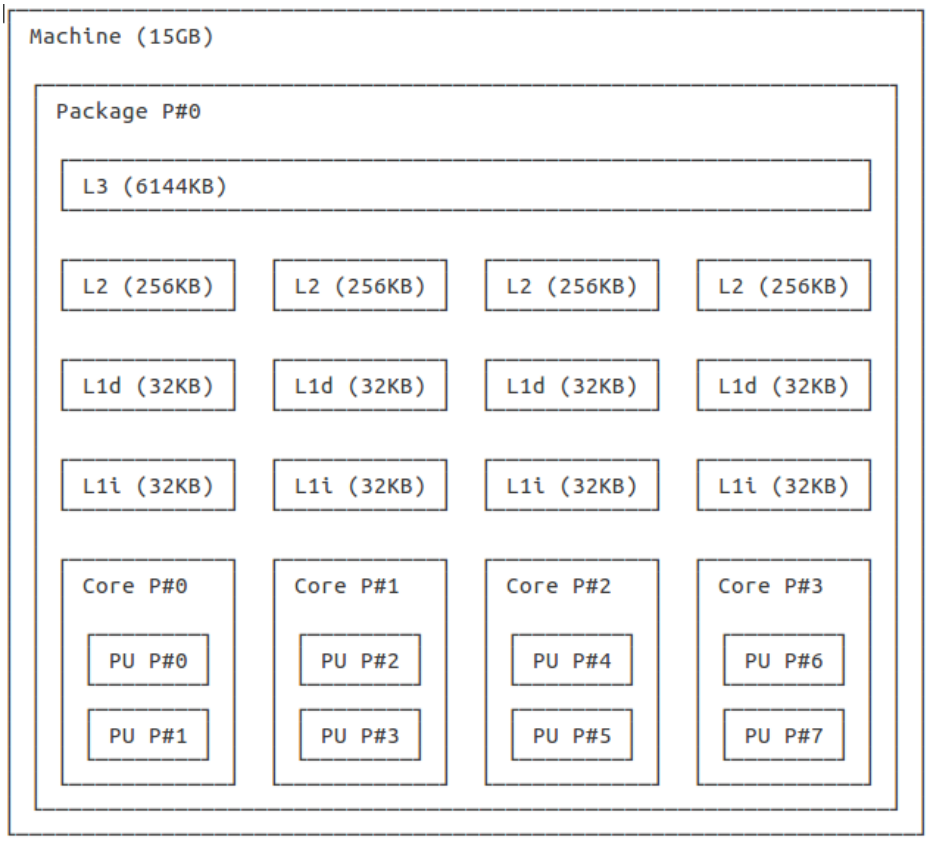
Κατά την εκτέλεση, όλα τα hardware contexts είναι ταυτόχρονα ενεργά και ανταγωνίζονται σε κάθε κύκλο ρολογιού για τους διαθέσιμους κοινούς πόρους. Αυτός ο δυναμικός διαμοιρασμός των functional units επιτρέπει στην SMT αρχιτεκτονική την αύξηση του throughput καταπολεμώντας τα δύο κύρια εμπόδια της χρήσης του επεξεργαστή, τα φαινόμενα καθυστερήσεων (latencies) καθώς και τον περιορισμένο παραλληλισμό σε επίπεδο εντολών (Instruction Level Parallelism - ILP) ανά thread. Παρόλα αυτά η ευελιξία αυτή εισάγει και ένα κόστος. Όταν πολλαπλά νήματα είναι ενεργά, τότε ο στατικός διαμοιρασμός των κοινών πόρων, για τους οποίους έχει γίνει σχετική μνεία, επηρεάζει κώδικες με σχετικά μεγάλο instruction throughput. Αυτός ο στατικός διαμοιρασμός, στην περίπτωση ταυτόσημων thread-level instruction streams, περιορίζει την απόδοση αλλά για την περίπτωση ανόμοιων streams μπορεί να δράσει ευγεργετικά. [13]

Στην περίπτωση των ταυτόχρονων αιτημάτων για χρήση των κοινών πόρων, η πρόσβαση σε αυτούς εναλλάσσεται ανά νήμα, συνήθως με έναν πιο fine-grained τρόπο (cycle-by-cycle). Αυτό έχει ως αποτέλεσμα, ενώσω και τα δύο νήματα είναι ενεργά, το μέγιστο bandwidth για instruction fetch, decode, issue, execution και retirement να ισούται περίπου με το μισό των δυνατοτήτων του πυρήνα. Επίσης, η αρχιτεκτονική κατάσταση, σε έναν επεξεργαστή με ενεργοποιημένη την λειτουργία της υπερνημάτωσης, αντιγράφεται ανά νήμα, όπως αντιγράφονται και οι instruction pointers, τα ITLBs, οι branch history buffers, το return stack αλλά και το renaming logic. Τα buffering queues, ο reorder buffer, καθώς επίσης και τα store/load queues διαμοιράζονται στατικά, έτσι ώστε κάθε ενεργό νήμα να μπορεί να χρησιμοποιήσει τις μισές θέσεις, δίνοντας έτσι τη δυνατότητα σε δύο νήματα να τρέχουν ανεξάρτητα μεταξύ τους και χωρίς η εκτέλεση του ενός να επηρεάζει την εκτέλεση του άλλου.

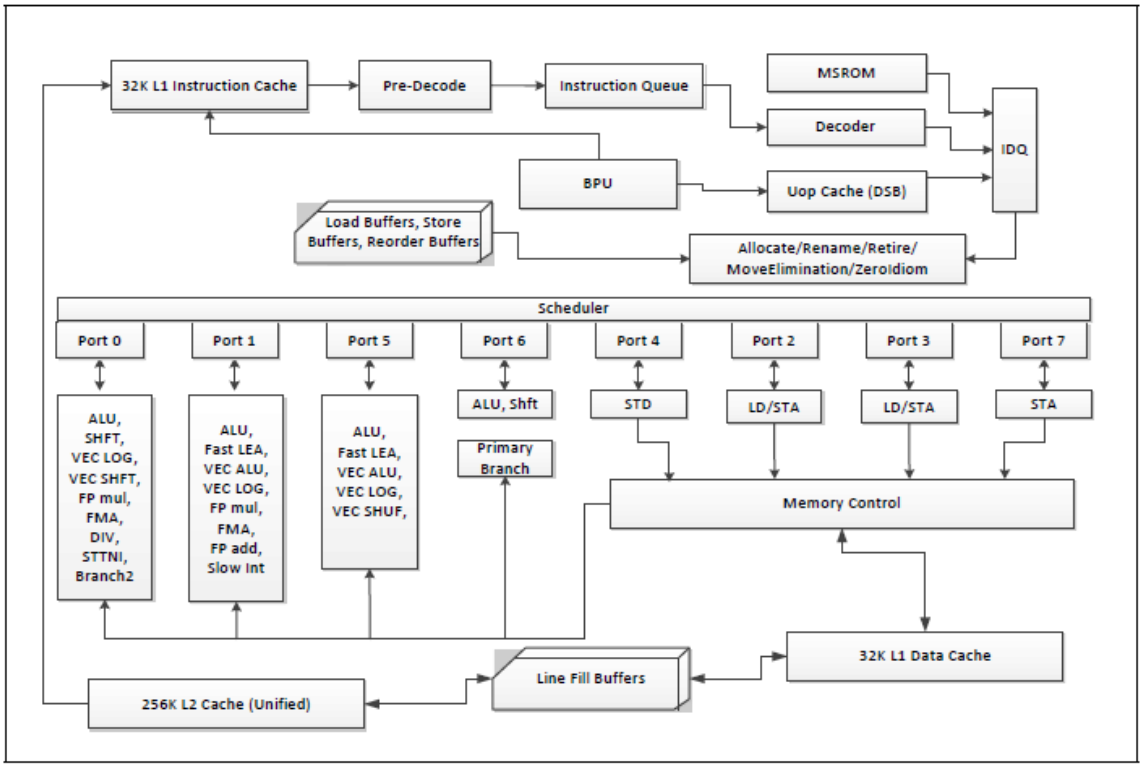
Ο scheduler του λειτουργικού Linux, χρησιμοποιεί την εντολή halt για να βελτιώσει την απόδοση των διεργασιών που τρέχουν. Όταν υπάρχει μόνο ένα ενεργό νήμα στον φυσικό πυρήνα, ο scheduler κάνει χρήση αυτής της εντολής στον ανενεργό «λογικό» πυρήνα έτσι ώστε να μπει ο φυσικός πυρήνας σε λειτουργία «μονού-νήματος» (Single Threaded). Αυτός ο χειρισμός του scheduler έχει ως αποτέλεσμα το ενεργό νήμα να μπορεί να εκμεταλλευτεί πλήρως όλα τα resources του επεξεργαστή. Τέλος, ο scheduler χρησιμοποιεί την εντολή halt και σε περίπτωση που δύο νήματα έχουνε διαφορετική προτεραιότητα. Έτσι το νήμα με μεγαλύτερη προτεραιότητα μπορεί να χρησιμοποιήσει τον πυρήνα σε λειτουργία Single Threaded.

Χρησιμοποιώντας την εντολή *lstopo –output-format txt –v –no-io --no-legend > lstopo.txt* του Unix, αφού πρώτα έχουμε εγκαταστήσει το πακέτο hwloc (portable hardware locality), μπορούμε να φτιάξουμε ένα απλοποιημένο διάγραμμα ASCII με την μορφή του επεξεργαστή μας σε ένα αρχείο τύπου txt. Η εικόνα 20 δείχνει το διάγραμμα αυτό για τον επεξεργαστή μας. Παρατηρούμε ότι κάθε φυσικός επεξεργαστής αποτελείται από δύο «λογικούς» επεξεργαστές, καθώς επίσης και ότι κάθε φυσικός επεξεργαστής έχει απομονωμένη μνήμη cache επιπέδου L1 (instruction & data) και L2.

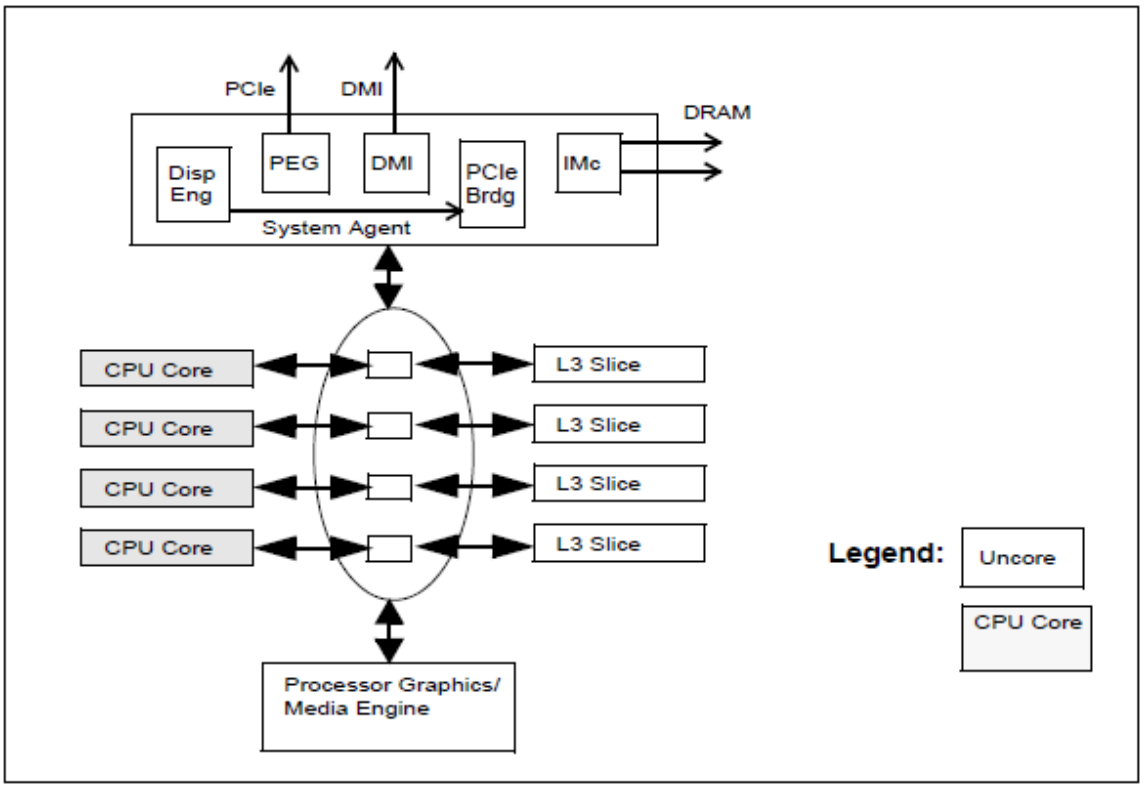
Η εικόνα 21 δείχνει την λειτουργικότητα του pipeline για την μικροαρχιτεκτονική του επεξεργαστή που χρησιμοποιούμε, γενιάς Intel Haswell. Στην εικόνα 22 βλέπουμε την αρχιτεκτονική ενός τετραπύρηνου συστήματος ενώ η εικόνα 23 δείχνει σε πόσα ports βρίσκονται οι αντίστοιχες μονάδες εκτέλεσης καθώς επίσης και τις εντολές που μπορούν να εκτελεσθούν σε αυτές.



Εικόνα 20: Τοπολογία επεξεργαστή - Διάγραμμα ASCII



Εικόνα 21: CPU Core Pipeline Functionality of the Haswell Microarchitecture.



Εικόνα 22: Four Core System Integration of the Haswell Microarchitecture.

Execution Unit	# of Ports	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa
SHFT	2	sal, shl, rol, adc, sarx, (adcx, adox)[1] etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc
SIMD Log	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)blendp*, vpblendd
SIMD_Shft	1	(v)psl*, (v)psr*
SIMD ALU	2	(v)padd*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)pcmpgt*
Shuffle	1	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)pblendw
SIMD Misc	1	(v)pmul*, (v)pmadd*, STTNI, (v)pclmulqdq, (v)psadw, (v)pcmpgtq, vpsllvd, (v)bendv*, (v)plendw,
FP Add	1	(v)addp*, (v)cmpp*, (v)max*, (v)min*,
FP Mov	1	(v)movap*, (v)movup*, (v)movsd/ss, (v)movd gpr, (v)andp*, (v)orp*
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

Εικόνα 23: Μονάδες εκτέλεσης και αντίστοιχες εντολές εκτέλεσεις για αυτές τις μονάδες - Μικροαρχιτεκτονική Haswell

Κεφάλαιο 2 – Αντικείμενο μελέτης

2.1 Περιγραφή προβλημάτων προς μελέτη

Στόχος μας είναι να μελετήσουμε δύο κατηγορίες προβλημάτων που αφορούν ερωτήματα προς βάσεις δεδομένων. Το πρώτο πρόβλημα που θα μελετήσουμε αφορά το filtering σε πίνακες βάσεων δεδομένων ενώ το δεύτερο αφορά μία κατηγορία από aggregation functions. Τέτοιου είδους βασικά ερωτήματα βρίσκουν συχνή εφαρμογή στον κλάδο της επιστήμης των δεδομένων και αποτελούν πολλές φορές τμήματα μεγαλύτερων και πιο πολύπλοκων ερωτημάτων.

Όσον αφορά το πρώτο πρόβλημα προς μελέτη, αυτό του filtering, στόχος μας είναι να βρούμε όλες τις εγγραφές του δειγματος με τιμή ίση με αυτή της τιμής-στόχου της εισόδου, με προκαθορισμένο selectivity κάθε φορά έτσι ώστε να αποφανθούμε ποιά από τις τρεις αρχιτεκτονικές που εξετάζουμε είναι η πιο αποτελεσματική για αυτό το είδος προβλήματος. Το παρόν πρόβλημα περιγράφεται από το ακόλουθο SQL statement:

```
SELECT * FROM TABLE_NAME WHERE TABLE_COLUMN = INPUT_TARGET;
```

Το υπό εξέταση πρόβλημα των aggregation functions χωρίζεται σε μικρότερα υποπροβλήματα. Πιο συγκεκριμένα, θα εξετάσουμε τα SUM, COUNT, MIN και MAX aggregation functions, όπου στόχος μας είναι να βρούμε το άθροισμα των στοιχείων εισόδου, να μετρήσουμε το πλήθος τους και να βρούμε την ελάχιστη και την μέγιστη τιμή μέσα στο δείγμα αντιστοίχως. Τα τέσσερα aggregation functions με τα οποία θα ασχοληθούμε, μπορούν να εκφραστούν αντιστοίχως με τα ακόλουθα SQL statements:

```
SELECT SUM(TABLE_COLUMN) FROM TABLE_NAME;
```

```
SELECT COUNT(1) FROM TABLE_NAME;
```

```
SELECT MIN(TABLE_COLUMN) FROM TABLE_NAME;
```

```
SELECT MAX(TABLE_COLUMN) FROM TABLE_NAME;
```

2.2 Υλοποιημένοι κώδικες - Scalar

2.2.1 Filtering

```
double scalar_equality_select(int32_t** relation, const size_t relation_size,
                              const int32_t value, int8_t** bitmap) {
  clock_t start, end;
  double select_duration;
  int i, j, bitmap_offset = 0;
  posix_memalign((void**)bitmap, 32, (relation_size / 8) * sizeof(int8_t));
  start = clock();
  for (i = 0; i < relation_size; i += 8) {
    int8_t bitmask = 0x00;
    for (int j = i; j < i + 8; ++j) {
      if ((*relation)[j] == value)
          bitmask |= 0x01 << (j-i);
    }
    (*bitmap)[bitmap_offset] = bitmask;
    bitmap_offset += 1;
  }
  end = clock();
  return ((double) (end - start)) / CLOCKS_PER_SEC;
}
```

Κώδικας 1: Scalar Filtering.

Για το πρόβλημα του filtering, παράγουμε ένα bitmap που δείχνει τις θέσεις εκείνων των στοιχείων που έχουν τιμή ίση με την τιμή στόχο. Το παραχθέν bitmap τροφοδοτείται ως είσοδος σε μία συνάρτηση hash και παράγεται ένας ακέραιος αριθμός ως τελική έξοδος του προγράμματος. Η συνάρτηση hash, για την οποία έγινε λόγος, παρουσιάζεται παρακάτω.

```
int32_t hash_bitmap(const int8_t* bitmap, const size_t relation_size) {
  size_t i;
  int32_t result;
  const size_t bitmap_size = relation_size / 8;
  for (i = 0; i < bitmap_size; ++i) {
    uint32_t v = bitmap[i];
    v ^= v << 13;
    v ^= v >> 17;
    v ^= v << 5;
    result ^= v;
  }
  return result;
}
```

Κώδικας 2: Bitmap hash function.

2.2.2 Aggregation functions – SUM

Στον παρακάτων κώδικα, για την scalar αρχιτεκτονική, αθροίζουμε τις τιμές όλων των στοιχείων εισόδου.

```
unsigned long scalar_sum(int32_t** relation, int32_t rel_size, double* function_time){

  clock_t t;
  unsigned long sum = 0; // prevent overflow phenomenon
  t = clock();
  for(int i = 0 ; i < rel_size ; i++){
    sum += (*relation)[i];
  }
  t = clock() - t;
  (*function_time) = ((double) t) / CLOCKS_PER_SEC;
  return sum;
}
```

Κώδικας 3: Scalar Sum.

2.2.3 Aggregation functions – COUNT

Ο κώδικας ο οποίος μετράει τον αριθμό των στοιχείων εισόδου και προσομοιάζει το count aggregation function παρουσιάζεται παρακάτω.

```
long scalar_count(int32_t** relation, int32_t rel_size, double* function_time){

  clock_t t;
  long count = 0L;
  t = clock();
  for(int i = 0 ; i < rel_size ; i++){
    count++;
  }
  t = clock() - t;
  (*function_time) = ((double) t) / CLOCKS_PER_SEC;
  return count;
}
```

Κώδικας 4: Scalar Count.

Τέλος, για την scalar αρχιτεκτονική παρουσιάζονται στην συνέχεια οι κώδικες για τα Min και Max aggregation functions αντίστοιχα. Τα προβλήματα είναι δυϊκώς αντίστροφα και αυτό αποτυπώνεται και στους κώδικες.

2.2.4 Aggregation functions – MIN

```
int scalar_min(int32_t** relation, int32_t rel_size, double* function_time){

  clock_t t;
  int32_t min = INT_MAX;
  t = clock();
  for(int i = 0 ; i < rel_size ; i++){
    if( (*relation)[i] < min ){
      min = (*relation)[i];
    }
  }
  t = clock() - t;
  (*function_time) = ((double) t) / CLOCKS_PER_SEC;
  return min;
}
```

Κώδικας 5: Scalar Min.

2.2.5 Aggregation functions - MAX

```
int scalar_max(int32_t** relation, int32_t rel_size, double* function_time){

  clock_t t;
  int max = INT_MIN;
  t = clock();
  for(int i = 0 ; i < rel_size ; i++){
    if( (*relation)[i] > max ){
      max = (*relation)[i];
    }
  }
  t = clock() - t;
  (*function_time) = ((double) t) / CLOCKS_PER_SEC;
  return max;
}
```

Κώδικας 6: Scalar Max.

2.3 Υλοποιημένοι κώδικες - SIMD

2.3.1 Filtering

```
double avx_equality_select(int32_t** relation, const size_t relation_size,
                           const int32_t value, int8_t** bitmap) {
  clock_t start, end;
  double select_duration;
  int i, bitmap_offset = 0;
  int32_t* mask;
  __m256i shift_buffer = _mm256_set_epi32(0x80, 0x40, 0x20, 0x10,
      0x08, 0x04, 0x02, 0x01);
  posix_memalign((void**) &mask, 32, 8 * sizeof(int32_t));
  posix_memalign((void**)bitmap, 32, (relation_size / 8) * sizeof(int8_t));
  for (i = 0; i < 8; ++i) {
    mask[i] = value;
  }
  __m256i equality_mask = _mm256_stream_load_si256((__m256i*) mask);
  __m256i equality_buffer;
  start = clock();
  for (i = 0; i < relation_size; i += 8) {
    equality_buffer = _mm256_stream_load_si256((__m256i*)&(*relation)[i]);
    __m256i eq_buffer = _mm256_cmpeq_epi32(equality_mask, equality_buffer);
    __m256i result_buffer = _mm256_and_si256(eq_buffer, shift_buffer);
    int8_t bitmask = 0x00;
    int* eq_ptr = (int*)&result_buffer;
    bitmask |= eq_ptr[0];
    bitmask |= eq_ptr[1];
    bitmask |= eq_ptr[2];
    bitmask |= eq_ptr[3];
    bitmask |= eq_ptr[4];
    bitmask |= eq_ptr[5];
    bitmask |= eq_ptr[6];
    bitmask |= eq_ptr[7];
    (*bitmap)[bitmap_offset] = bitmask;
    bitmap_offset += 1;
  }
  end = clock();
  return ((double) (end - start)) / CLOCKS_PER_SEC;
}
```

Κώδικας 7: AVX Filtering.

Στον παραπάνω κώδικα αξίζει να σταθούμε σε τρία σημεία-εντολές. Οι εντολές είναι οι εξής:

- **_mm256_stream_load_si256():** Με αυτήν την εντολή φορτώνουμε οκτώ ακέραιους αριθμούς σε διάνυσμα τύπου AVX2, έναν καταχωρητή τύπου YMM δηλαδή.
- **_mm256_cmpeq_epi32():** Η εντολή αυτή συγκρίνει δύο διανύσματα AVX2 και επιστρέφει ως αποτέλεσμα ένα διάνυσμα μεγέθους οκτώ κελιών, όπου το κάθε κελί περιέχει τιμή ίση με μηδέν ή ένα. Τιμή ίση με ένα προκύπτει σε περίπτωση ισότητας των δύο συγκρινόμενων τιμών, ενώ μηδενική τιμή σε αντίθετη περίπτωση.
- **_mm256_and_si256():** Εφαρμόζει την συνάρτηση λογικού «και» (logical AND) μεταξύ δύο AVX2 διανυσμάτων στα αντίστοιχα στοιχεία τους.

2.3.2 Aggregation functions - SUM

```
double avx_sum(int32_t** relation, int32_t rel_size, int** result){

  clock_t t;
  __m256i sum = _mm256_setzero_si256();
  __m256i input_buffer;
  t = clock();
  for(int32_t i = 0 ; i < rel_size ; i += 8 ){
    input_buffer = _mm256_stream_load_si256((__m256i*)&(*relation)[i]);
    sum = _mm256_add_epi32(input_buffer, sum);
  }
  t = clock() - t;
  double time_result = ((double) t) / CLOCKS_PER_SEC;
  (*result) = (int*)&sum;
  return time_result;
}
```

Κώδικας 8: AVX Sum.

Με την εντολή *_mm256_add_epi32()* αθροίζουμε καθ' αντιστοιχία δύο διανύσματα τύπου AVX-256.

2.3.3 Aggregation functions - COUNT

```
double avx_count(int32_t** relation, int32_t rel_size, int** result){

  clock_t t;
  __m256i count = _mm256_setzero_si256();
  __m256i input_buffer = _mm256_set1_epi32(1);
  t = clock();
  for(int32_t i = 0 ; i < rel_size ; i += 8 ){
    count = _mm256_add_epi32(input_buffer, count);
  }
  t = clock() - t;
  double time_result = ((double) t) / CLOCKS_PER_SEC;
  (*result) = (int*)&count;
  return time_result;
}
```

Κώδικας 9: AVX Count.

2.3.4 Aggregation functions - MIN

```
double avx_min(int32_t** relation, int32_t rel_size, int** result){

  clock_t t;
  __m256i input_buffer;
  __m256i min = _mm256_set_epi32(INT32_MAX, INT32_MAX, INT32_MAX, INT32_MAX,
                                 INT32_MAX, INT32_MAX, INT32_MAX, INT32_MAX);
  t = clock();
  for(int i = 0 ; i < rel_size ; i += 8){
    input_buffer = _mm256_stream_load_si256((__m256i*)&(*relation)[i]);
    min = _mm256_min_epi32(min, input_buffer);
  }
  t = clock() - t;
  double time_result = ((double) t) / CLOCKS_PER_SEC;
  (*result) = (int*)&min;
  return time_result;
}
```

Κώδικας 10: AVX Min.

Η εντολή *_mm256_min_epi32()* συγκρίνει τα δύο διανύσματα τύπου AVX2 που λαμβάνει ως παραμέτρους και επιστρέφει ως αποτέλεσμα ένα διάνυσμα τύπου AVX2 μεγέθους οκτώ ακεραίων, έχοντας σε κάθε θέση την μικρότερη από τις δύο τιμές προς σύγκριση.

2.3.5 Aggregation functions - MAX

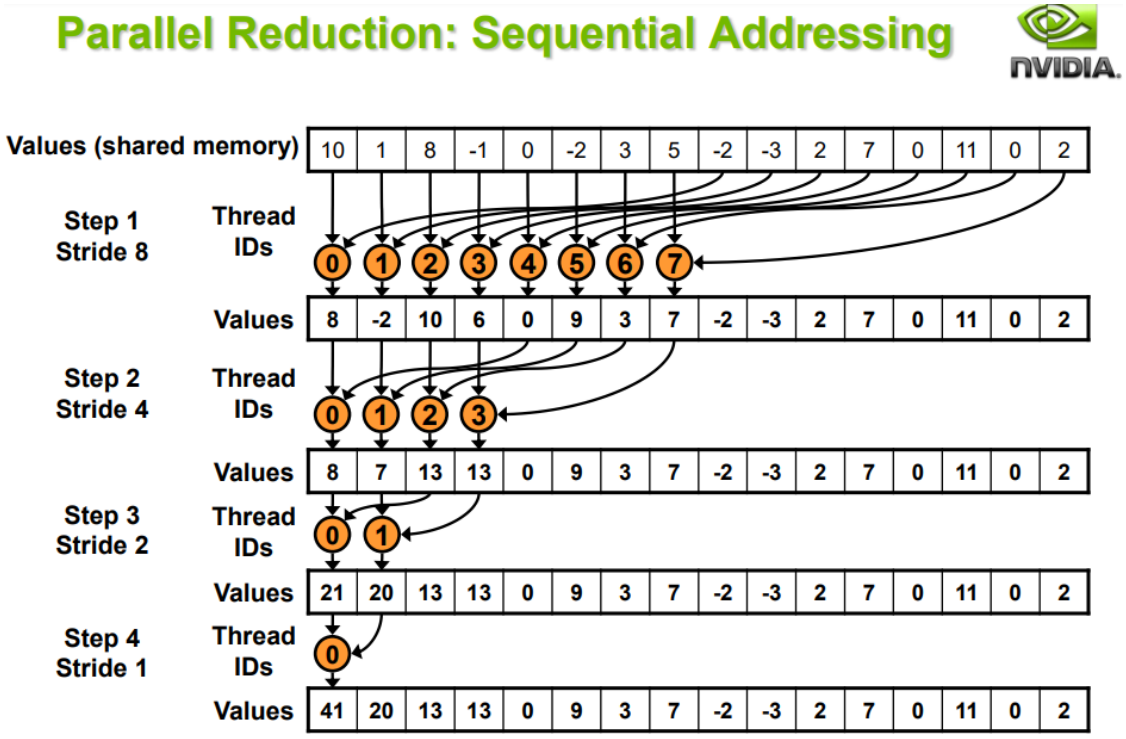
```
double avx_max(int32_t** relation, int32_t rel_size, int** result){

    clock_t t;
    __m256i input_buffer;
    __m256i max = _mm256_set_epi32(INT32_MIN, INT32_MIN, INT32_MIN, INT32_MIN,
                                   INT32_MIN, INT32_MIN, INT32_MIN, INT32_MIN);
    t = clock();
    for(int i = 0 ; i < rel_size ; i += 8){
      input_buffer = _mm256_stream_load_si256((__m256i*)&(*relation)[i]);
      max = _mm256_max_epi32(max, input_buffer);
    }
    t = clock() - t;
    double time_result = ((double) t) / CLOCKS_PER_SEC;
    (*result) = (int*)&max;
    return time_result;
}
```

Κώδικας 11: AVX Max.

Η εντολή *_mm256_max_epi32()* είναι η δυϊκώς αντίθετη της εντολής *_mm256_min_epi32()* που αναφέρθηκε νωρίτερα. Το διάνυσμα που επιστρέφεται ως αποτέλεσμα περιέχει την μέγιστη από τις δύο, κάθε φορά, συγκρινόμενες τιμές και είναι επίσης μεγέθους οκτώ ακεραίων.

2.4 Υλοποιημένοι κώδικες - SIMT

Για την υλοποίηση των αντίστοιχων αλγορίθμων για την SIMT αρχιτεκτονική, χρησιμοποιήσαμε την τεχνική της «παράλληλης μείωσης» (Parallel reduction) [14]. Σύμφωνα με αυτην την τεχνική, ένας αριθμός thread επεξεργάζεται έναν αριθμό στοιχείων στην μνήμη και σε κάθε βήμα ο αριθμός των στοιχείων προς επεξεργασία και των thread που τα επεξεργάζονται μειώνονται λογαριθμικά. Ένα τέτοιο παραδειγμα φαίνεται στην εικόνα 24, όπου τα στοιχεία της κοινής μνήμης αθροίζονται μέχρις ότου να προκύψει το τελικό αποτέλεσμα του βήματος 4.



Εικόνα 24: Απεικόνιση της τεχνικής Parallel Reduction.

Στο δοθέν παράδειγμα, σκοπός μας είναι να αθροίσουμε τα δεκαέξι στοιχεία του πίνακα εισόδου. Αρχικά χρησιμοποιούνται οκτώ thread, με το καθένα να επεξεργάζεται δύο στοιχεία του πίνακα. Στο δεύτερο βήμα, αφού τα στοιχεία προς άθροιση έχουνε μειωθεί στο μισό, στο μισό μειώνεται και ο αριθμός των ενεργών thread. Στο παράδειγμά μας, αυτή η μείωση συνεχίζεται εώς ότου μείνουνε δύο στοιχεία προς άθροιση, την οποία πράξη αναλαμβάνει να εκτελέσει ένα και μόνο thread.

Συνεπώς απαιτούνται τέσσερα βήματα για την ολοκλήρωση της διαδικασίας άθροισης δεκαέξι στοιχείων, με την μείωση των υπό εξέταση στοιχείων να είναι λογαριθμική σε κάθε βήμα. Άλλωστε, $\log_2 16 = 4$.

2.4.1 Filtering

```
template<unsigned int blockSize>
__global__ void reduce_scan_kernel(int8_t *g_idata, int8_t *g_odata,
                                   int chunkSize, int8_t target_value){

  unsigned int i = blockIdx.x * (blockSize * 2) + threadIdx.x;
  unsigned int gridSize = blockSize * 2 * gridDim.x;

  while (i < chunkSize) {
    // avoid divergency, conditions are evaluated either to 1 for true
    // or 0 for false, from all threads in the warp
    g_odata[i] = (int8_t) (g_idata[i] == target_value);
    g_odata[i+blockSize] = (int8_t) (g_idata[i+blockSize] == target_value);
    i += gridSize;
  }
}
```

Κώδικας 12: SIMT Filtering

2.4.2 Aggregation functions – SUM

```
template<unsigned int blockSize>
__global__ void reduce_sum_kernel(int8_t *g_idata, int *g_odata, int chunkSize){
  extern __shared__ int sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * (blockSize * 2) + threadIdx.x;
  unsigned int gridSize = blockSize * 2 * gridDim.x;

  sdata[tid] = 0;

  while (i < chunkSize) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
  }
  __syncthreads();

  if(blockSize >= 1024){
    if(tid < 512){
      sdata[tid] += sdata[tid + 512];
    }
    __syncthreads();
  }

  if(blockSize >= 512){
    if(tid < 256){
      sdata[tid] += sdata[tid + 256];
    }
    __syncthreads();
  }

  if(blockSize >= 256){
    if(tid < 128){
      sdata[tid] += sdata[tid + 128];
    }
    __syncthreads();
  }
```

```
  if(blockSize >= 128){
    if(tid < 64){
      sdata[tid] += sdata[tid + 64];
    }
    __syncthreads();
  }

  if (tid < 32) {
    warpReduce<blockSize>(sdata, tid);
  }
  //write result for this block to global mem
  if(tid == 0){
    g_odata[blockIdx.x] = sdata[0];
  }
}
```

Κώδικας 13: SIMT Sum

Σε επίπεδο warp κάνουμε reduce με το παρακάτω μπλοκ κώδικα:

```
template<unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid){
  if(blockSize >= 64) sdata[tid] += sdata[tid+32];
  if(blockSize >= 32) sdata[tid] += sdata[tid+16];
  if(blockSize >= 16) sdata[tid] += sdata[tid+8];
  if(blockSize >= 8) sdata[tid] += sdata[tid+4];
  if(blockSize >= 4) sdata[tid] += sdata[tid+2];
  if(blockSize >= 2) sdata[tid] += sdata[tid+1];
}
```

Κώδικας 14: SIMT Sum Warp reduce

2.4.3 Aggregation functions – COUNT

```
template<unsigned int blockSize>
__global__ void reduce_count_kernel(int8_t *g_idata, int *g_odata, int chunkSize){
  extern __shared__ int sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * (blockSize * 2) + threadIdx.x;
  unsigned int gridSize = blockSize * 2 * gridDim.x;

  sdata[tid] = 0;

  while (i < chunkSize) {
    sdata[tid] += 2;
    i += gridSize;
  }
  __syncthreads();

  if(blockSize >= 1024){
    if(tid < 512){
      sdata[tid] += sdata[tid + 512];
    }
    __syncthreads();
  }
```

```
  if(blockSize >= 512){
    if(tid < 256){
      sdata[tid] += sdata[tid + 256];
    }
    __syncthreads();
  }

  if(blockSize >= 256){
    if(tid < 128){
      sdata[tid] += sdata[tid + 128];
    }
    __syncthreads();
  }

  if(blockSize >= 128){
    if(tid < 64){
      sdata[tid] += sdata[tid + 64];
    }
    __syncthreads();
  }

  if (tid < 32) {
    warpReduce<blockSize>(sdata, tid);
  }
  //write result for this block to global mem
  if(tid == 0){
    g_odata[blockIdx.x] = sdata[0];
  }
}
```

Κώδικας 15: SIMT Count

Σε επίπεδο warp κάνουμε reduce με το παρακάτω μπλοκ κώδικα:

```
template<unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid){
  if(blockSize >= 64) sdata[tid] += sdata[tid+32];
  if(blockSize >= 32) sdata[tid] += sdata[tid+16];
  if(blockSize >= 16) sdata[tid] += sdata[tid+8];
  if(blockSize >= 8) sdata[tid] += sdata[tid+4];
  if(blockSize >= 4) sdata[tid] += sdata[tid+2];
  if(blockSize >= 2) sdata[tid] += sdata[tid+1];
}
```

Κώδικας 16: SIMT Count Warp reduce

2.4.4 Aggregation functions – MIN

```
template<unsigned int blockSize>
__global__ void reduce_min_kernel(int8_t *g_idata, int8_t *g_odata, int chunkSize){
  extern __shared__ int sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * (blockSize * 2) + threadIdx.x;
  unsigned int gridSize = blockSize * 2 * gridDim.x;

  sdata[tid] = 100;

  while (i < chunkSize) {
    if(sdata[tid] > g_idata[i]){
      sdata[tid] = g_idata[i];
    }
    if(sdata[tid] > g_idata[i+blockSize]){
      sdata[tid] = g_idata[i+blockSize];
    }
    i += gridSize;
  }
  __syncthreads();

  if(blockSize >= 1024){
    if(tid < 512){
      if(sdata[tid] > sdata[tid + 512]){
          sdata[tid] = sdata[tid + 512];
      }
    }
    __syncthreads();
  }

  if(blockSize >= 512){
    if(tid < 256){
      if(sdata[tid] > sdata[tid + 256]){
          sdata[tid] = sdata[tid + 256];
      }
    }
    __syncthreads();
  }

  if(blockSize >= 256){
    if(tid < 128){
      if(sdata[tid] > sdata[tid + 128]){
          sdata[tid] = sdata[tid + 128];
      }
    }
    __syncthreads();
  }

  if(blockSize >= 128){
    if(tid < 64){
      if(sdata[tid] > sdata[tid + 64]){
          sdata[tid] = sdata[tid + 64];
      }
    }
    __syncthreads();
  }
```

```
  if (tid < 32) {
    warpReduce<blockSize>(sdata, tid);
  }
  //write result for this block to global mem
  if(tid == 0){
    g_odata[blockIdx.x] = sdata[0];
  }
}
```

Κώδικας 17: SIMT Min

Σε επίπεδο warp κάνουμε reduce με τον παρακάτω κώδικα:

```
template<unsigned int blockSize>
__device__ void warpReduce(volatile int8_t* sdata, int tid){
  if(blockSize >= 64){
    if (sdata[tid] > sdata[tid+32]) {
      sdata[tid] = sdata[tid+32];
    }
  }

  if(blockSize >= 32){
    if (sdata[tid] > sdata[tid+16]) {
      sdata[tid] = sdata[tid+16];
    }
  }

  if(blockSize >= 16){
    if (sdata[tid] > sdata[tid+8]) {
      sdata[tid] = sdata[tid+8];
    }
  }

  if(blockSize >= 8){
    if (sdata[tid] > sdata[tid+4]) {
      sdata[tid] = sdata[tid+4];
    }
  }

  if(blockSize >= 4){
    if (sdata[tid] > sdata[tid+2]) {
      sdata[tid] = sdata[tid+2];
    }
  }

  if(blockSize >= 2){
    if (sdata[tid] > sdata[tid+1]) {
      sdata[tid] = sdata[tid+1];
    }
  }
}
```

Κώδικας 18: SIMT Min Warp reduce

2.4.5 Aggregation functions – MAX

```
template<unsigned int blockSize>
__global__ void reduce_max_kernel(int8_t *g_idata, int8_t *g_odata, int chunkSize){
  extern __shared__ int sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * (blockSize * 2) + threadIdx.x;
  unsigned int gridSize = blockSize * 2 * gridDim.x;

  sdata[tid] = 0;

  while (i < chunkSize) {
    if(sdata[tid] < g_idata[i]){
      sdata[tid] = g_idata[i];
    }
    if(sdata[tid] < g_idata[i+blockSize]){
      sdata[tid] = g_idata[i+blockSize];
    }
    i += gridSize;
  }
  __syncthreads();

  if(blockSize >= 1024){
    if(tid < 512){
      if(sdata[tid] < sdata[tid + 512]){
          sdata[tid] = sdata[tid + 512];
      }
    }
    __syncthreads();
  }

  if(blockSize >= 512){
    if(tid < 256){
      if(sdata[tid] < sdata[tid + 256]){
          sdata[tid] = sdata[tid + 256];
      }
    }
    __syncthreads();
  }

  if(blockSize >= 256){
    if(tid < 128){
      if(sdata[tid] < sdata[tid + 128]){
          sdata[tid] = sdata[tid + 128];
      }
    }
    __syncthreads();
  }

  if(blockSize >= 128){
    if(tid < 64){
      if(sdata[tid] < sdata[tid + 64]){
          sdata[tid] = sdata[tid + 64];
      }
    }
    __syncthreads();
  }
```

```
  if (tid < 32) {
    warpReduce<blockSize>(sdata, tid);
  }
  //write result for this block to global mem
  if(tid == 0){
    g_odata[blockIdx.x] = sdata[0];
  }
}
```

Κώδικας 19: SIMT Max

Σε επίπεδο warp κάνουμε reduce με τον παρακάτω κώδικα:

```
template<unsigned int blockSize>
__device__ void warpReduce(volatile int8_t* sdata, int tid){
  if(blockSize >= 64){
    if (sdata[tid] > sdata[tid+32]) {
      sdata[tid] = sdata[tid+32];
    }
  }

  if(blockSize >= 32){
    if (sdata[tid] > sdata[tid+16]) {
      sdata[tid] = sdata[tid+16];
    }
  }

  if(blockSize >= 16){
    if (sdata[tid] > sdata[tid+8]) {
      sdata[tid] = sdata[tid+8];
    }
  }

  if(blockSize >= 8){
    if (sdata[tid] > sdata[tid+4]) {
      sdata[tid] = sdata[tid+4];
    }
  }

  if(blockSize >= 4){
    if (sdata[tid] > sdata[tid+2]) {
      sdata[tid] = sdata[tid+2];
    }
  }

  if(blockSize >= 2){
    if (sdata[tid] > sdata[tid+1]) {
      sdata[tid] = sdata[tid+1];
    }
  }
}
```

Κώδικας 20: SIMT Max Warp reduce

2.5 Διαθέσιμο Hardware

Το μηχάνημα που χρησιμοποιούμε για να κάνουμε τις προσομοιώσεις αποτελείται από έναν επεξεργαστή Intel Core i7-4702MQ, μνήμη RAM μεγέθους 16Gb και την GeForce GT750M καρτα γραφικών. Πιο συγκεκριμένα, τα χαρακτηριστικά του επεξεργαστή και της κάρτας γραφικών συνοψίζονται στους δύο παρακάτω πίνακες.

Επεξεργαστής Intel Core i7-4702MQ:

Product Collection	4th Generation Intel Core i7 Processors
Code Name	Products formerly Haswell
Processor Number	i7-4702MQ
Launch Date	Q2'13
Lithography	22 nm
# of Cores	4
# of Threads	8
Processor Base Frequency	2.20 GHz
Max Turbo Frequency	3.20 GHz
Cache	6 MB SmartCache
Instruction Set	64-bit
Instruction Set Extensions	Intel SSE4.1, Intel SSE4.2, Intel AVX2

Κάρτα γραφικών GeForce GT 750M:

Total amount of global memory	4040 MBytes (4235788288 bytes)
Multiprocessors	2
CUDA Cores/MP	192
Total CUDA Cores	384
GPU Max Clock rate	1085 MHz (1.09 GHz)
Memory Clock rate	900 Mhz
Memory Bus Width	128-bit
L2 Cache Size	262144 bytes
Total amount of shared memory per block	49152 bytes
Total number of registers available per block	65536
Warp size	32
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Concurrent copy and kernel execution:	Yes with 1 copy engine(s)
Architecture	Tesla K10
Compute capability	3.0

Για την υλοποίηση των αλγορίθμων χρησιμοποιήσαμε τις γλώσσες C/C++ και το CUDA 10.1 SDK.

Κεφάλαιο 3 – Αποτελέσματα

3.1 Εισαγωγή

Το δείγμα που θα χρησιμοποιήσουμε ως είσοδο στους αλγορίθμους μας αποτελείται από 2^{30} στοιχεία ακεραίου τύπου με τιμές στο διάστημα [0, 5]. Το δείγμα εισόδου πριν τροφοδοτηθεί στους αλγορίθμους υφίσταται επεξεργασία στη σειρά εμφάνισης των επιμέρους στοιχείων του, ώστε η κατανομή τους να είναι τυχαία. Για να πετύχουμε μία τυχαία κατανομή, κάνουμε χρήση του αλγορίθμου Fisher-Yates shuffle [15]. Η υλοποίησή του φαίνεται παρακάτω:

```
double fisherYatesShuffle(int32_t** relation, int32_t REL_SIZE){

  int32_t temp,random;
  srand(time(NULL));
  clock_t t;
  t = clock();
  for(int i = REL_SIZE-1 ; i > 0 ; i--){
    random = rand() % (i+1); // provides us a random index-number
                             // within the variable range each time
    temp = (*relation)[random];
    (*relation)[random] = (*relation)[i];
    (*relation)[i] = temp;
  }
  t = clock() - t;
  return ((double) t) / CLOCKS_PER_SEC;
}
```

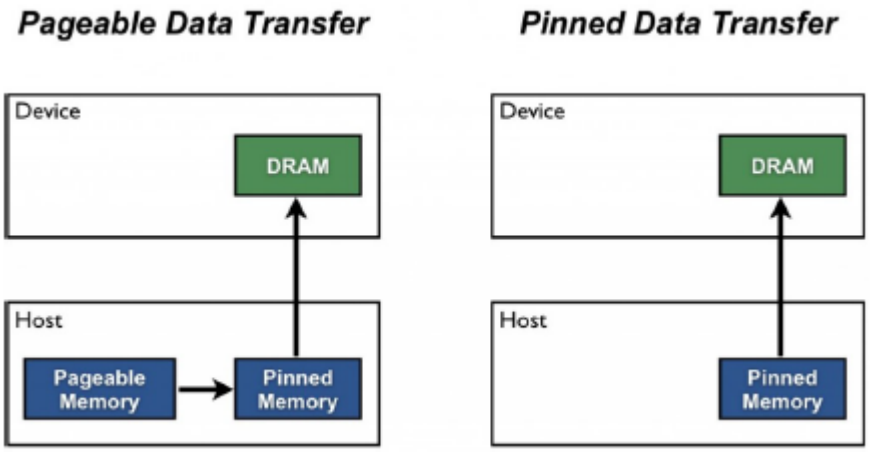
Κώδικας 21: Υλοποίηση αλγορίθμου Fisher-Yates

Για το filtering πρόβλημα που μελετάμε η συχνότητα εμφάνισης της τιμής στόχου θα λαμβάνει διαφορετικές τιμές κάθε φορά. Οι τιμές του selectivity που θα εξετάσουμε ποικίλουν και είναι από 10% εώς και 100% επί του συνολικού δείγματος, με βήμα δέκα ποσοστιαίες μονάδες κάθε φορά, καθώς επίσης και μεμονωμένες τιμές όπως 0.1%, 1% και 25%.

Τα προγράμματα γίνονται compile χρησιμοποιώντας τους gcc και nvcc compilers, για τους κώδικες που αφορούν την CPU και την GPU αντίστοιχα. Τα προγράμματα που αφορούν τις scalar και AVX αρχιτεκτονικές γίνονται compile με δύο τρόπους. Ο πρώτος τρόπος είναι ο default τρόπος, χωρίς δηλαδή να εφαρμόσει κάποιο optimization ο gcc compiler ενώ κατά τον δεύτερο τρόπο κάνουμε activate το O3 flag του compiler και αυτος εφαρμόζει όλα τα δυνατά optimizations.

Για το profiling της εκτέλεσης των αλγορίθμων χρησιμοποιήσαμε το Perf Suite του λειτουργικού Linux και τον NVVP profiler της Nvidia.

Λόγω του περιορισμού μνήμης που θέτει το hardware και ειδικότερα η κάρτα γραφικών, τα στοιχεία εισόδου για την SIMT αρχιτεκτονική είναι του τύπου int8_t, έτσι ώστε να μπορούν να χωρέσουν με ένα πέρασμα όλα στην μνήμη της κάρτας γραφικών. Άλλωστε, η είσοδός μας αποτελείται από στοιχεία στο διάστημα [0, 5], όπως έχουμε ήδη αναφέρει. Συνεπώς το μέγεθος της εισόδου που μεταφέρεται στην κάρτα γραφικών είναι της τάξης του 1GB. Για την δέσμευση της μνήμης στο host μηχάνημα υπάρχουνε δύο τύποι μνήμης, η pinned memory και η pageable (non-pinned) memory.



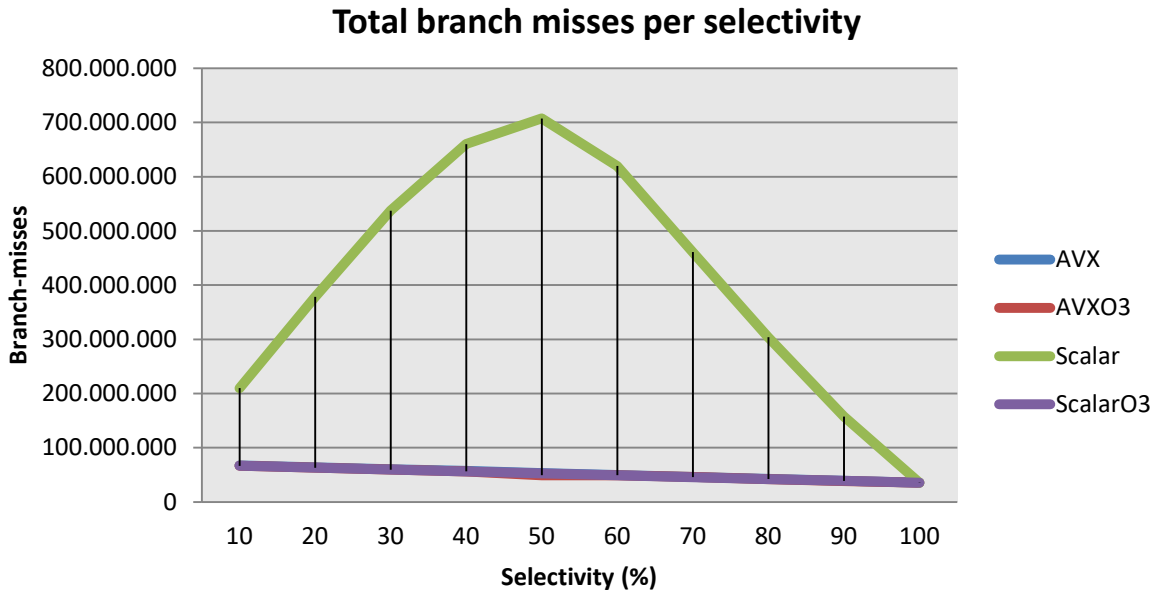
Εικόνα 25: Pageable και Pinned μοντελα μνήμης

Γενικότερα η pinned memory είναι πιο γρήγορη σε ταχύτητα από την pageable memory. Ο CUDA driver ελέγχει εάν το εύρος μνήμης που ζητείται είναι κλειδωμένο ή όχι και αναλόγως χρησιμοποιεί διαφορετικό τρόπο χειρισμού. Χρησιμοποιώντας την pinned memory κρατάμε όλα τα στοιχεία αποθηκευμένα στην RAM, χωρίς να επιτρέπεται page fault με τον δίσκο. Έτσι, επιτρέπουμε την λειτουργία της ασύγχρονης αντιγραφής (σ.σ: Asynchronous copy - DMA). Αντίθετα, η pageable memory μπορεί να γεννήσει page fault, αφου τα στοιχεία δεν αποθηκεύονται μόνο στην RAM αλλά και στον δίσκο. Έτσι ο CUDA driver πρέπει να αντιγράψει την σελίδα που δημιούργησε το page fault στην pinned memory (σ.σ: Synchronous copy). Εν συντομία, χρησιμοποιώντας την pinned memory πετυχαίνουμε καλύτερες ταχύτητες αντιγραφής μέσω του διαύλου PCI, καθώς επίσης και ασύχρονη αντιγραφή από το host στο device (σ.σ: GPU) και αντίστροφα, χωρίς να παγώνει η ροή εκτέλεσης του κύριου προγράμματος (overlapping της διαδικασίας αντιγραφής με κώδικα που τρέχει στο host μηχάνημα). [16]

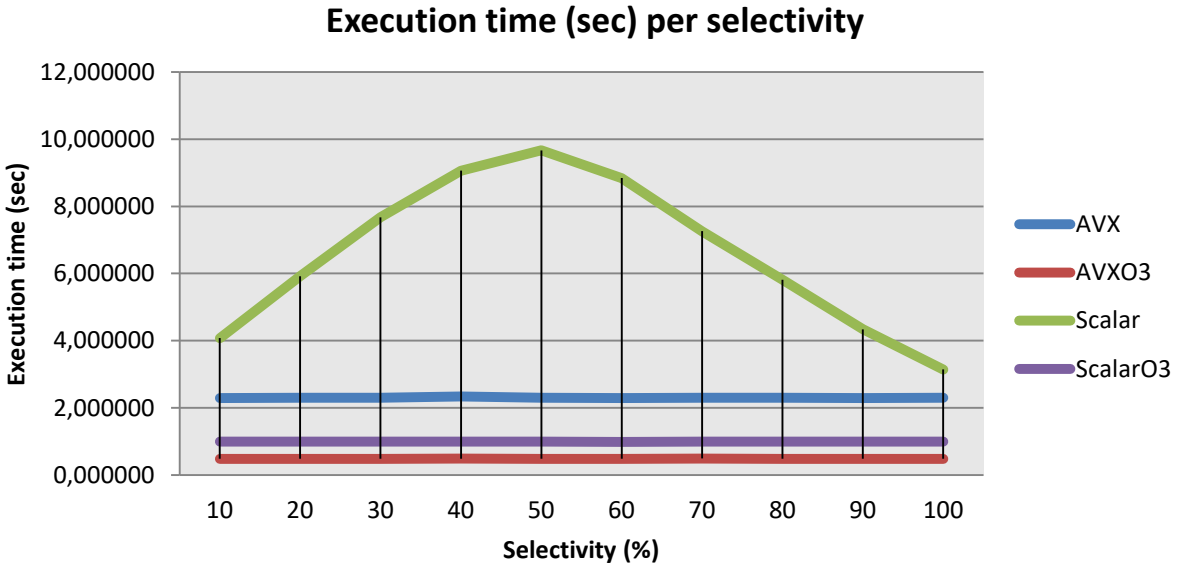
Συμφωνα με μια μελετη του πανεπιστημίου της Virginia [17] η pinned memory επιφέρει καλύτερο speed-up από την non-pinned memory, όταν τα δεδομένα που μεταφέρονται από το host μηχάνημα στο device μηχάνημα ξεπερνούν σε μέγεθος τα 128 MB, για την περίπτωση που τα δεδομένα μεταφέρονται μόνο προς την μία κατεύθυνση. Η μελέτη αυτή αποτέλεσε και τον κινητήριο μοχλό στην επιλογή του κατάλληλου τύπου μνήμης που θα χρησιμοποιήσουμε.

3.2 Filtering: Σύγκριση Scalar – AVX

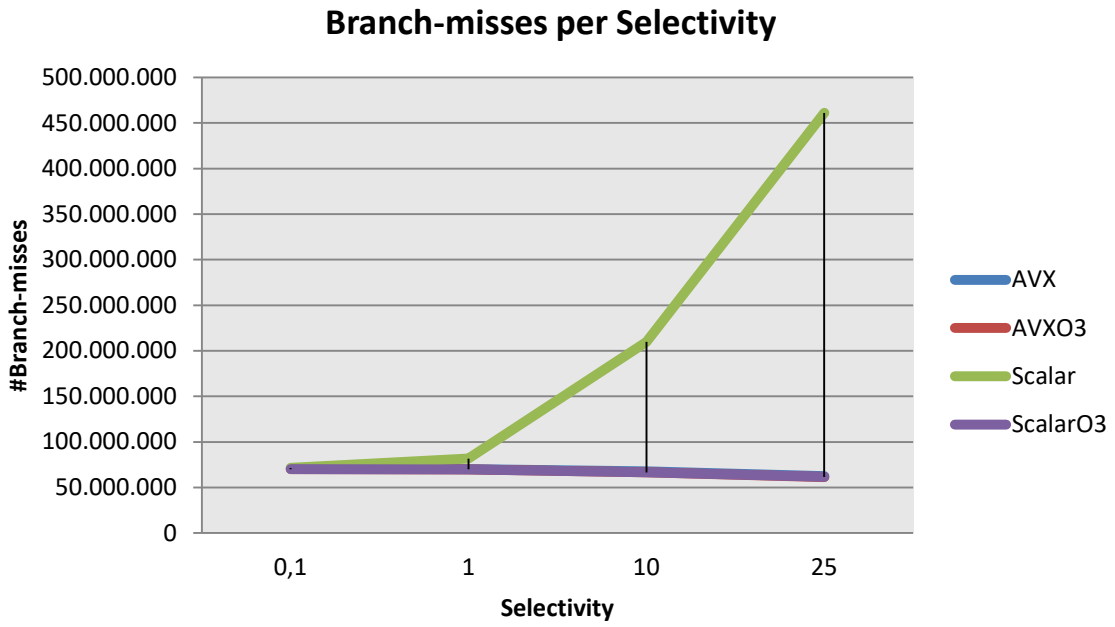
Η σύγκριση των αποτελεσμάτων ξεκινάει με το πρόβλημα του filtering για τις scalar και AVX αρχιτεκτονικές. Χρησιμοποιώντας την σουίτα Perf αντλήσαμε κάποιες μετρικές που θα μας βοηθήσουν για να εξηγήσουμε καλύτερα τα αποτελέσματα. Στο γράφημα 1 φαίνεται ο αριθμός των branch misses για κάθε μία εκτέλεση, για selectivity από 10% εώς 100% με βήμα 10% κάθε φορά. Ο αριθμός των branch misses αφορά την εκτέλεση όλου του προγράμματος, από την δημιουργία της εισόδου και την ανακατανομή των στοιχείων της μέχρι και την εκτέλεση του αλγορίθμου filtering και την παραγωγή του αντίστοιχου bitmap. Αυτό όμως δεν παύει να δείχνει την διαφορά ανάμεσα στην εκτέλεση ενός προγράμματος σε scalar αρχιτεκτονική χωρίς τα optimization του compiler με τις άλλες τρεις περιπτώσεις.



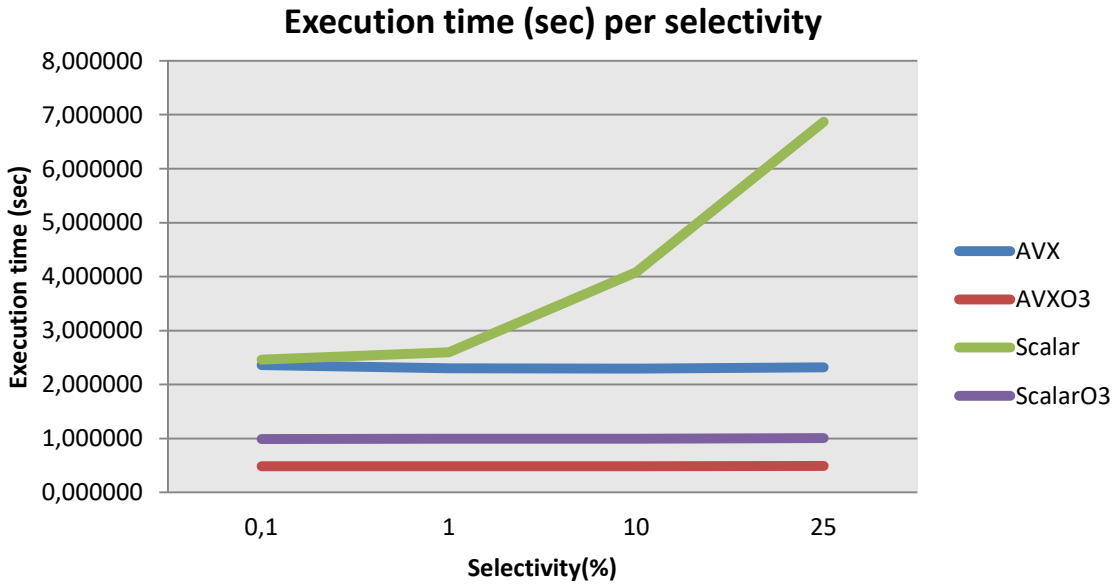
Γράφημα 1: Branch misses per selectivity (I)



Γράφημα 2: Χρόνος εκτέλεσης ανά selectivity(I)



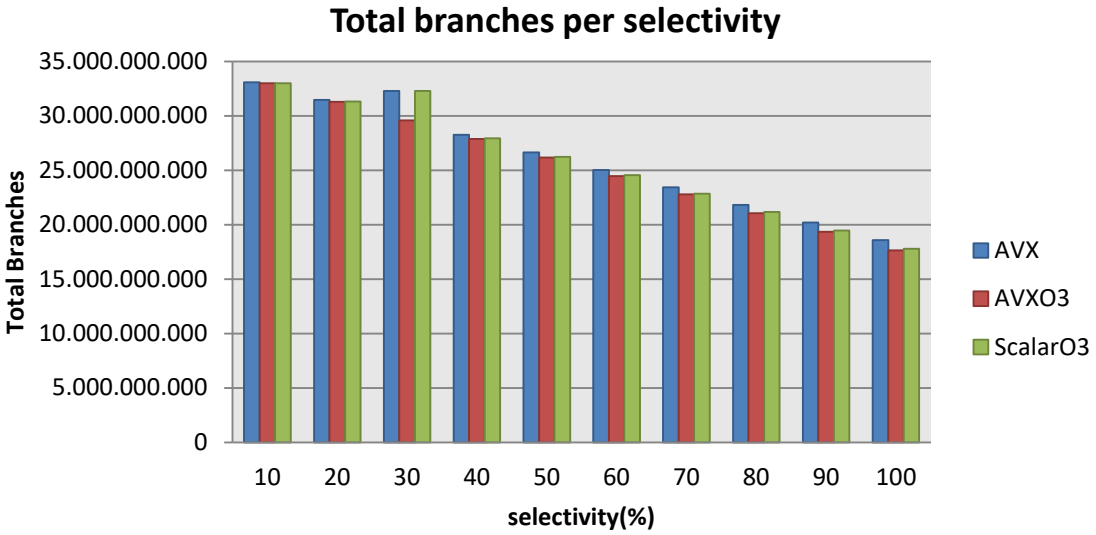
Γράφημα 3: Branch misses per selectivity (II)



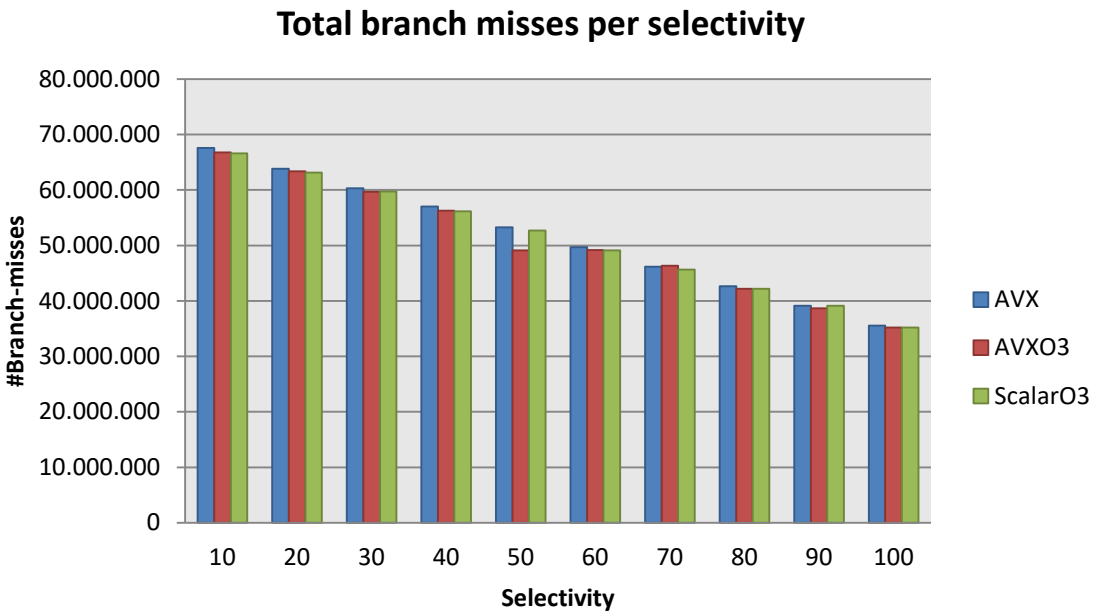
Γράφημα 4: Χρόνος εκτέλεσης ανά selectivity(II)

Παρατηρούμε λοιπόν ότι οι καμπύλες που περιγράφουν το χρόνο εκτέλεσης για την scalar αρχιτεκτονική χωρίς τα optimizations στα γραφήματα 2 και 4 ακολουθούν την ίδια τάση με τις αντίστοιχες καμπύλες των γραφημάτων 1 και 3 αντίστοιχα. Γίνεται λοιπόν αμέσως κατανοητό ότι ο χρόνος εκτέλεσης για την scalar αρχιτεκτονική, χωρίς εφαρμογή κανόνων βελτιστοποίησης από τον compiler, επηρεάζεται από των αριθμό των branch misses.

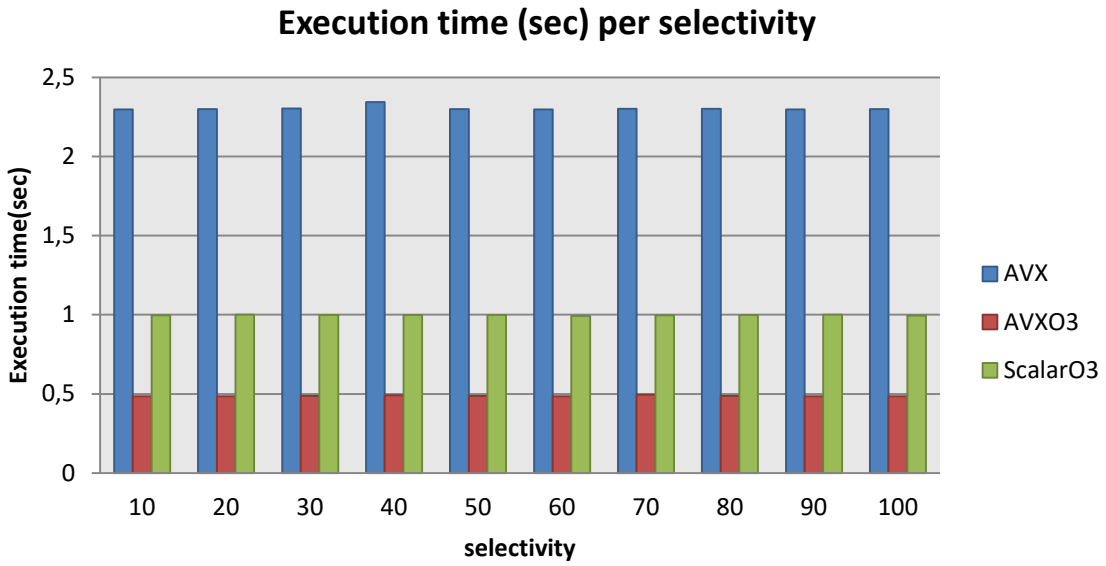
Από τα γραφήματα 1 έως 4 αφαιρέθηκε η είσοδος που αφορά την περίπτωση scalar (χωρίς optimizations) και δημιουργήθηκαν τα γραφήματα 6, 7, 9 και 10. Επίσης προστέθηκαν τα γραφήματα 5 και 8 τα οποία δείχνουνε τον συνολικό αριθμό από branches που συνέβησαν καθόλη την διάρκεια της εκτέλεσης.



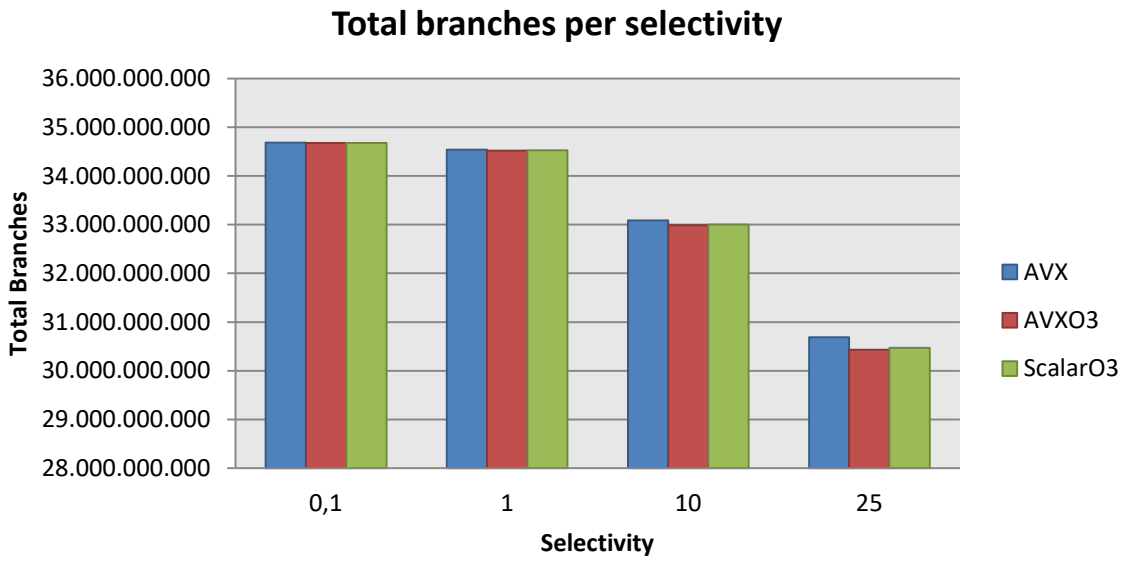
Γράφημα 5: Συνολικός αριθμός branch ανά selectivity (III)



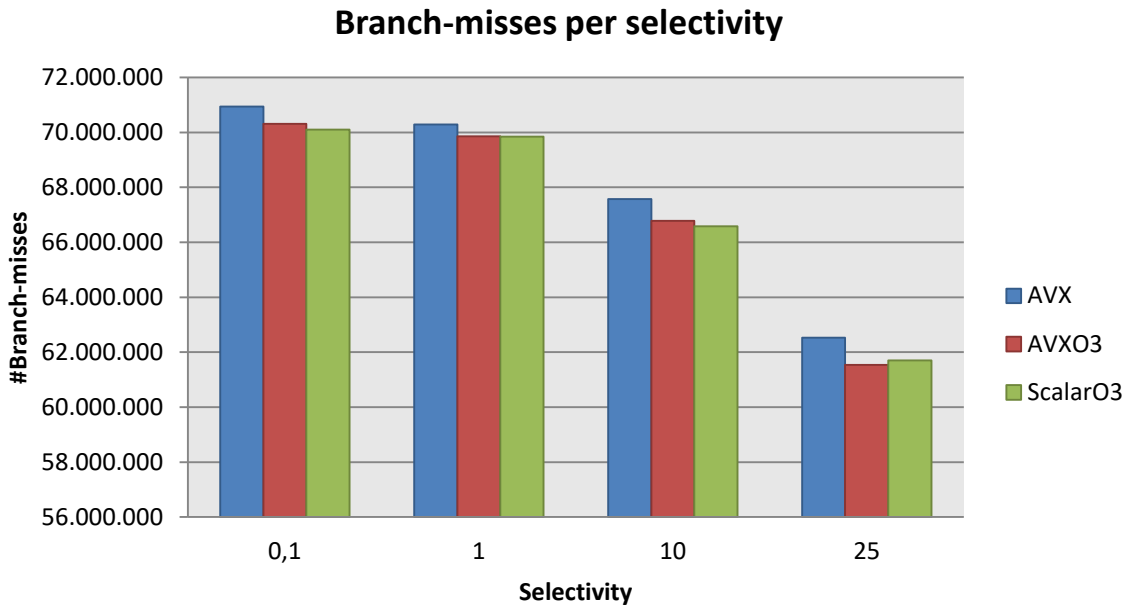
Γράφημα 6: Branch misses ανά selectivity (III)



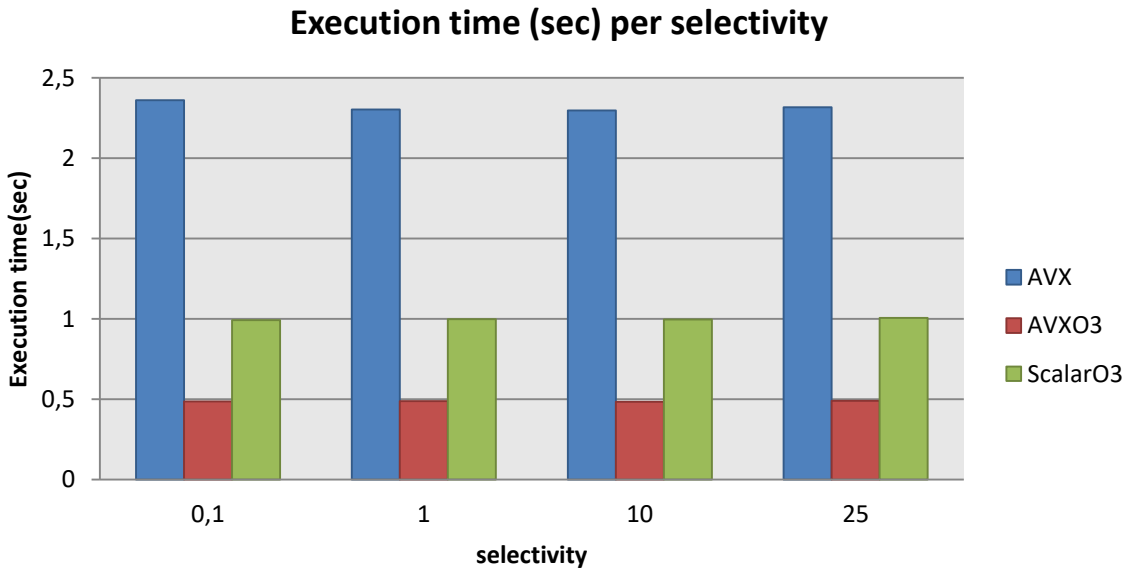
Γράφημα 7: Χρόνος εκτέλεσης ανά selectivity (III)



Γράφημα 8: Συνολικός αριθμός branch ανά selectivity (IV)



Γράφημα 9: Branch misses ανά selectivity (IV)



Γράφημα 10: Χρόνος εκτέλεσης ανά selectivity (IV)

Από τα γραφήματα 5, 6 και 8, 9 φαίνεται ότι ο συνολικός αριθμός των branch misses ακολουθεί την τάση του συνολικού αριθμού των branches, δηλαδή μία φθήνουσα πορεία σε συνάρτηση με την αύξηση του selectivity. Αντίθετα, από τα γραφήματα 7 και 10 φαίνεται ότι ο χρόνος εκτέλεσης παραμένει σταθερός και είναι ανεξάρτητος από την τιμή που θα λάβει κάθε φορά το selectivity.

Γεννάται λοιπόν το ερώτημα αφού δεν επηρεάζεται η εκτέλεση από τα branch mispredictions τότε από που προέρχεται αυτό το speed-up στην απόδοση; Η απάντηση μπορεί να δοθεί εάν κοιτάξουμε στον assembly κώδικα για κάθε μία από τις τρεις πλέον περιπτώσεις. Για την περίπτωση του scalarO3 βλέπουμε ότι χρησιμοποιούνται οι βασικοί

καταχωρητές (registers) της x86 αρχιτεκτονικής καθώς επίσης και η τεχνική του loop unrolling από τον compiler. Για τις περιπτώσεις του AVX, χρησιμοποιούνται τόσο οι XMM καταχωρητές όσο και οι YMM. Υπενθυμίζουμε ότι οι YMM έχουνε το διπλάσιο μέγεθος από τους αντίστοιχους XMM, δηλαδή χωρητικότητα 256 bits. Σε ό,τι αφορά την διαφορά απόδοσης ανάμεσα στην AVX εκτέλεση και την AVXO3 εκτέλεση αυτή οφείλεται στα optimization που εφαρμόζει ο compiler. Γενικότερα αν και με την χρησιμοποίηση καταχωρητών μεγαλύτερων σε μέγεθος μειώνεται προσωρινά το latency, επειδή χρειάζεται περισσότερος χρόνος για να γεμίσει ένας μεγαλύτερος καταχωρητής με στοιχεία από την μνήμη, έχουμε αύξηση του τελικού throughput με αποτέλεσμα την επίτευξη μεγαλύτερου speed-up για την εφαρμογή μας.

Στον πίνακα που ακολουθεί παρατίθεται η σχετική επιτάχυνση που πετυχαίνουμε κάθε φορά:

FILTERING	
ΤΥΠΟΣ ΒΕΛΤΙΩΣΗΣ	**SPEED-UP**
Από scalar σε scalarO3	Από 59% έως 89%
Από scalar σε AVX	Από 4% έως 76%
Από AVX σε AVXO3	78%
Από scalarO3 σε AVXO3	50%

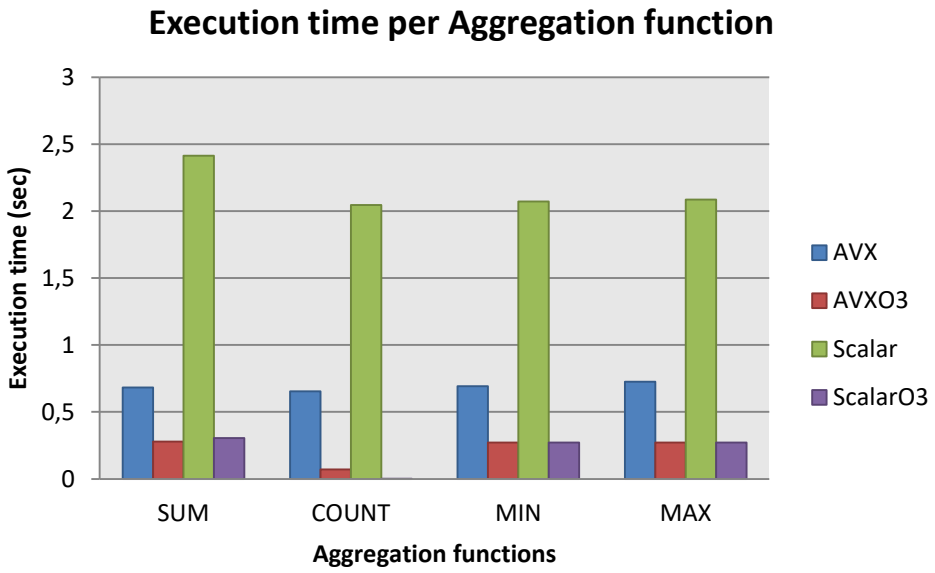
Τέλος, παρατίθενται οι ακριβείς χρόνοι εκτέλεσης, σε δευτερόλεπτα (sec), ανά διαφορετική τιμή selectivity στους παρακάτω δύο πίνακες.

	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
AVX	2,297	2,300	2,304	2,345	2,299	2,298	2,301	2,301	2,297	2,299
AVXO3	0,484	0,484	0,486	0,490	0,487	0,484	0,494	0,486	0,484	0,485
Scalar	4,078	5,921	7,675	9,058	9,66	8,839	7,257	5,815	4,338	3,146
ScalarO3	0,996	1,001	0,998	0,998	0,999	0,992	0,996	0,999	1,000	0,995

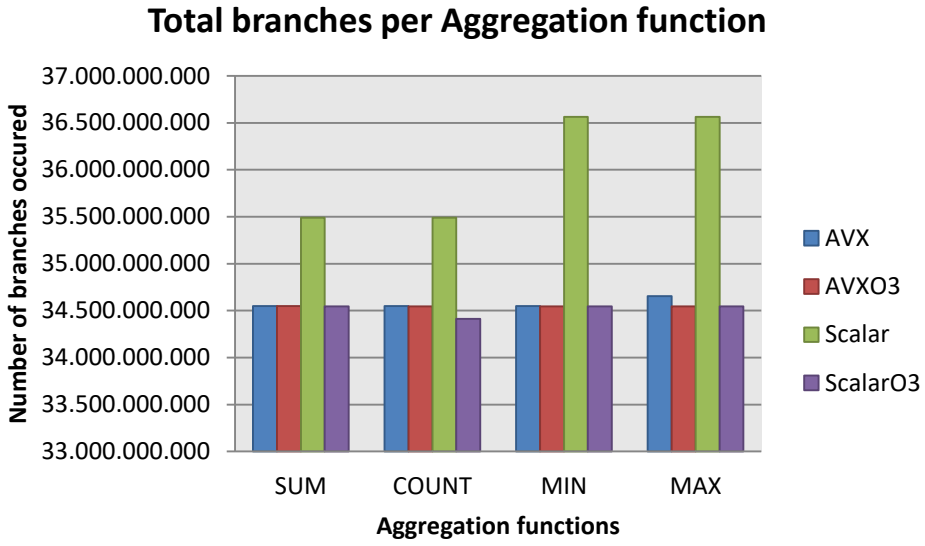
	0,1%	1%	10%	25%
AVX	2,360	2,301	2,297	2,316
AVXO3	0,485	0,486	0,484	0,488
Scalar	2,459	2,598	4,078	6,870
ScalarO3	0,992	0,996	0,996	1,006

3.3 Aggregation functions: Σύγκριση Scalar – AVX

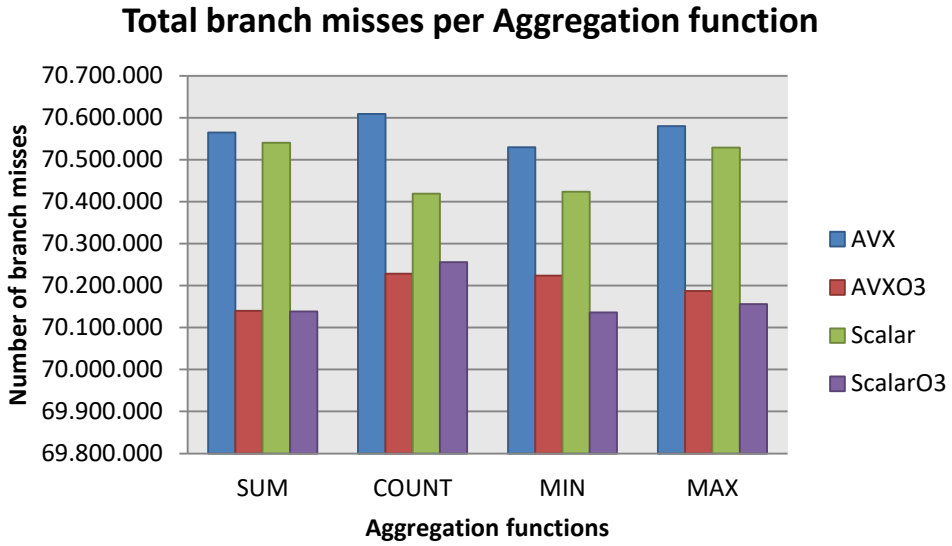
Η ανάλυση συνεχίζεται για τα aggregation functions που αφορούν τις αρχιτεκτονικές scalar και AVX. Στο γράφημα 11 παρουσιάζονται οι χρόνοι εκτέλεσης, με μονάδα μέτρησης το ένα δευτερόλεπτο (sec). Γίνεται αμέσως αντιληπτό ότι η απλή περίπτωση της scalar αρχιτεκτονικής επιφέρει και την χειρότερη απόδοση. Αντίθετα βλέπουμε ότι για τις optimized περιπτώσεις (σ.σ: AVXO3 και scalarO3) οι χρόνοι εκτέλεσης είναι πανομοιότυποι.



Γράφημα 11: Χρόνοι εκτέλεσης ανά aggregation function



Γράφημα 12: Συνολικός αριθμός branch ανά aggregation function



Γράφημα 13: Συνολικός αριθμός branch misses ανά aggregation function

Στα γραφήματα 12 και 13 φαίνεται ο συνολικός αριθμός από branches και branch misses αντίστοιχα. Υπενθυμίζουμε ότι τα νούμερα που παρουσιάζονται σε αυτά τα δύο γραφήματα αφορούν την συνολική εκτέλεση του προγράμματος. Παρόλα αυτά είναι ευκρινής η διαφορά στον αριθμό των branches που συνέβησαν στην απλή περίπτωση της scalar αρχιτεκτονικής για τα Min και Max aggregation functions, σε αντίθεση με τον αντίστοιχο αριθμό για τα Sum και Count aggregations.

Η διαφορά αυτή είναι αναμενώμενη, αφού εάν κοιτάξει κανείς τους κώδικες 5 και 6 θα παρατηρήσει ότι ότι έχουμε και ένα επιπλέον branch για κάθε ένα στοιχείο του δείγματος. Υπενθυμίζουμε ότι η είσοδός μας έχει 2^{30} , ή ισοδύναμα 1.073.741.824, στοιχεία όση δηλαδή είναι και η διαφορά για την οποία κάνουμε λόγο.

Πριν προχωρήσουμε σε περαιτέρω ανάλυση, πρέπει να σημειώσουμε ότι η μετρική του branch predicting δεν έχει νόημα για τα Sum και Count aggregations, μιας και από τους κώδικες 3 και 4 φαίνεται ότι σε αυτές τις συναρτήσεις δεν υπάρχει κάποια εντολή διακλάδωσης. Για αυτά τα δύο aggregation functions, παρατηρούμε ότι η περίπτωση της απλής scalar εκτέλεσης δημιουργεί περίπου ένα εκατομμύριο περισσότερα branches. Δεν πρέπει να ξεχνάμε ότι χρησιμοποιώντας τις SIMD εντολές πετυχαίνουμε παραλληλία τάξης οκτώ, αφού επεξεργαζόμαστε ταυτόχρονα οκτώ ακέραιους αριθμούς. Κοιτάζοντας το loop για την AVX αρχιτεκτονική, βλέπουμε ότι η συνθήκη ελέγχου θα αποτιμηθεί οκτώ φορές λιγότερες, άρα αντί για 2^{30} αποτιμήσεις της συνθήκης ελέγχου θα γίνουν 2^{27} αποτιμήσεις (σ.σ:134.217.728 αποτιμήσεις). Αυτό σημαίνει ότι οι υπόλοιπες 2^{30} - 2^{27} αποτιμήσεις δεν θα συμβούνε ποτέ (σ.σ: ο ακριβής αριθμός είναι 939.524.096). Αυτή είναι και η εξήγηση γιατί βλέπουμε μικρότερο αριθμό από branches για τις AVX εκτελέσεις.

Συνεχίζουμε την ανάλυση με το κομμάτι των branch misses. Ο αριθμός των branch misses για την απλή scalar περίπτωση είναι συγκρίσιμος με τον αριθμό των branch misses για την απλή AVX περίπτωση. Αυτό εξηγείται από το γεγονός ότι τα branch misses αφορούν κομμάτια κώδικα πέρα από την προς εξέταση συνάρτηση. Ο ισχυρισμός αυτός ενισχύεται από το γεγονός ότι τα branch misses για όλες τις optimized περιπτώσεις είναι συγκριτικά μικρότερος. Αυτό οφείλεται στο γεγονός ότι με την ενεργοποίηση του O3 flag ο

compiler βελτιστοποιεί και τις συναρτήσης παραγωγής και ανακατανομής των στοιχείων εισόδου. Το ότι βλέπουμε δηλαδή μικρότερο αριθμό branch misses οφείλεται στην βελτιστοποίηση άλλων τμημάτων κώδικα. Άλλωστε για τα Sum και Count aggregations δεν έχουμε branches. Όσον αφορά τα branch misses για τα Min και Max aggregation functions, λόγω του μικρού εύρους τιμών που έχουμε ως είσοδο, ο branch predictor μπορεί εύκολα μετά από ένα συγκεκριμένο αριθμό επαναλήψεων να προβλέψει σωστά την έκβαση του branch πετυχαίνοντας έτσι μεγάλο αριθμό από branch hits.

Παρατηρώντας τους χρόνους εκτέλεσης και σε συνδυασμό με τα όσα αναφέρθηκαν περί branches και branch misses, εύκολα καταλαβαίνει κανείς ότι η απάντηση στην ερώτηση από που προέρχεται τελικά αυτό το speed up δεν μπορεί να βασιστεί αποκλειστικά και μόνο σε αυτές τις δύο μετρικές. Αναζητήσαμε λοιπόν αυτά τα αίτια στον παραχθέντα assembly κώδικα για κάθε μία περίπτωση.

Για την απλή περίπτωση του scalar Count το κύριο κομμάτι υπολογισμού σε assembly φαίνεται στην παρακάτω εικόνα. Χρησιμοποιούνται οι βασικοί καταχωρητές και η μεταβολή της τιμής τους γίνεται απευθείας με αναφορά σε διεύθυνση μνήμης. Για την αντίστοιχη περίπτωση όπου εφαρμόζουμε το O3 flag, ο κώδικας assembly φαίνεται στην εικόνα 27. Εκεί ο compiler κάνει optimize το loop, καταλαβαίνοντας ότι το τελικό αποτέλεσμα του loop ισούται και με το εύρος του, εξού και ο χρόνος εκτέλεσης.

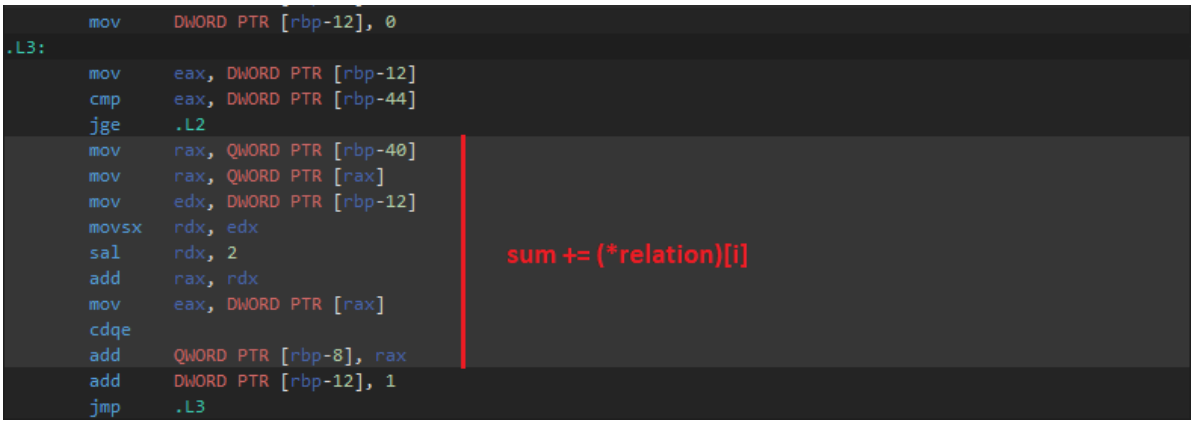
```
        mov     DWORD PTR [rbp-12], 0
.L3:
        mov     eax, DWORD PTR [rbp-12]
        cmp     eax, DWORD PTR [rbp-44]
        jge     .L2
        add     QWORD PTR [rbp-8], 1      | count++;
        add     DWORD PTR [rbp-12], 1
        jmp     .L3
```

Εικόνα 26: Assembly scalar Count - O0

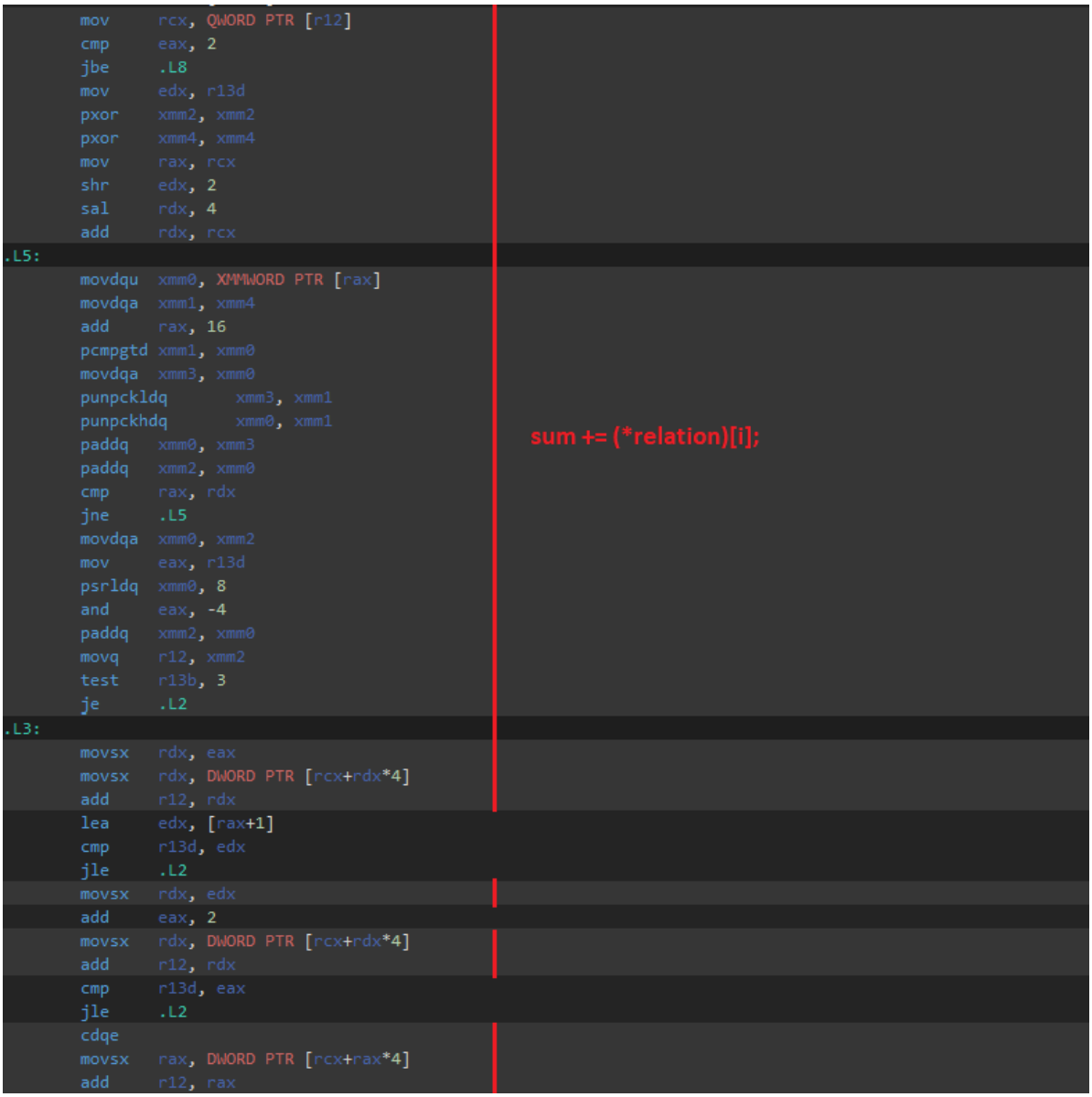
```
        lea     r13d, [rbx-1]
        add     r13, 1
```

Εικόνα 27: Assembly scalar Count - O3

Στην περίπτωση του scalar Sum, ο παραχθέν κώδικας assembly για την O0 (no optimizations) περίπτωση φαίνεται στην εικόνα 28. Και σε αυτήν την περίπτωση χρησιμοποιούνται οι βασικοί καταχωρητές. Στην εικόνα 29 φαίνεται ο assembly κώδικας για την περίπτωση που ενεργοποιούμε το O3 optimization flag. Παρατηρούμε ότι παράγεται πιο εκτεταμένος κώδικας assembly, αλλά σε αυτήν την περίπτωση ο compiler χρησιμοποιεί τεχνικές vectorization, αφού χρησιμοποιούνται οι XMM καταχωρητές.



Εικόνα 28: Assembly scalar Sum - O0

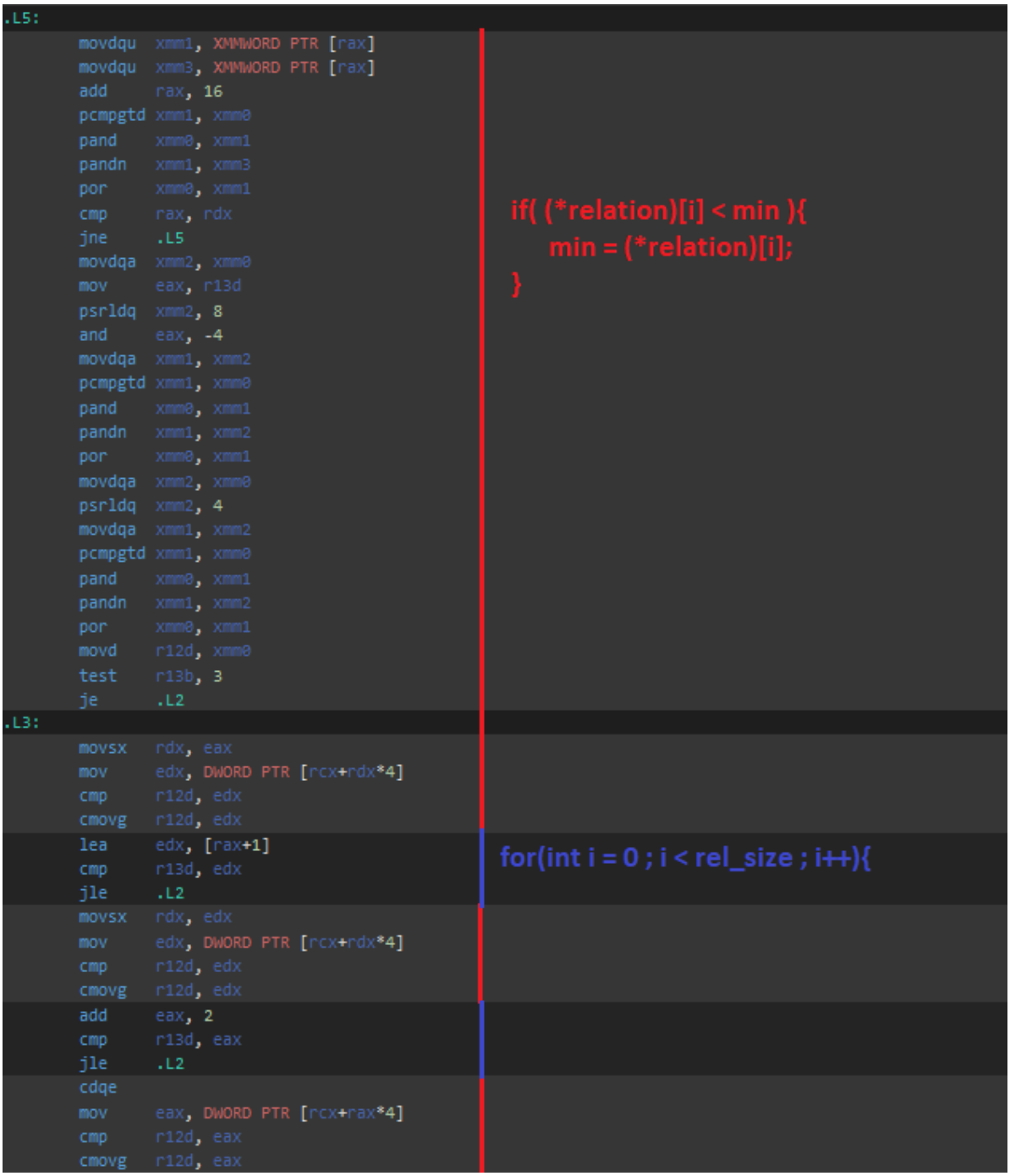


Εικόνα 29: Assembly scalar Sum - O3

Εικόνα 30: Assembly scalar Min - O0

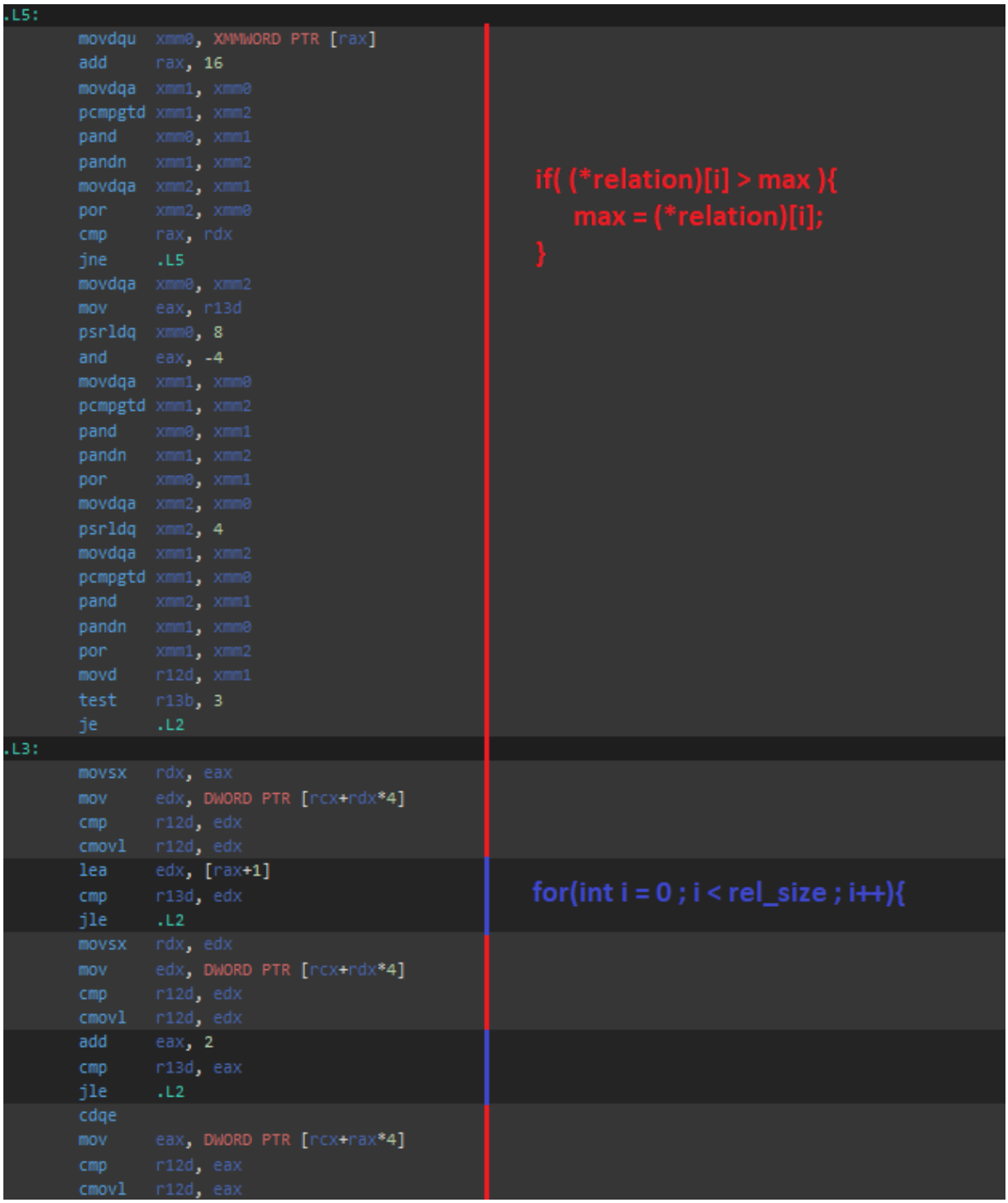
Ομοίως και για την περίπτωση του scalar Min – O0, χρησιμοποιούνται οι βασικοί καταχωρητές, ενώ όταν ενεργοποιείται το O3 flag ο assembly κώδικας γίνεται πιο εκτεταμένος αλλά ο compiler κάνει χρήση του auto-vectorization και φαίνεται ότι στο κρίσιμο κομμάτι υπολογισμού χρησιμοποιούνται οι YMM καταχωρητές. Οι παρατηρήσεις αυτές προέρχονται από τις εικόνες 30 και 31 αντίστοιχα.

Οι ίδιες ακριβώς παρατηρήσεις ισχύουν και για το δυϊκώς αντίστροφο πρόβλημα, αυτό του Max, όπως φαίνεται και στις εικόνες 32 και 33.



Εικόνα 31: Assembly scalar Min - O3

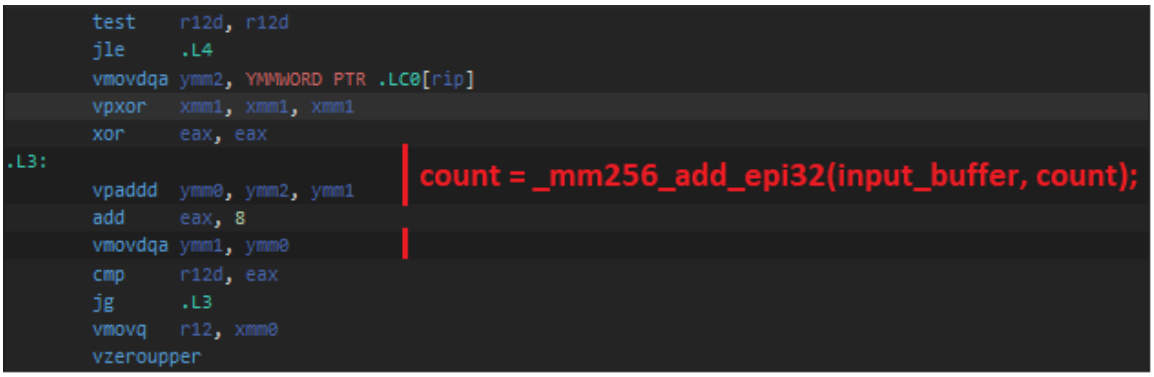
Εικόνα 32: Assembly scalar Max - O0



Εικόνα 33: Assembly scalar Max - O3

Εξετάζοντας τους αντίστοιχους κώδικες για την περίπτωση του AVX, λαμβάνουμε τα κάτωθι αποτελέσματα.

Εικόνα 34: Assembly AVX Count - O0



Εικόνα 35: Assembly AVX Count - O3

Εικόνα 36: Assembly AVX Sum - O0

```
        test    r14d, r14d
        jle     .L4
        lea     edx, [r14-1]
        mov     rcx, QWORD PTR [r12]
        xor     eax, eax
        vpxor   xmm1, xmm1, xmm1
        shr     edx, 3
        inc     rdx
        sal     rdx, 5
.L3:
        vmovntdqa       ymm0, YMMWORD PTR [rcx+rax]
        add     rax, 32
        vpaddd  ymm0, ymm0, ymm1
        vmovdqa ymm1, ymm0
        cmp     rdx, rax
        jne     .L3
        vmovq   r12, xmm0
        vzeroupper
```

```
for(int i = 0 ; i < rel_size ; i += 8 ){

sum = _mm256_add_epi32(input_buffer,
sum);
```

Εικόνα 37: Assembly AVX Sum - O3

```
.L6:
        mov     eax, DWORD PTR [rsp+284]
        cmp     eax, DWORD PTR [rsp+20]
        jge     .L3
        mov     rax, QWORD PTR [rsp+24]
        mov     rax, QWORD PTR [rax]
        mov     edx, DWORD PTR [rsp+284]
        movsx   rdx, edx
        sal     rdx, 2
        add     rax, rdx
        mov     QWORD PTR [rsp+88], rax
        mov     rax, QWORD PTR [rsp+88]
        vmovdqa ymm0, YMMWORD PTR [rax]
        vmovdqa YMMWORD PTR [rsp+224], ymm0
        vmovdqa ymm0, YMMWORD PTR [rsp+32]
        vmovdqa YMMWORD PTR [rsp+128], ymm0
        vmovdqa ymm0, YMMWORD PTR [rsp+224]
        vmovdqa YMMWORD PTR [rsp+96], ymm0
        vmovdqa ymm0, YMMWORD PTR [rsp+96]
        vmovdqa ymm1, YMMWORD PTR [rsp+128]
        vpminsd ymm0, ymm1, ymm0
        nop
        vmovdqa YMMWORD PTR [rsp+32], ymm0
        add     DWORD PTR [rsp+284], 8
        jmp     .L6
```

```
for(int i = 0 ; i < rel_size ; i += 8) {

_mm256_load_si256((__m256i*)
                  &(*relation)[i]);

min = _mm256_min_epi32(min,
                       input_buffer);
```

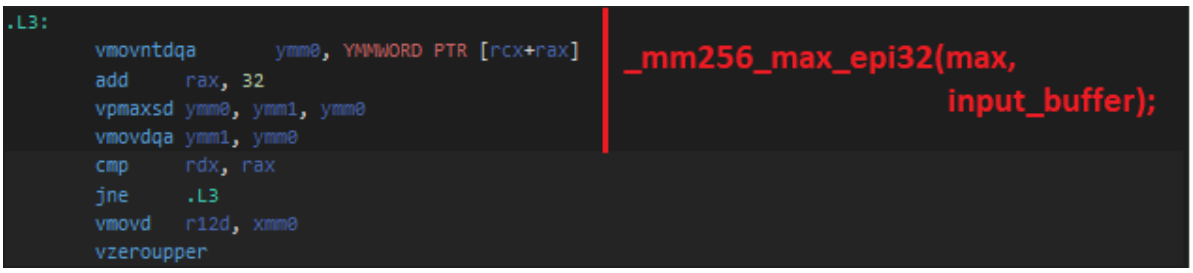
Εικόνα 38: Assembly AVX Min - O0

```
.L9:
        add     rax, 32
.L3:
        vpminsd ymm0, ymm1, YMMWORD PTR [rcx]
        mov     rcx, rax
        vmovdqa ymm1, ymm0
        cmp     rdx, rax
        jne     .L9
        vmovd   r14d, xmm0
        vzeroupper
```

```
min = _mm256_min_epi32(min,
                       input_buffer);
```

Εικόνα 39: Assembly AVX Min - O3

Εικόνα 40: Assembly AVX Max - O0



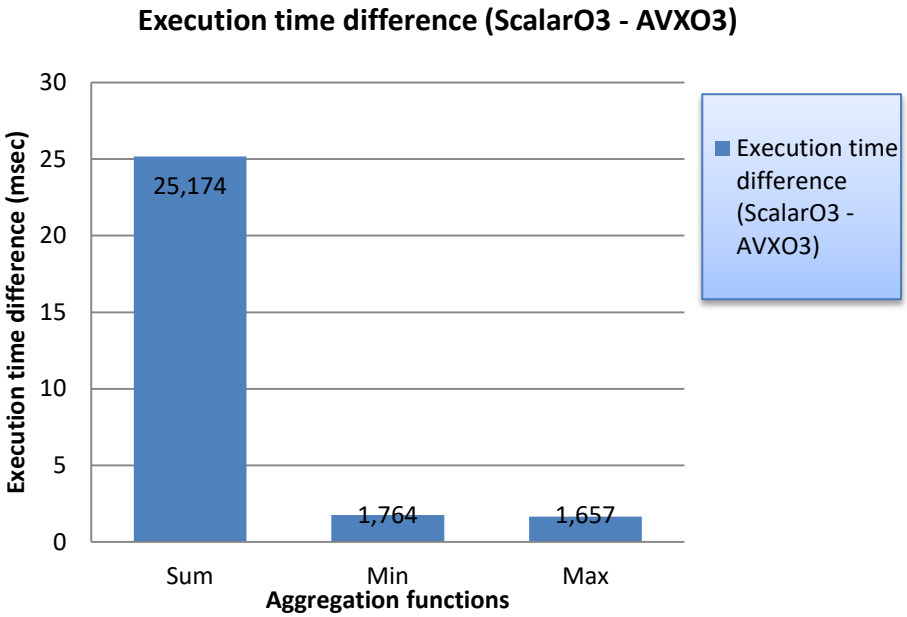
Εικόνα 41: Assembly AVX Max - O3

Παρατηρούμε ότι με την ενεργοποίηση του O3 flag, ο παραγόμενος κώδικας assembly είναι πιο συμπτηγμένος, πολλές εντολές παρακάμπτονται και έτσι εξοικονομούνται αρκετοί κύκλοι ρολογιού. Για την περίπτωση του AVX και για όλα τα Aggregation functions, με ή χωρίς το optimization flag, χρησιμοποιούνται οι YMM καταχωρητές.

Συγκεντρωτικά, οι διαφορές που παρατηρούνται ανάμεσα στην scalar και την AVX περίπτωση, για κάθε ένα από τα aggregation functions είναι οι εξής:

- **Count:** Δεν μπορεί να γίνει σύγκριση δεδομένου του optimization του compiler για την ScalarO3 περίπτωση.
- **Sum:** Η assembly στην περίπτωση του AVXO3 είναι συμπυκνωμένη και γίνεται χρήση των YMM καταχωρητών, ενώ στην ScalarO3 περίπτωση χρειάζονται περισσότερες εντολές για το ίδιο αποτέλεσμα κάνοντας χρήση των XMM καταχωρητών.
- **Min:** Συμπυκνωμένος κώδικας assembly για την AVXO3 περίπτωση, με ταυτόχρονη χρήση των YMM καταχωρητών. Αντίθετα, στην ScalarO3 περίπτωση χρησιμοποιούνται περισσότερες εντολές αλλά και οι XMM καταχωρητές.
- **Max:** Παραμοίως με την περίπτωση του Min.

Στο γράφημα 14 παρουσιάζεται η διαφορά στο χρόνο εκτέλεσης ανάμεσα στις ScalarO3 και AVXO3 υλοποιήσεις. Η διαφορά χρόνου μετράται στην κλίμακα των milliseconds.



Γράφημα 14: Διαφορά στο χρόνο εκτέλεσης ανάμεσα στις ScalarO3 και AVXO3 υλοποιήσεις

Στους πίνακες που ακολουθούν παρατίθεται η σχετική επιτάχυνση (speed-up) που επιτυγχάνεται κάθε φορά για κάθε ένα aggregation function:

COUNT	
ΤΥΠΟΣ ΒΕΛΤΙΩΣΗΣ	**SPEED-UP**
Από scalar σε scalarO3	Δεν ορίζεται
Από scalar σε AVX	68%
Από AVX σε AVXO3	89%
Από scalarO3 σε AVXO3	Δεν ορίζεται

SUM	
ΤΥΠΟΣ ΒΕΛΤΙΩΣΗΣ	**SPEED-UP**
Από scalar σε scalarO3	87%
Από scalar σε AVX	71.7%
Από AVX σε AVXO3	59%
Από scalarO3 σε AVXO3	8.2%

MIN	
ΤΥΠΟΣ ΒΕΛΤΙΩΣΗΣ	**SPEED-UP**
Από scalar σε scalarO3	87%
Από scalar σε AVX	66.6%
Από AVX σε AVXO3	61%
Από scalarO3 σε AVXO3	0.61%

MAX	
ΤΥΠΟΣ ΒΕΛΤΙΩΣΗΣ	**SPEED-UP**
Από scalar σε scalarO3	87%
Από scalar σε AVX	65%
Από AVX σε AVXO3	62.6%
Από scalarO3 σε AVXO3	0.61%

Τέλος, οι ακριβείς χρόνοι εκτέλεσης για όλες τις περιπτώσεις μετρούμενοι στην κλίμακα τους ενός δευτερολέπτου συνοψίζονται στον πίνακα που ακολουθεί.

	SUM	COUNT	MIN	MAX
AVX	0,682103	0,654899	0,691875	0,724926
AVXO3	0,279595	0,071223	0,270542	0,270761
Scalar	2,414236	2,046196	2,071976	2,085844
ScalarO3	0,304769	0,000100	0,272306	0,272418

3.4 Filtering: Σύγκριση SIMT – AVX

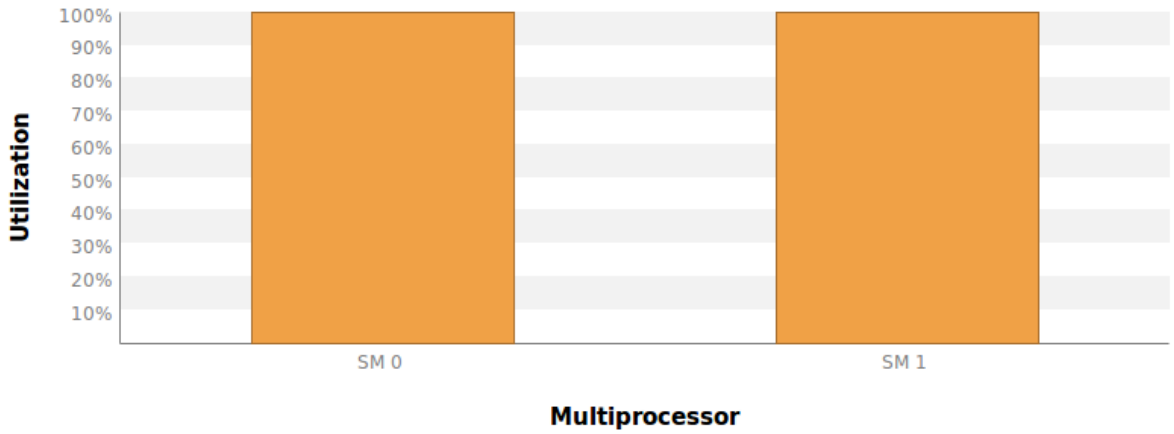
Η μέχρι στιγμής σύγκριση ανάμεσα στην scalar αρχιτεκτονική και την AVX αρχιτεκτονική έχει αναδείξει ως νικήτρια την δεύτερη. Ιδιαίτερα για το πρόβλημα filtering η AVX αρχιτεκτονική επιτυγχάνει απόδοση κατά 50% καλύτερη σε σχέση με την scalar. Η σύγκριση λοιπόν που θα ακολουθήσει αφορά τις AVX και SIMT αρχιτεκτονικές. Στην εικόνα 42 παρουσιάζεται η εκτέλεση του kernel για το filtering πρόβλημα.

Αρχικά στο στάδιο *cudaMemcpy* μεταφέρονται τα δεδομένα από την μνήμη του host μηχανήματος στην μνήμη του device μηχανήματος. Βλέπουμε ότι έχουμε δύο κλήσεις του συγκεκριμένου CUDA API. Αυτό οφείλεται στο γεγονός ότι αντιγράφουμε τόσο την είσοδο όσο και την έξοδο από το host στο device. Κατά την εκτέλεση του kernel από το device μηχάνημα, καλείται από το host το API *cudaDeviceSynchronize*, το οποίο παύει την εκτέλεση εντολών από το host έως ότου τελειώσει η εκτέλεση του kernel. Αυτό συμβαίνει διότι δεν χρειάζεται να εκτελεστεί κάποιο κομμάτι κώδικα στο host μηχάνημα ενόσω εκτελείται ο κώδικας στο device. Διαφορετικά θα μπορούσαμε να έχουμε overlap εκτέλεσης

κώδικα στο host με το device. Τέλος, αφού τελειώσει η εκτέλεση του kernel μεταφέρουμε το παραχθέν bitmap από το device στο host μηχάνημα. Η μνήμη που καταλαμβάνει η είσοδος με την οποία τροφοδοτήθηκε ο αλγόριθμος απελευθερώνεται και δεν χρειάζεται να αντιγράψουμε ξανά την είσοδο από το device στο host.

Εικόνα 42: Εκτέλεση filtering's kernel.

Κοιτάζοντας στα χαρακτηριστικά της GPU, τα οποία και αναφέρονται στο υποκεφάλαιο 2.5, βλέπουμε ότι η κάρτα γραφικών είναι εφοδιασμένη με δύο streaming multiprocessors. Κατά την κλήση του kernel από το host χρησιμοποιούμε 32 block των 1024 thread έκαστο. Έτσι πετυχαίνουμε το μέγιστο occupancy για την κάρτα γραφικών μας χωρίς να μένει κάποιος multiprocessor αδρανής. Ο ισχυρισμός μας εδραιώνεται από τις δύο εικόνες που ακολουθουν, τις 43 και 44, οι οποίες απεικονίζουν τα επίπεδα utilization που επιτυγχάνουμε για τους multiprocessor. Το γεγονός αυτό υποδεικνύει ότι η απόδοση του kernel δεν περιορίζεται από το occupancy των multiprocessor.



Εικόνα 43: Multiprocessors' utilization.

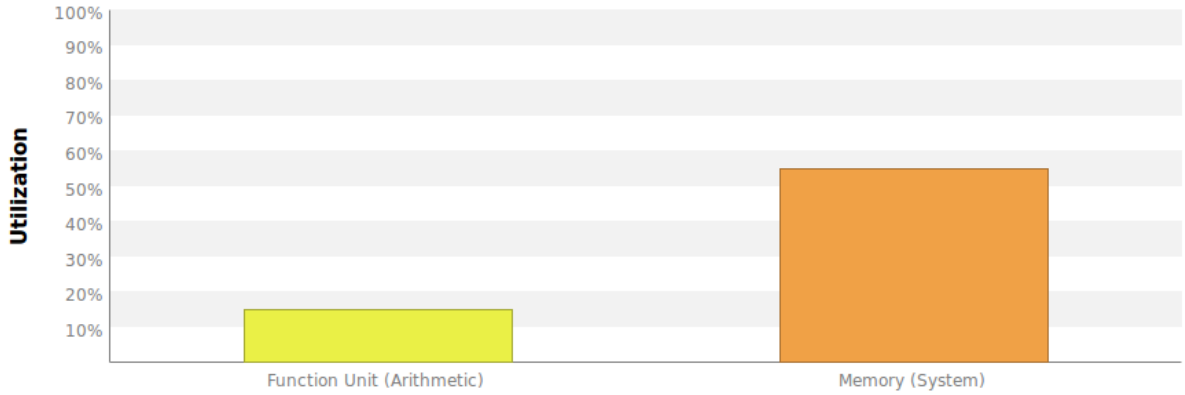
Variable	Achieved	Theoretical	Device Limit	Grid Size: [32,1,1] (32 blocks)Block Size: [1024,1,1] (1024 threads)
Occupancy Per SM				
Active Blocks		2	16	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Active Warps	64	64	64	0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64
Active Threads		2048	2048	0 256 512 768 1024 1280 1536 1792 2048
Occupancy	100%	100%	100%	0% 15% 30% 45% 60% 75% 90% 100%

Εικόνα 44: Multiprocessors' occupancy

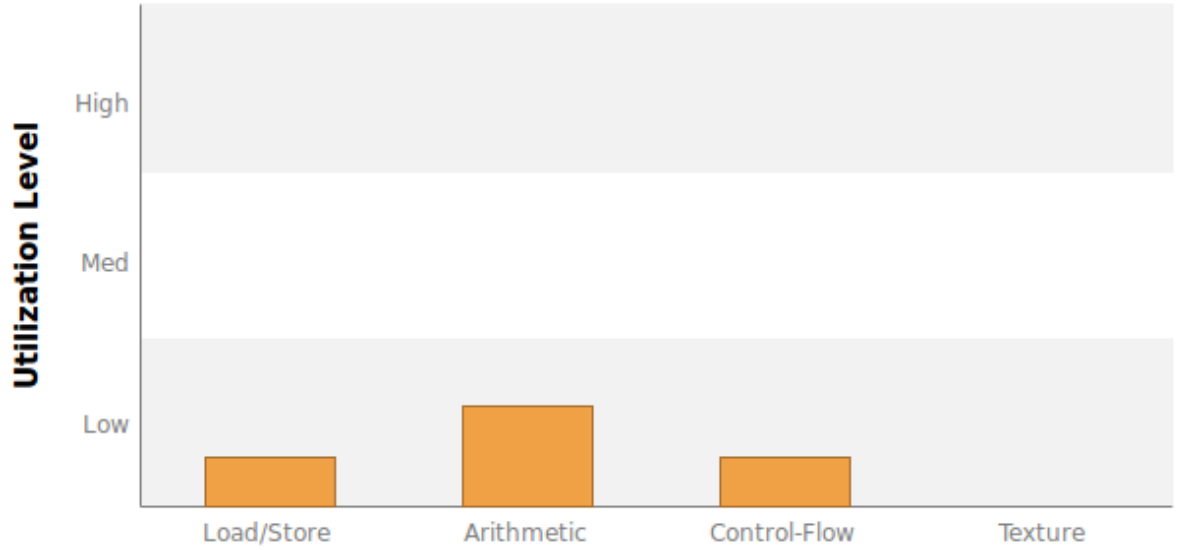
Αντιθέτως, παρατηρούμε ότι το compute throughput και το memory bandwidth utilization λαμβάνουν μικρές τιμές και πιο συγκεκριμένα μικρότερες του 60%. Το γεγονός αυτό τυπικά υποδεικνύει latency issues, τα οποία και μειώνουν την απόδοση του kernel. Παρόλα αυτά, επειδή χρησιμοποιούμε 32 block των 1024 thread, η επιρροή των latency issues στην τελική απόδοση της εκτέλεσης είναι μικρότερη από ότι παρουσιάζεται για έναν

kernel. Να υπενθυμίσουμε επιπλέον ότι τα δεδομένα εισόδου είναι τύπου streaming, γεγονός το οποίο σημαίνει ότι θα υποστούν επεξεργασία μόνο μία φορά έκαστο.

Στην εικόνα 45 φαίνεται το επίπεδο χρησιμοποίησης κάθε function unit καθενός multiprocessor. Διακρίνεται εύκολα ότι τα επίπεδα αυτά είναι αρκετά χαμηλά, πράγμα το οποίο υποδηλώνει ότι δεν περιορίζουν την απόδοση του kernel.



Εικόνα 45: Kernel performance boundary



Εικόνα 46: Function unit utilization

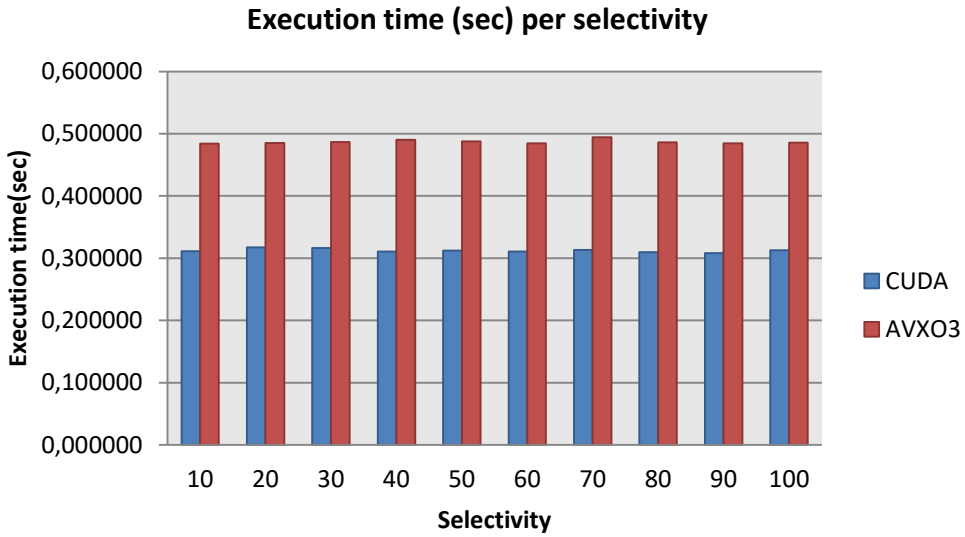
- Load/Store unit: load/store εντολές για local, shared, global και άλλου τύπου μνήμες
- Arithmetic unit: όλες οι αριθμητικές εντολές για ακεραίους, αριθμούς κινητής υποδιαστολής, λογικές και δυαδικές εντολές κ.λπ
- Control-flow unit: direct και indirect branches, jumps and calls
- Texture unit: για εντολές που χρησιμοποιούν την texture memory

Στο υποκεφάλαιο 1.3.4 αναφέρθηκαν οι πιθανοί σχεδιαστικοί λόγοι οι οποίοι μπορεί να περιορίσουν την απόδοση της εφαρμογής. Όσον αφορά τα shared memory bank conflicts, δεν έχουνε επίδραση στην εκτέλεση του kernel αφού δεν κάνουμε χρήση της shared memory μέσα στον κώδικά μας. Επιπροσθέτως, ο περιορισμός του coalesced access

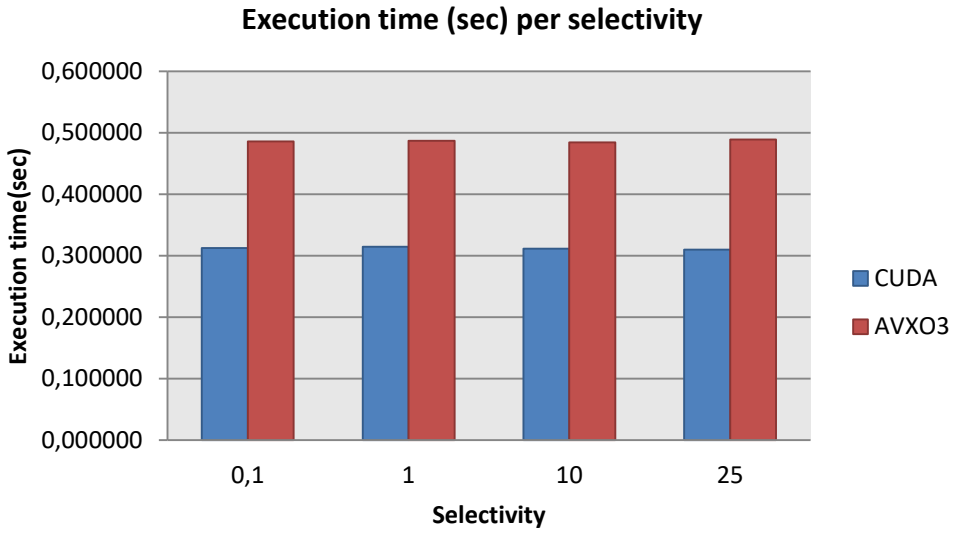
memory pattern ικανοποιείται από τον κώδικά μας, διότι ένα warp προσπελαύνει γειτονικές θέσεις μνήμης.

Τέλος, και πριν προχωρήσουμε στην παρουσίαση των αποτελεσμάτων που αφορούν τους χρόνους εκτέλεσης για όλες τις τιμές του selectivity, πρέπει να σημειώσουμε ότι η κατάσταση σε ό,τι φορά του utilization που φαίνεται στις εικόνες 43 έως και 46 παραμένει αμετάβλητη καθώς το selectivity αλλάζει, οπότε η προηγούμενη ανάλυση είναι καθολική για τις τιμές που λαμβάνει το selectivity.

Στα γραφήματα 15 και 16 φαίνονται οι χρόνοι εκτέλεσης για τις δύο κατηγορίες selectivity τις οποίες μελετάμε. Η σύγκριση γίνεται για τις SIMT και AVXO3 περιπτώσεις και αφορά τον καθαρό χρόνο εκτέλεσης για την SIMT αρχιτεκτονική, χωρίς να λαμβάνουμε υπόψιν το χρόνο που απαιτείται για την μεταφορά των δεδομένων από και προς την κάρτα γραφικών.



Γράφημα 15: Χρόνος εκτέλεσης ανά selectivity



Γράφημα 16: Χρόνος εκτέλεσης ανά selectivity

Παρατηρούμε ότι ο απαιτούμενος χρόνος εκτέλεσης παραμένει σταθερός καθώς το selectivity μεταβάλλεται. Σε αυτό βοηθάει και ο τρόπος με τον οποίο έγινε η αντίστοιχη υλοποίηση σε επίπεδο κώδικα, διότι ο κώδικας σε CUDA δεν παρουσιάζει φαινόμενα branch divergence κατά την διαδικασία παραγωγής του bitmap. Αυτός είναι και ο λόγος για τον οποίο ο χρόνος εκτέλεσης δεν εξαρτάται από τις διάφορες τιμές του selectivity.

Οι ακριβείς χρόνοι εκτέλεσης, μετρούμενοι στην κλίμακα του δευτερολέπτου (sec), για τις διάφορες τιμές του selectivity, παρουσιάζονται στους δύο παρακάτω πίνακες.

	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
CUDA	0,311	0,317	0,316	0,310	0,312	0,310	0,313	0,309	0,308	0,312
AVXO3	0,484	0,484	0,486	0,490	0,487	0,484	0,494	0,486	0,484	0,485

	0,1%	1%	10%	25%
CUDA	0,312	0,314	0,311	0,309
AVXO3	0,485	0,486	0,484	0,488

Παρατηρούμε ότι σε ό,τι αφορά τον καθαρό χρόνο εκτέλεσης, το επιτευχθέν speed-up κυμαίνεται περίπου στο 35%. Τέλος, ο χρόνος που χρειάστηκε για την αντιγραφή των δεδομένων από το host στο device και τούμπαλιν είναι ίσος με 0,913652 δευτερόλεπτα.

3.5 Aggregation functions: Σύγκριση SIMT – AVX

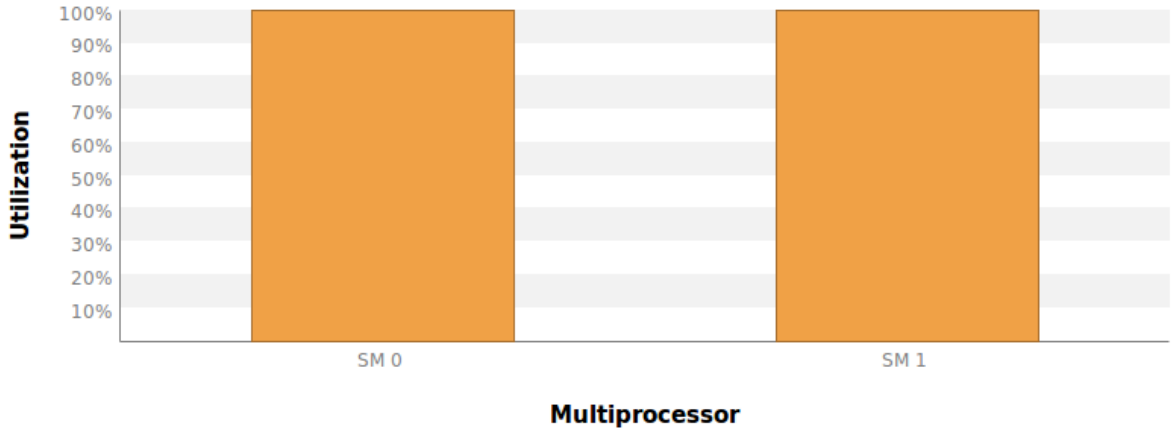
Η σύγκριση για τα τέσσερα προς εξέταση aggregation functions για τις SIMT και SIMD αρχιτεκτονικές θα γίνει σε τέσσερα τμήματα, ένα για το κάθε aggregation function ξεχωριστά, μιας και κάποια στοιχεία που αφορούν το utilization της κάρτας γραφικών είναι διαφορετικά ανά aggregation function. Πριν όμως ξεκινήσουμε αυτήν την σύγκριση, θα παραθέσουμε δύο κοινά στοιχεία εκτέλεσης για όλα τα aggregation functions.

Για όλα τα aggregations χρησιμοποιούμε 32 block των 1024 thread, καταφέρνοντας να πετύχουμε occupancy στους multiprocessors της GPU της τάξεως του 100% και έτσι η απόδοση να μην περιορίζεται επειδή κάποιος multiprocessor είναι αδρανής.

Επίσης, επειδή κάνουμε χρήση του αλγορίθμου parallel reduction, κάθε εκτέλεση κάποιου kernel εμφανίζει ένα σταθερό και αναπόφευκτο branch divergence της τάξης του 3,1%. Το γεγονός προκαλεί η εντολή ελέγχου *if (tid == 0)*. Τα 1024 threads αντιστοιχούν σε 32 warps των 32 thread έκαστο. Το φαινόμενο του branch divergence οφείλεται στο γεγονός ότι ένα thread μέσα στο πρώτο warp θα αποτιμήσει την προαναφερθείσα συνθήκη ως taken, ενώ τα υπόλοιπα 31 ως not taken. Έτσι, 32 εκτελέσεις, από τις 1024 συνολικά, της εντολής θα είναι divergent προκαλώντας αυτό το πάγιο ποσοστό branch divergence για όλες τις εκτελέσεις.

Οι δύο αυτοί ισχυρισμοί συνοψίζονται στις εικόνες 47, 48 και 49 αντίστοιχα. Στην εικόνα 49 φαίνεται αριστερά το κομμάτι κώδικα γραμμένο σε C/C++ ενώ δεξιά ένα κομμάτι assembly κώδικα από αυτό το μπλοκ. Εδώ φαίνεται πως ο nvcc compiler χρησιμοποιεί τα

κατηγορήματα. Το κόκκινο χρώμα αριστερά της εντολής διακλάδωσης δείχνει τον αριθμό των inactive threads. Για την συγκεκριμένη περίπτωση τα threads αυτά έχουνε χαρακτηριστεί ως inactive, διότι για αυτά η εκτέλεση του kernel έχει τελειώσει.



Εικόνα 47: Ποσοστό χρησιμοποίησης των multiprocessors για όλα τα aggregation functions

Occupancy Per SM

Active Blocks		2	16
Active Warps	64	64	64
Active Threads		2048	2048
Occupancy	100%	100%	100%

Εικόνα 48: Multiprocessors' occupancy για όλα τα aggregation functions

```
if(tid == 0){                          ISETP.NE.AND P2, PT, R7, RZ, PT;
    g_odata[blockIdx.x] = sdata[0];    @!P0 LDS R3, [R2];
}                                      @!P0 LDS R6, [R2+0x100];
}                                      @!P0 IADD R3, R3, R6;
                                       @!P0 STS [R2], R3;
                                       BAR.SYNC 0x0;
                                       @P1 NOP.S;
```

Εικόνα 49: Σημείο ύπαρξης divergence κατά το στάδιο του parallel reduction.

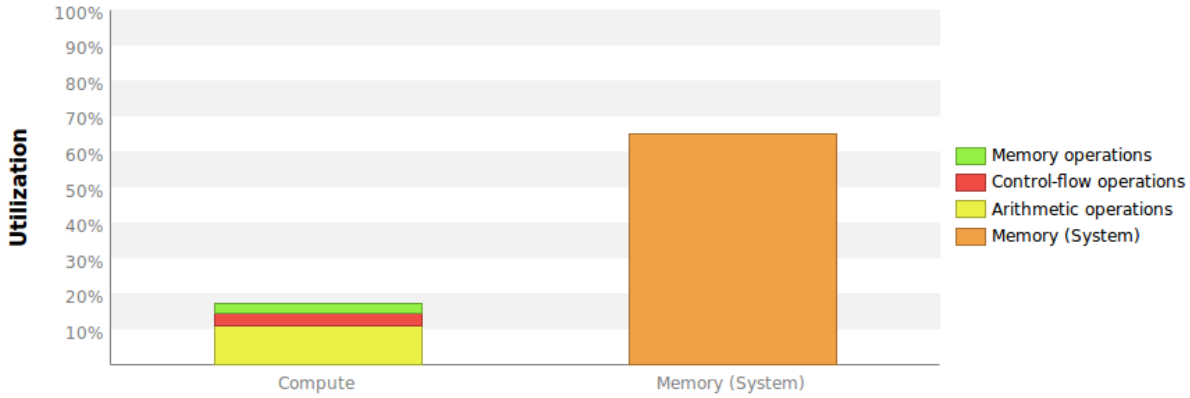
Τελειώνοντας την καθολική ανάλυση, αναφέρουμε ότι ο NVVP profiler εμφανίζει την ένδειξη “Low Global Memory Store Efficiency” με τιμή ίση με 12.5% για όλα τα kernel. Η μετρική αυτή ορίζεται ως ο λόγος ανάμεσα στον αριθμό των byte που έγιναν store προς τον αριθμό των byte που χρειάστηκε να μεταφερθούν για να γίνει αυτό το store. Σε μορφή κλάσματος έχουμε το παρακάτω:

$$Global\ Memory\ Store\ Efficiency = \frac{\#bytes\ stored}{\#bytes\ that\ were\ transferred\ to\ device\ memory}$$

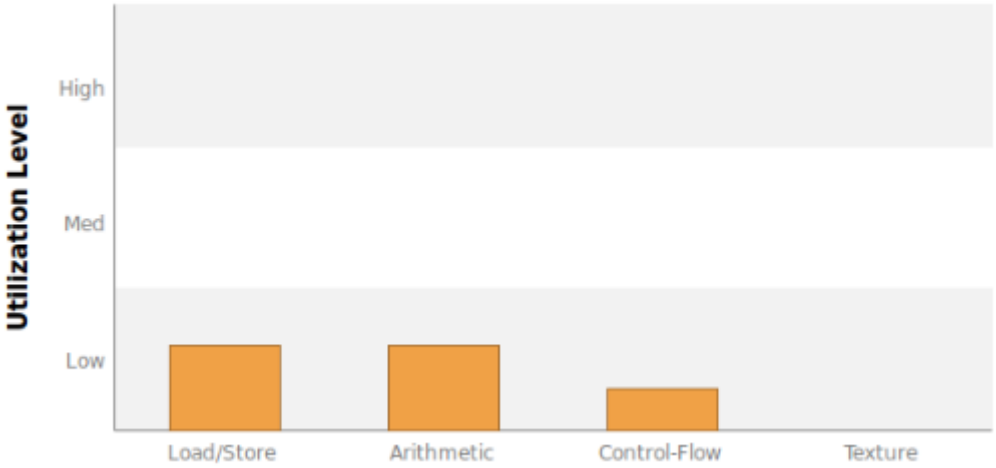
Ανατρέχουμε τον αναγνώστη στο υποκεφάλαιο 1.3.4, όπου και έχει αναφερθεί ότι για λόγους απόδοσης 32 byte transaction συμβαίνει όταν το warp διαβάζει είσοδο με τύπο αναπαράστασης τα 8 bit. Αυτό διότι 8 bit είναι ίσα με 1 byte, άρα 32 thread του warp διαβάζουν 32 byte, όσο και το μέγεθος του transaction. Το “πρόβλημα” που εντοπίζει ο profiler βρίσκεται στο store στάδιο και όχι στο load, αφού όπως έχουμε αναφέρει η πρόσβαση στην μνήμη σέβεται το coalesced memory access pattern, όπου εκεί η αντίστοιχη μετρική ισούται με 100%. Κάθε kernel επιστρέφει μία τιμή ως αποτέλεσμα, η οποία και

γράφεται από την shared memory στην global memory. Η global memory έχει τύπο αναπαράστασης μεγέθους 32 bit και ο τύπος της τιμής που πρέπει να γίνει store είναι επίσης 32 bit, δηλαδή 4 byte. Το hardware, όπως έχει αναφερθεί, προσπαθεί να βελτιώσει τα transactions που γίνονται και επειδή μόνο ένα thread θα κάνει store μία τιμή σε αυτό το transaction, το transaction καταλήγει να έχει μέγεθος 32 byte. Άρα από τον ορισμό του Global Memory Store Efficiency έχουμε ότι $\frac{4\ byte}{32\ byte} = \frac{1}{8} = 0.125$ ή 12.5% .

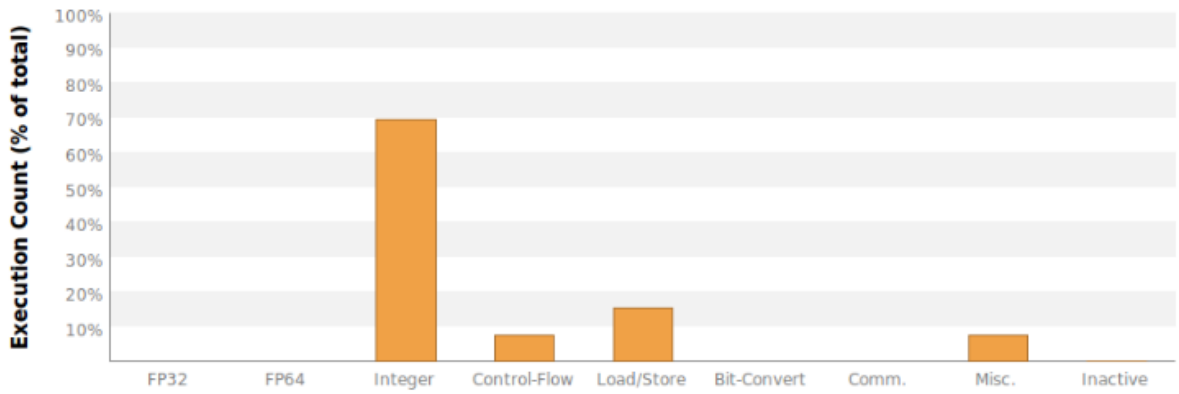
Η ανάλυση θα ξεκινήσει με το Sum aggregation function. Στην εικόνα 50 βλέπουμε ότι η απόδοση περιορίζεται από το memory bandwidth και όχι από το compute utilization, μιας και τα επίπεδα του δεύτερου είναι σε αρκετά χαμηλό σημείο. Η εικόνα 51 επιβεβαιώνει αυτό το σενάριο αφού το utilization του κάθε function unit παραμένει σε χαμηλά επίπεδα, άρα και δεν αποτελεί ανασταλτικό παράγοντα για την τελική απόδοση του kernel. Η εικόνα 52 παρουσιάζει το ποσοστό κύκλων εκτέλεσης για κάθε κλάση εντολών επί του συνόλου των κύκλων εκτέλεσης που χρειάστηκαν για να τερματίσει η εκτέλεση του kernel. Η πλειοψηφία των κύκλων εκτέλεσης αφιερώθηκαν για εντολές της κλάσης Integer, όπως και ήτανε αναμενώμενο βάση κώδικα. Παρατηρούμε επίσης ότι οι κύκλοι που αφορούν τα inactive threads είναι σχεδόν μηδενικοί, ένδειξη η οποία είναι θετική για την απόδοση του kernel.



Εικόνα 50: Kernel performance boundary – Sum.

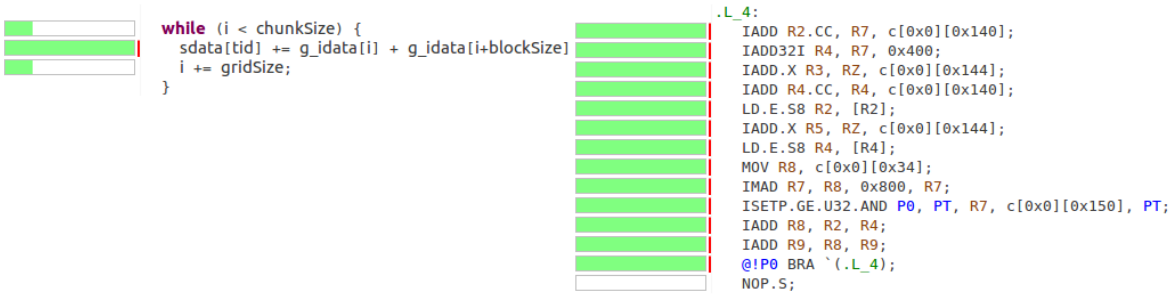


Εικόνα 51: Function unit utilization – Sum.



Εικόνα 52: Instruction execution counts – Sum.

Στο αριστερό μέρος της εικόνας 53 φαίνεται το κύριο τμήμα υπολογισμού του SUM aggregation function, πριν αρχίσει η εκτέλεση του parallel reduction. Στο δεξί μέρος της ίδιας εικόνας βλέπουμε τον αντίστοιχο κώδικα σε assembly. Το πράσινο χρώμα στις μπάρες δίπλα από τις εντολές δείχνει ότι τα thread που εκτελούν τις εντολές αυτές είναι active, δηλαδή όλα τα thread του kernel εκτελούν τις εντολές χωρίς να εμφανίζεται φαινόμενο branch divergence, όπως άλλωστε είναι αναμενόμενο. Βλέπουμε άλλωστε ότι το μόνο κατηγόρημα που θα υπολογιστεί στον assembly κώδικα αφορά την συνθήκη διακλάδωσης του loop.

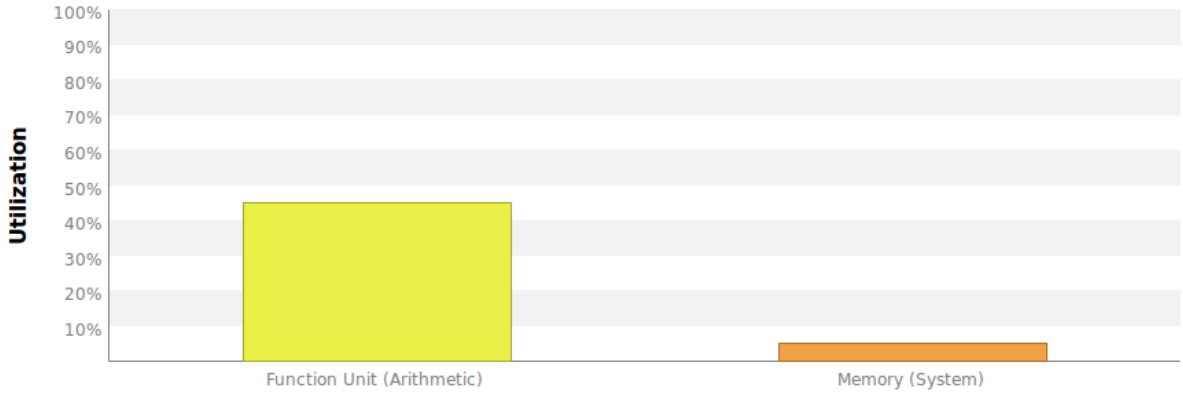


Εικόνα 53: Κομμάτι κώδικα C/C++ και ο αντίστοιχος κώδικας assembly – Sum.

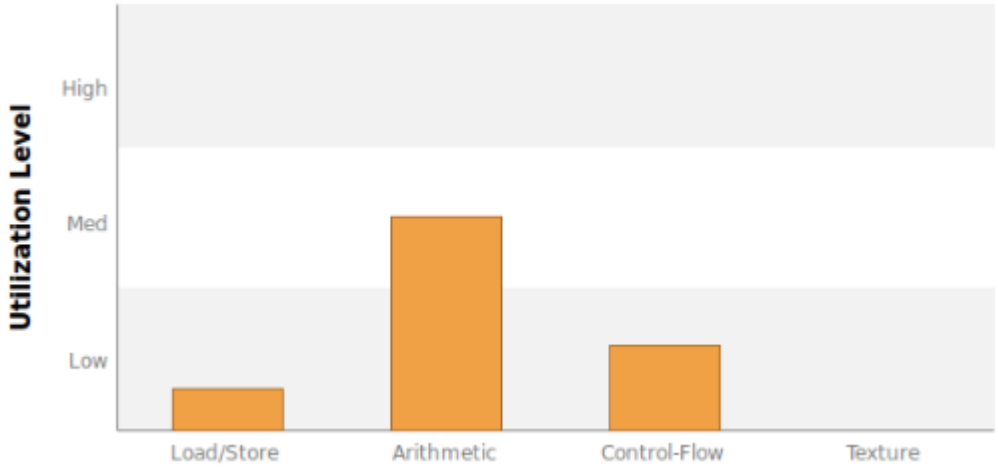
Όλα τα thread χρησιμοποιούν την shared memory και το καθένα χρησιμοποιεί διαφορετική διεύθυνση της shared memory. Όλα τα threads αποθηκέυουν στην shared memory από έναν αριθμό τύπου integer, μεγέθους 4 bytes. Υπενθυμίζουμε ότι η shared memory είναι χωρισμένη σε 32 banks των 4 byte έκαστο. Όλες αυτές οι συνθήκες λοιπόν καθιστούν την πρόσβαση στην shared memory bank conflict free, δηλαδή δεν εμφανίζονται φαινόμενα bank conflict τα οποία θα μπορούσανε να μειώσουν την απόδοση της εκτέλεσης.

Συνοψίζοντας λοιπόν, ο κώδικας δεν εμφανιζει φαινόμενο branch divergence, πέραν του πάγιου divergence που έχει αναφερθεί και είναι ασήμαντο, σέβεται το coalesced memory access pattern και δεν εμφανιζει και φαινόμενα shared memory bank conflict.

Η ανάλυση συνεχίζεται για το Count aggregation function. Σε αντίθεση με το πρόβλημα του Sum, εδώ βλέπουμε ότι το utilization των function units που εκτελούν arithmetic operations είναι μεγαλύτερο από το utilization της μνήμης, γεγονός αρκετά λογικό αφού για το συγκεκριμένο πρόβλημα απλά μετράμε τον αριθμό των στοιχείων χωρίς να κάνουμε κάποια πράξη πάνω σε αυτά. Συγκρίνοντας την εικόνα 54 του προβλήματος Count με την εικόνα 51 του προβλήματος Sum, βλέπουμε ότι το utilization level για load / store εντολές είναι μικρότερο και ότι το αντίστοιχο utilization για τις arithmetic εντολές είναι μεγαλύτερο.



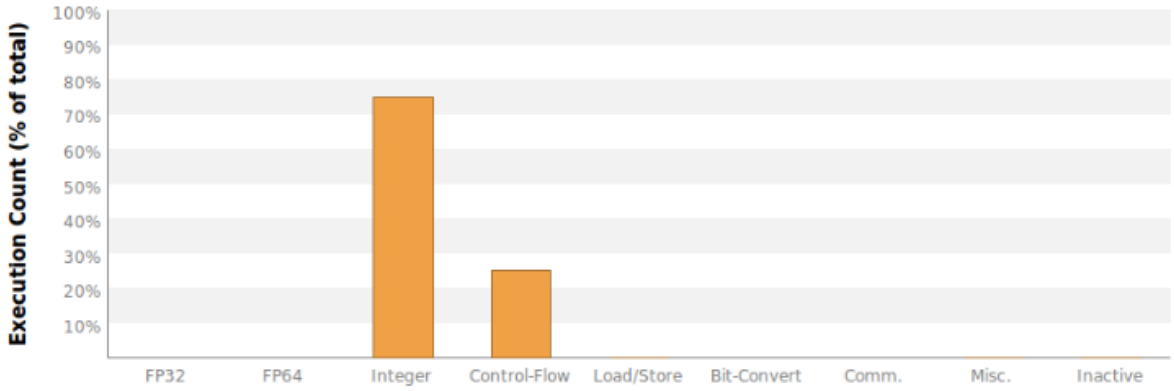
Εικόνα 54: Kernel performance boundary – Count.



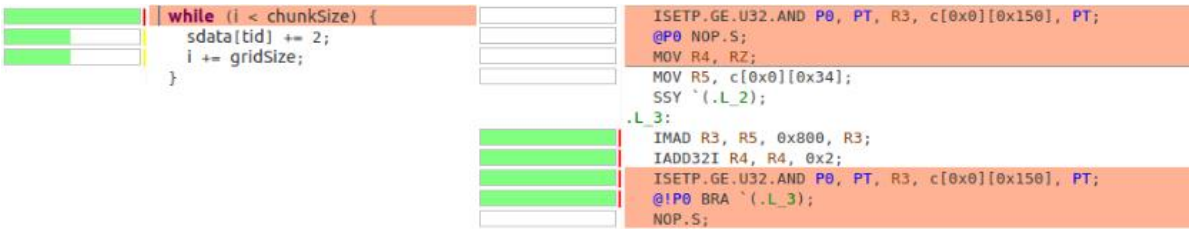
Εικόνα 55: Function unit utilization – Count.

Στην εικόνα 56 φαίνεται το ποσοστό κύκλων εκτέλεσης επί των συνολικών κύκλων για κάθε τύπο εντολών. Οι απαιτούμενοι κύκλοι για τις εντολές τύπου load/store είναι ελάχιστοι, επαληθεύοντας έτσι το χαμηλό utilization αυτών των function units. Λογικό είναι επίσης και το γεγονός ότι το ποσοστό των απαιτούμενων κύκλων για control-flow εντολές είναι μεγαλύτερο από το αντίστοιχο για την Sum υλοποίηση. Δεν θα πρέπει να παραβλέψουμε το γεγονός άλλωστε ότι αυτά είναι σχετικά νούμερα και όχι απόλυτα. Στο κύριο κομμάτι υπολογισμού για το Count πρόβλημα δεν έχουμε εντολές load/store οπότε ο συνολικών αριθμός των κύκλων για την εκτέλεση του προγράμματος είναι μικρότερος, αυξάνοντας έτσι το σχετικό μέγεθος των κύκλων για εντολές τύπου control-flow. Και για

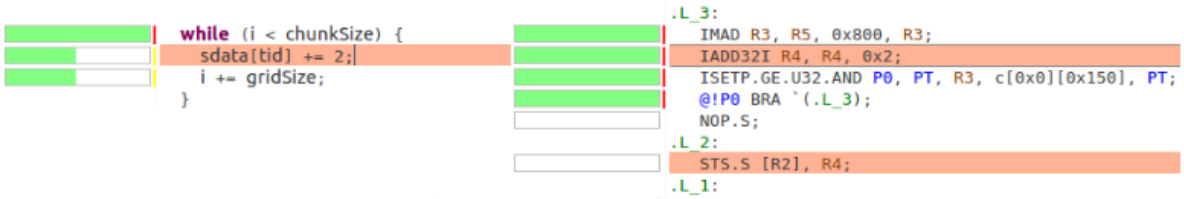
αυτό το aggregation function παρατηρούμε ότι ο αριθμός των inactive κώκλων είναι μηδενικός, αποφεύγοντας φαινόμενα branch divergence καθώς όλα τα threads εκτελούνε όλες τις εντολές χωρίς να είναι κάποια inactive ή predicated off, όπως παρουσιάζουν και οι εικόνες 57, 58 και 59 στις οποίες φαίνεται όλη η ροή εκτέλεσης του κύριου υπολογισμού καθ' αντιστοιχία του κώδικα σε C/C++ με την assembly.



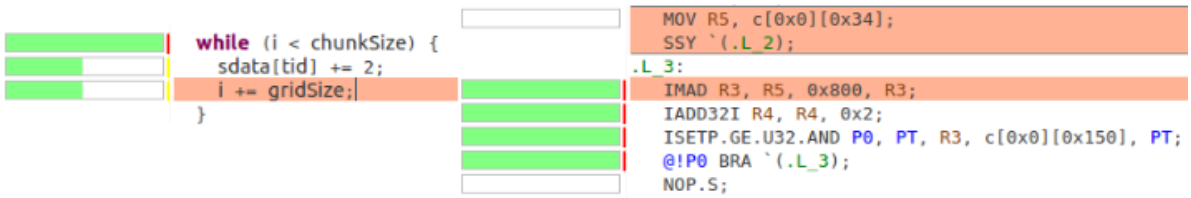
Εικόνα 56: Instruction execution counts - Count.



Εικόνα 57: Κομμάτι κώδικα C/C++ και η αντίστοιχη assembly – συνθήκη while – Count.

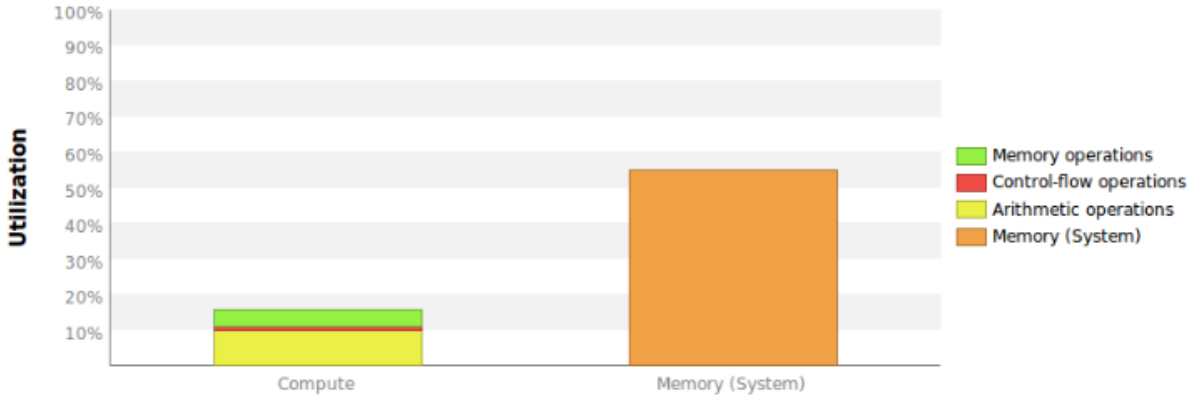


Εικόνα 58: Κομμάτι κώδικα C/C++ και η αντίστοιχη assembly – κύριο τμήμα – Count.



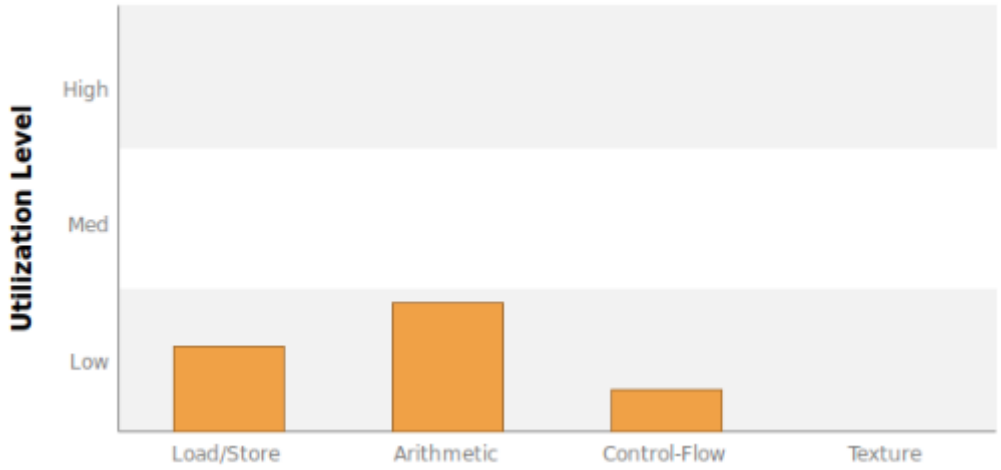
Εικόνα 59: Κομμάτι κώδικα C/C++ και η αντίστοιχη assembly – υπολογισμός iterator – Count.

Η αντίστοιχη ανάλυση για το Min πρόβλημα φαίνεται παρακάτω. Στην εικόνα 60 βλέπουμε ότι το utilization που αφορά την μνήμη είναι συγκριτικά μεγαλύτερο από το utilization που αφορά υπολογιστικά ζητήματα. Την ίδια περίπτωση συναντήσαμε και για το Sum πρόβλημα, εν αντιθέσει με το Count, όπου το memory utilization ήτανε μικρό. Αυτό οφείλεται στο ότι τα δεδομένα φορτώνονται από την μνήμη και έτσι περνούν από τον PCIe δίαυλο, αυτό μας δείχνει και η εικόνα 60. Μάλιστα, προσεγγιστικά πετυχαίνουμε memory bandwidth της τάξης του 7 GB / sec.

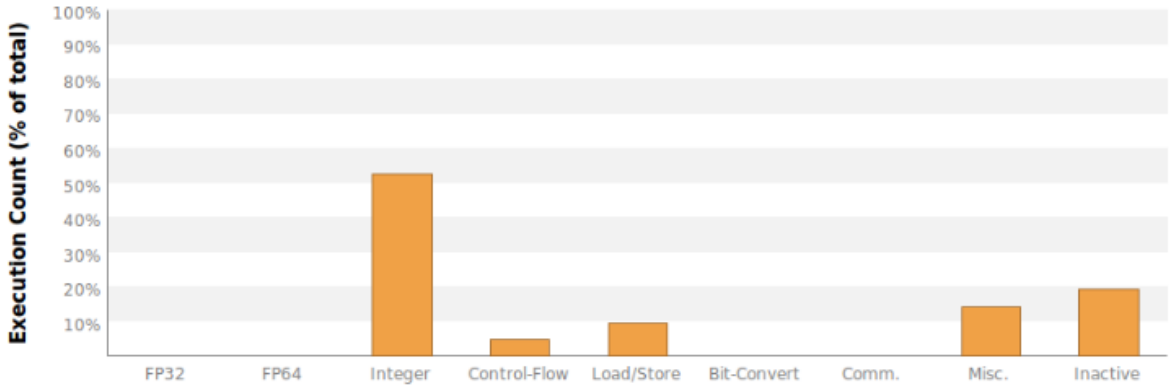


Εικόνα 60: Kernel performance boundary - Min.

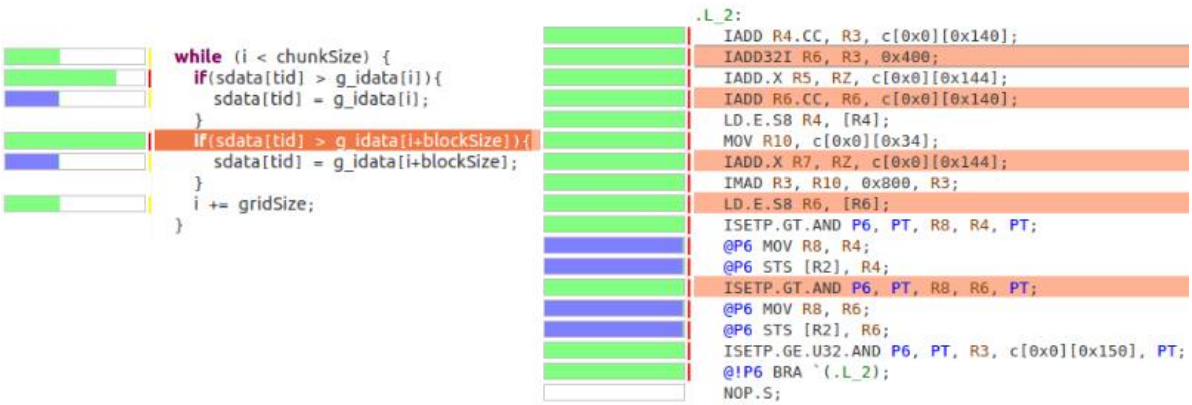
Παρατηρούμε επίσης ότι το function unit utilization για το Min πρόβλημα είναι παρόμοιο με αυτό του Sum, γεγονός αρκετά λογικό, διότι οι εντολές για load / store είναι ίδιες, οι arithmetic εντολές είναι ίδιες περίπου στον αριθμό και για τις δύο περιπτώσεις (σ.σ: δύο συγκρίσεις έναντι δύο αθροίσεων) και το utilization για τις control flow εντολές αντισταθμίζεται από την ύπαρξη predicated-off threads, τα οποία φτάνουν στο 98%. Οι εντολές που θα εκτελεστούν από predicated-off threads φαίνονται στην εικόνα 63, με μπλε χρώμα στα αριστερά.



Εικόνα 61: Function unit utilization - Min.

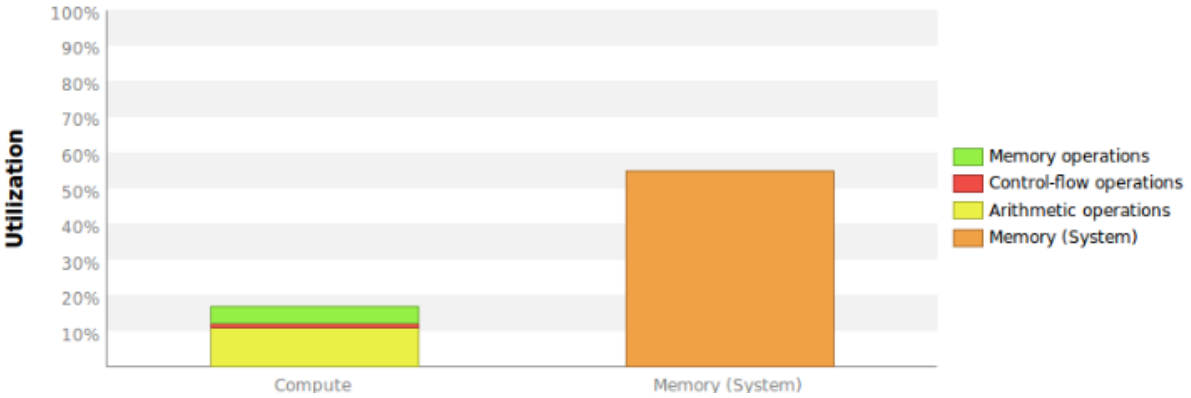


Εικόνα 62: Instruction execution counts - Min.

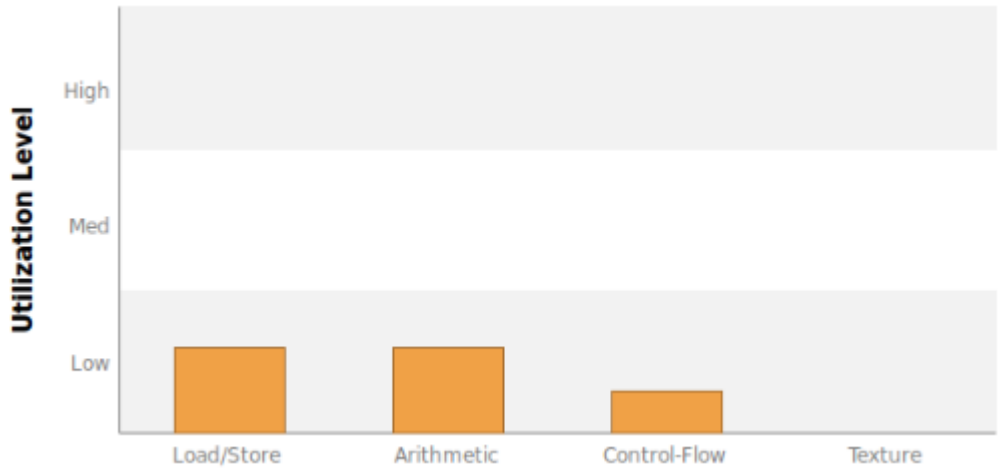


Εικόνα 63: Κομμάτι κώδικα C/C++ και η αντίστοιχη assembly - Min.

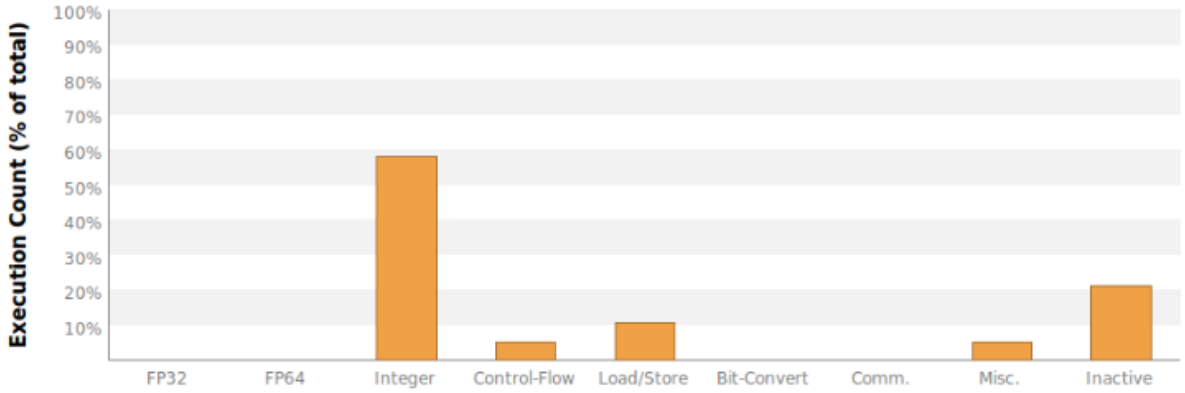
Στη συνέχεια, παρατίθενται οι ίδιες μετρικές για το αντίστοιχο πρόβλημα Max, το οποίο είναι το δυϊκώς αντίστροφο από το Min πρόβλημα που αναλύσαμε προηγουμένως.



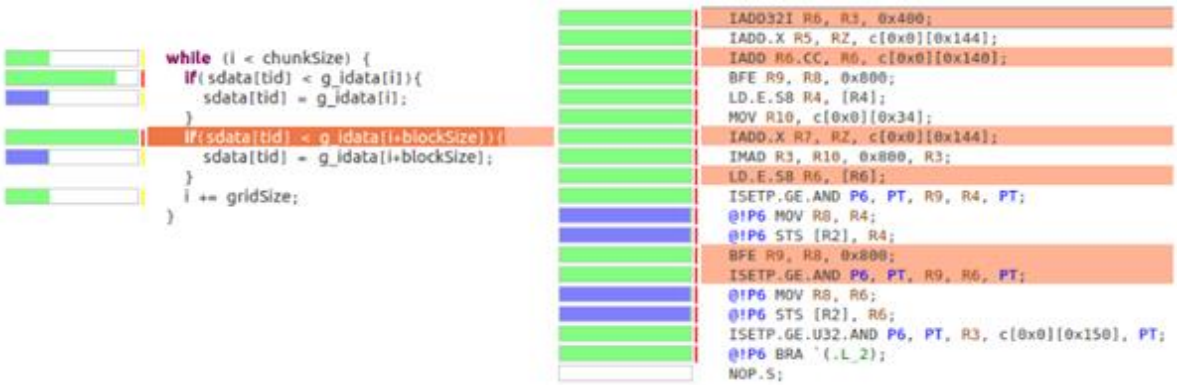
Εικόνα 64: Kernel performance boundary - Max.



Εικόνα 65: Function uniti utilization - Max.

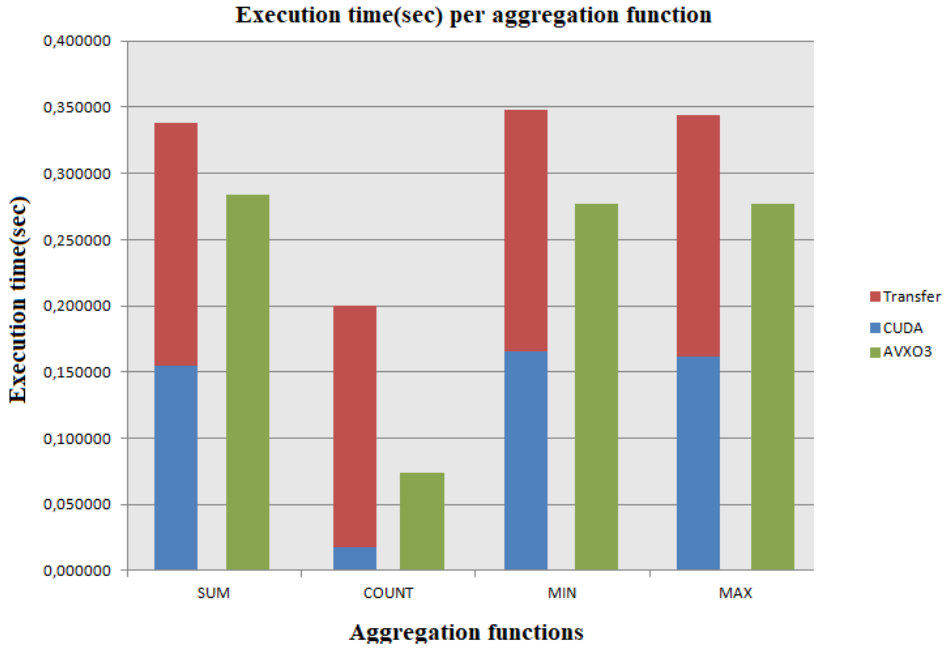


Εικόνα 66: Instruction execution counts - Max.



Εικόνα 67: Κομμάτι κώδικα C/C++ και η αντίστοιχη assembly - Max.

Τα τελικά αποτελέσματα φαίνονται στο γράφημα 17. Με κόκκινο χρώμα απεικονίζεται ο απαιτούμενος χρόνος για την μεταφορά των δεδομένων στην μνήμη της κάρτας γραφικών, με μπλε ο καθαρός χρόνος εκτέλεσης μέσα στην κάρτα γραφικών και με πράσινο χρώμα ο απαιτούμενος χρόνος εκτέλεσης στην AVX-256 αρχιτεκτονική. Στον πίνακα που ακολουθεί υπάρχουν οι ακριβείς χρόνοι εκτέλεσης μετρούμενοι στην κλίμακα του ενός δευτερολέπτου.



Γράφημα 17: Χρόνοι εκτέλεσης – Σύγκριση CUDA - AVX

	SUM	COUNT	MIN	MAX
CUDA	0,154960	0,017391	0,165330	0,161130
AVXO3	0,279595	0,071223	0,270542	0,270761

Τέλος, μιλώντας σε όρους παραλληλίας, όλες οι εκτελέσεις στην CPU είναι μονονηματικές και για την περίπτωση του AVX, επεξεργάζονται παράλληλα οκτώ στοιχεία. Αντίθετα, στην περίπτωση της GPU το μέγιστο όριο ενεργών νημάτων ισούται με 384.

Κεφάλαιο 4 – Μελέτη πολυνηματικής υλοποίησης σε Scalar και AVX

4.1 Πολυνηματική υλοποίηση

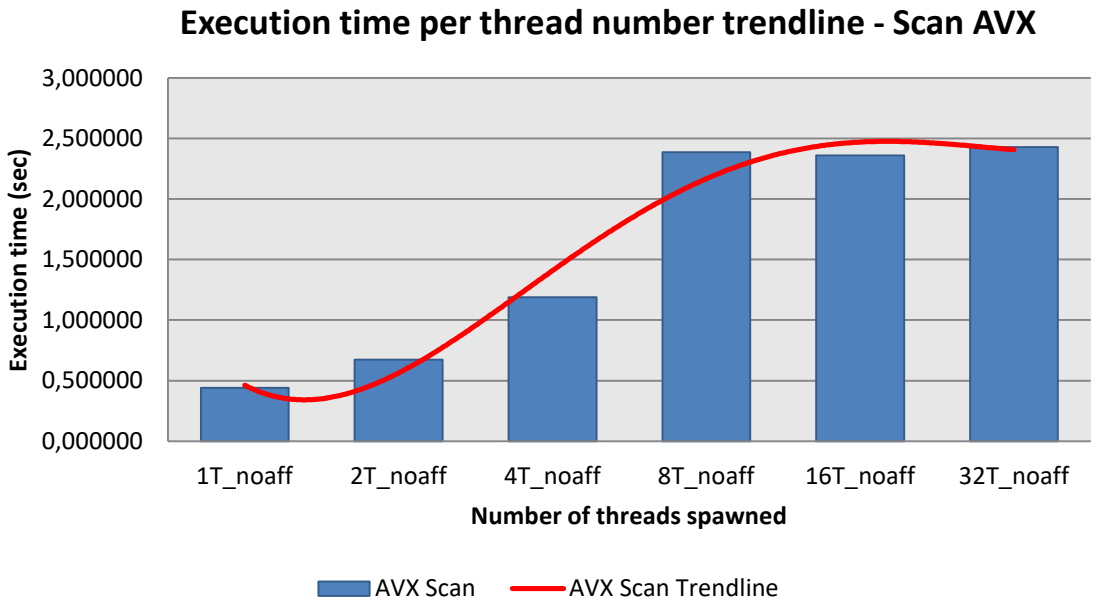
Όλα τα benchmarks που υλοποιήσαμε και τρέξανε στην CPU αφορούν υλοποίηση ενός νήματος. Στην συνέχεια θα παρουσιάσουμε τα αποτελέσματα της πολυνηματικής υλοποίησης για αυτούς τους αλγορίθμους. Στο σημείο αυτό πρέπει να αναφέρουμε ότι κατά

την δημιουργία των posix threads δεν θέτουμε συγκεκριμένη τιμή processor affinity. Affinity είναι η διαδικασία αντιστοίχισης software threads με hardware threads. Χρησιμοποιώντας την τεχνική affinity η εκτέλεση του κάθε νήματος φαίνεται αποκλειστικά και μόνο σε έναν φυσικό πυρήνα και από συγκεκριμένο hardware thread («λογικός» πυρήνας). Το λειτουργικό σύστημα έχει την δυνατότητα να προχωρήσει σε thread migration από πυρήνα σε πυρήνα, εφόσον δεν έχει τεθεί η affinity mask.

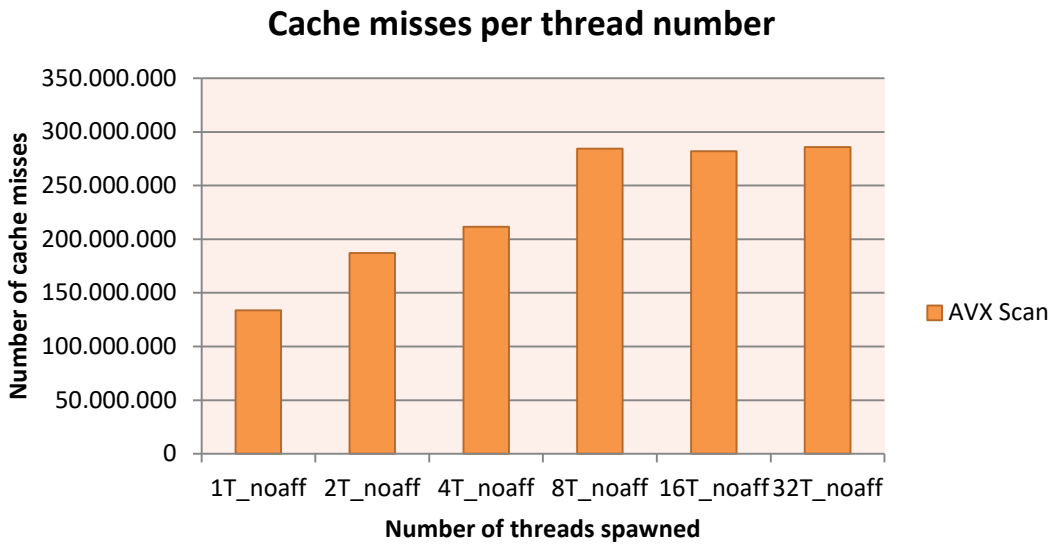
Εξετάσαμε τις περιπτώσεις όπου δημιουργούνται δύο, τέσσερα και τέλος οκτώ νήματα, όπου το καθένα έχει το δικό του κομμάτι δεδομένων για να επεξεργαστεί. Για όλες τις υπό εξέταση περιπτώσεις έχει γίνει ενεργοποίηση του O3 flag.

4.1.1 Πολυνηματική υλοποίηση – Filtering

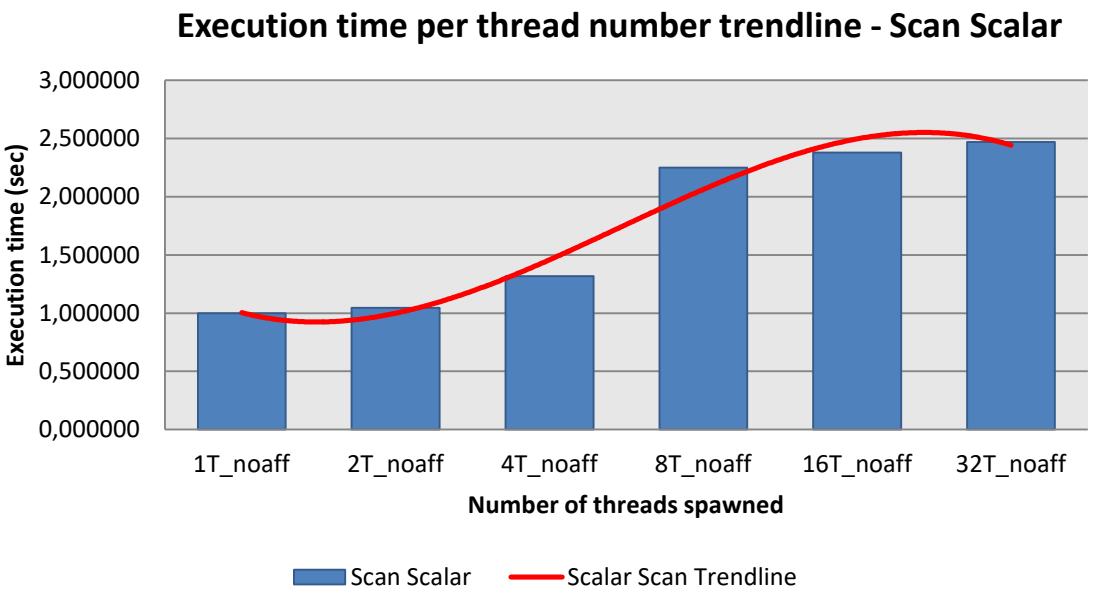
Τόσο στην περίπτωση του AVX όσο και σε αυτήν του Scalar ο χρόνος εκτέλεσης αυξάνεται καθώς αυξάνεται και ο αριθμός των νημάτων. Το γεγονός αυτό γίνεται αντιληπτό από τα γραφήματα 18 και 20 που ακολουθούν. Παρατηρούμε επίσης ότι τάση του χρόνου εκτέλεσης ως προς τον αριθμό των νημάτων που χρησιμοποιούνται κάθε φορά, προσομοιάζει σε συνάρτηση τύπου Σίγμα και αναπαρίσταται με την γραμμή κόκκινου χρώματος. Παρόμοια τάση ακολουθεί και ο αριθμός των cache misses και για τις δύο περιπτώσεις, όπως διακρίνεται εύκολα στα γραφήματα 19 και 21. Πιο συγκεκριμένα, ο αριθμός των cache misses για την μετάβαση από την εκτέλεση με ένα νήμα στην εκτέλεση με οκτώ ή περισσότερα νήματα έχει τουλάχιστον διπλασιαστεί, εάν λάβουμε υπόψιν ότι στην εκτέλεση μονού νήματος συμπεριλαμβάνονται και τα cache misses που μπορεί να προκύψουν κατά την δημιουργία των στοιχείων εισόδου. Το ίδιο ισχύει και για την scalar περίπτωση, όπου παρατηρούμε ότι για την μετάβαση από την εκτέλεση ενός νήματος στην πολυνηματική εκτέλεση οκτώ ή περισσότερων νημάτων ο αριθμός των cache misses έχει σχεδόν τετραπλασιαστεί.



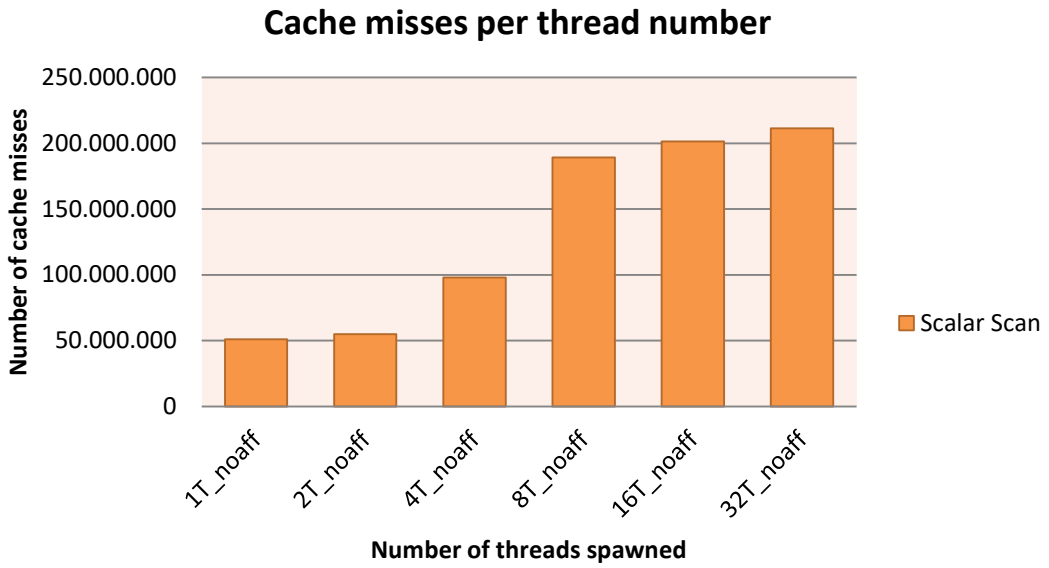
Γράφημα 18: Χρόνος εκτέλεσης ανά αριθμό νημάτων - AVX



Γράφημα 19: Αριθμός cache misses ανά αριθμό νημάτων - AVX



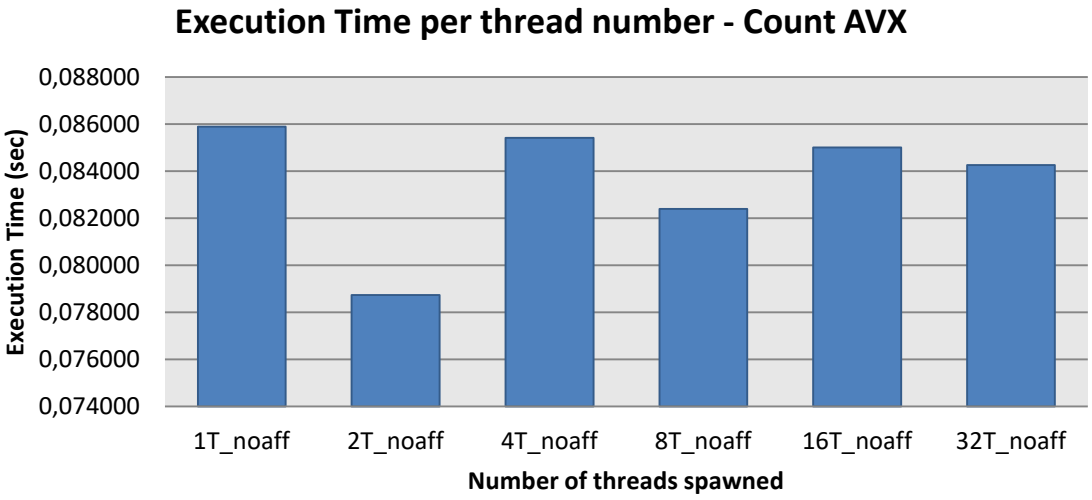
Γράφημα 20: Χρόνος εκτέλεσης ανά αριθμό νημάτων - Scalar



Γράφημα 20: Αριθμός cache misses ανά αριθμό νημάτων - Scalar

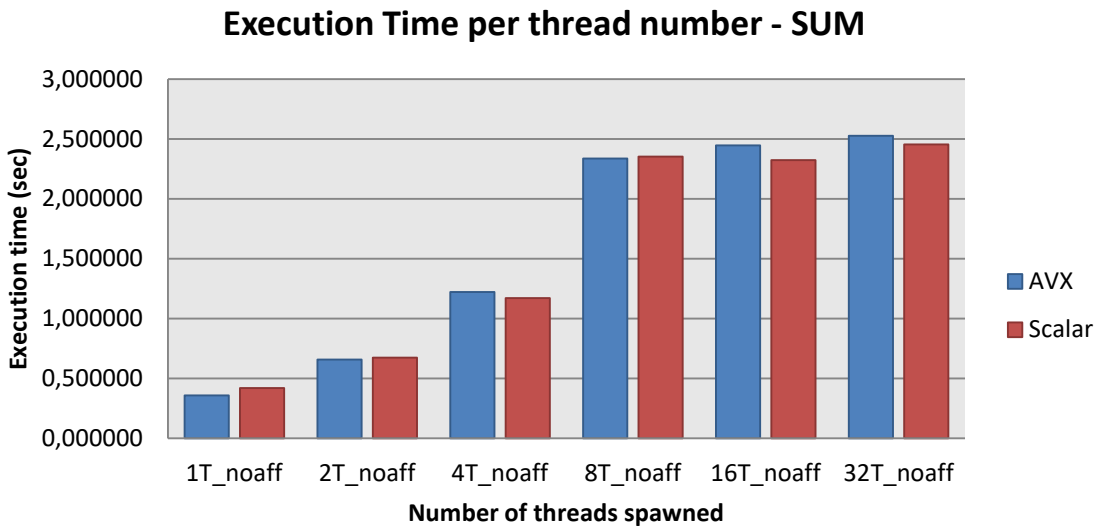
4.1.2 Πολυνηματική υλοποίηση – Aggregation functions

Μελετώντας την περίπτωση των aggregation functions, τα αποτελέσματα κινούνται στο ίδιο μήκος κύματος με την περίπτωση του Filtering. Τα αποτελέσματα της πολυνηματικής εκτέλεσης του Count Aggregation function παρατίθενται για λόγους πληρότητας στο γράφημα 21, μιας και στην περίπτωση της ενεργοποιήσης του O3 flag, ο compiler εφαρμόζει optimization για τον scalar κώδικα, μην αφήνοντας περιθώρια σύγκρισης μεταξύ των δύο περιπτώσεων. Στο σημείο αυτό να πούμε ότι ο απαιτούμενος χρόνος εκτέλεσης δεν δείχνει να εξαρτάται από τον αριθμό των νημάτων που δημιουργούνται, δεδομένου ότι θεωρούμε ότι μπορεί να υπάρξει και ένα σφάλμα της τάξεως του ±10%, περίπτωση η οποία καλύπτει και τον χρόνο εκτέλεσης για δύο νήματα.

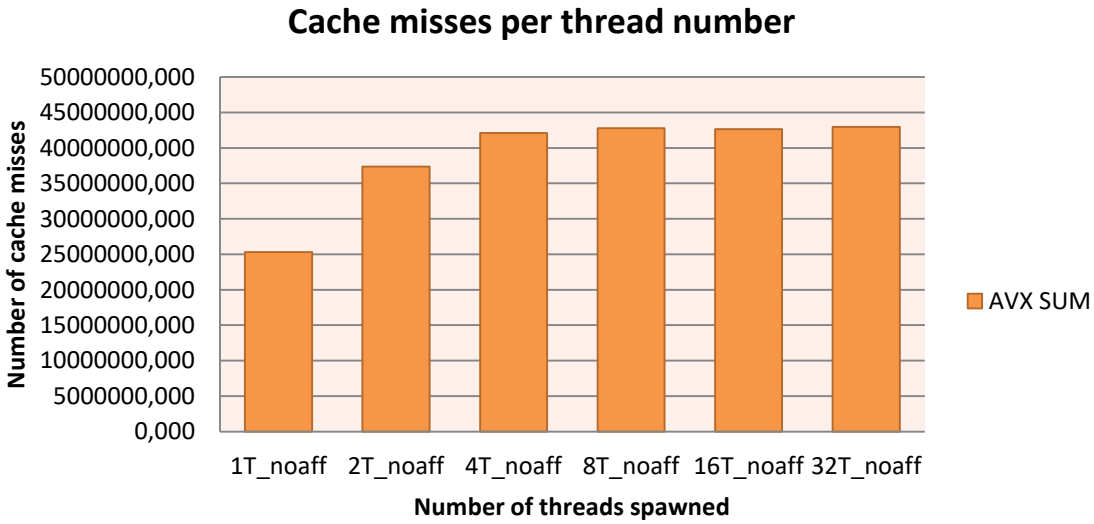


Γράφημα 21: Χρόνος εκτέλεσης ανά αριθμό νημάτων - Count AVX

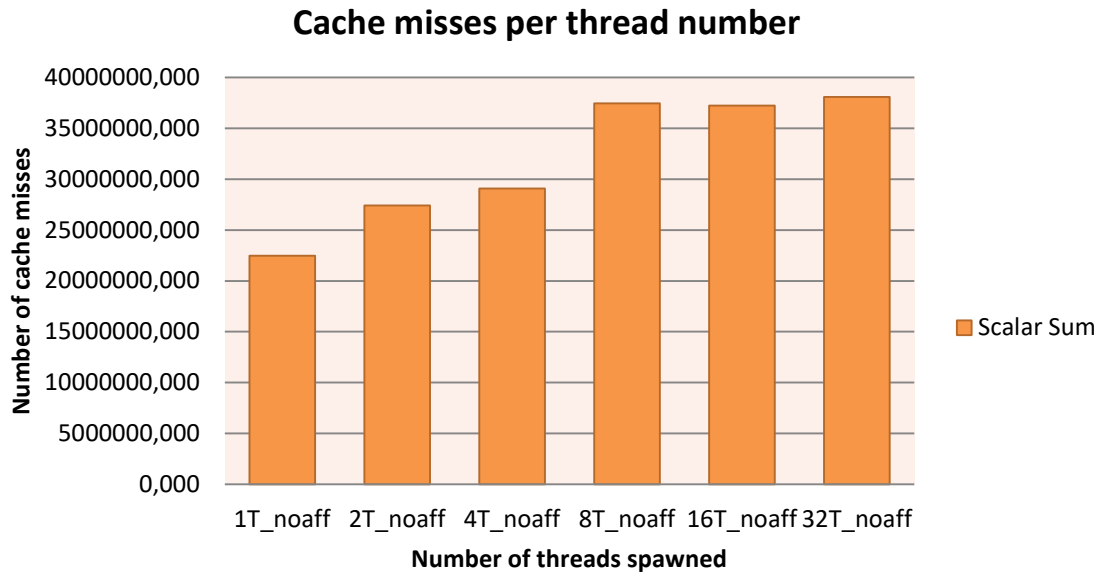
Συνεχίζοντας, παραθέτουμε αρχικά το συγκεντρωτικό γράφημα των χρόνων εκτέλεσης για το Sum. Παρομοίως, ο χρόνος εκτέλεσης αυξάνεται καθώς αυξάνεται και ο αριθμός νημάτων. Την ίδια αυξητική τάση, αλλά με διαφορετικό ρυθμό, παρουσιάζει και ο αριθμός των cache misses και για τις δύο περιτπώσεις όπως φαίνεται στα γραφήματα 23 και 24.



Γράφημα 22: Χρόνος εκτέλεσης ανά αριθμό νημάτων - Sum

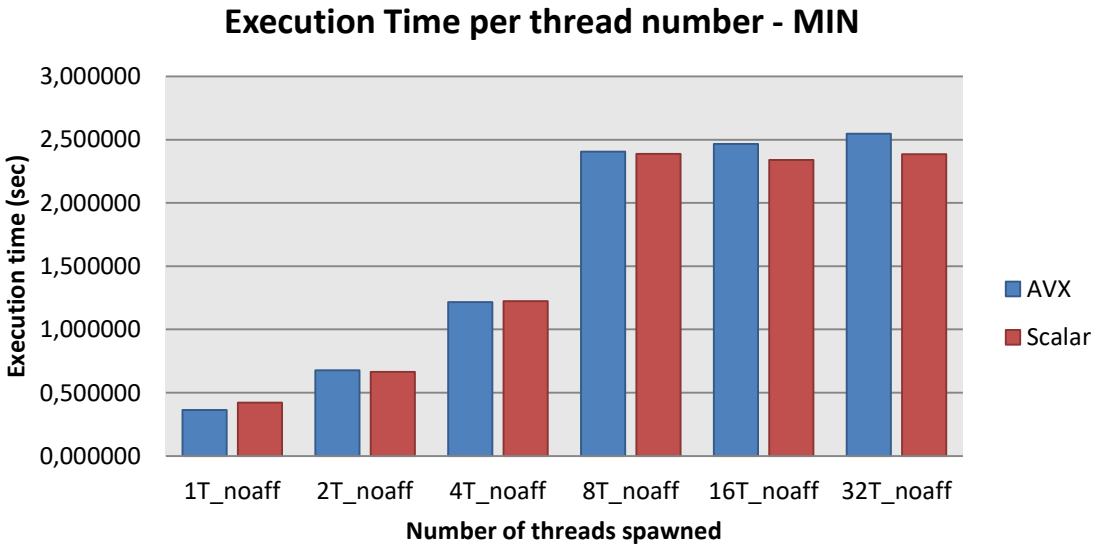


Γράφημα 23: Αριθμός cache misses ανά αριθμό νημάτων – AVX Sum

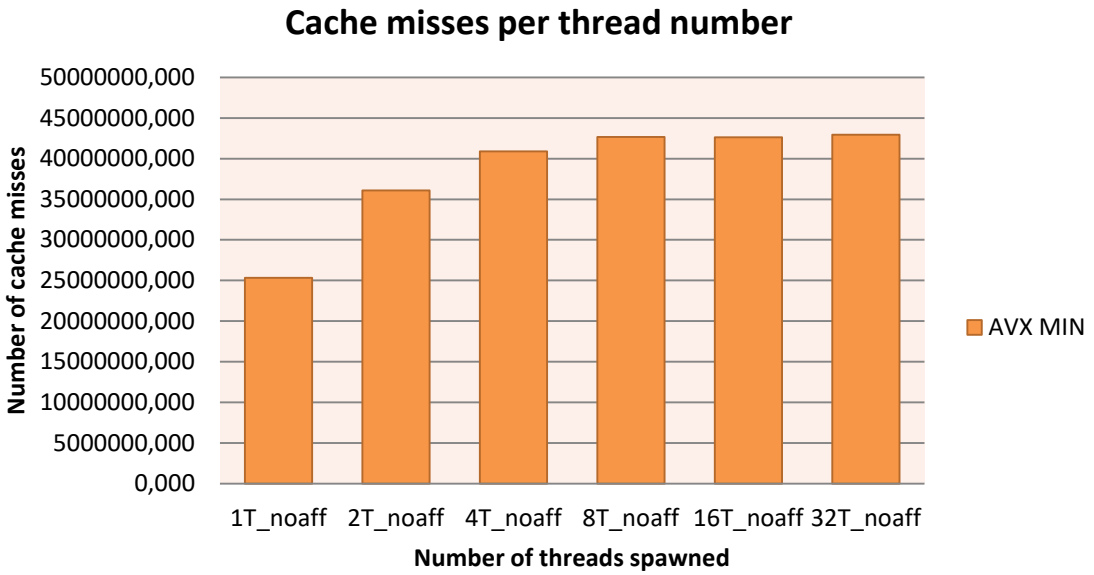


Γράφημα 24: Αριθμός cache misses ανά αριθμό νημάτων - Scalar Sum

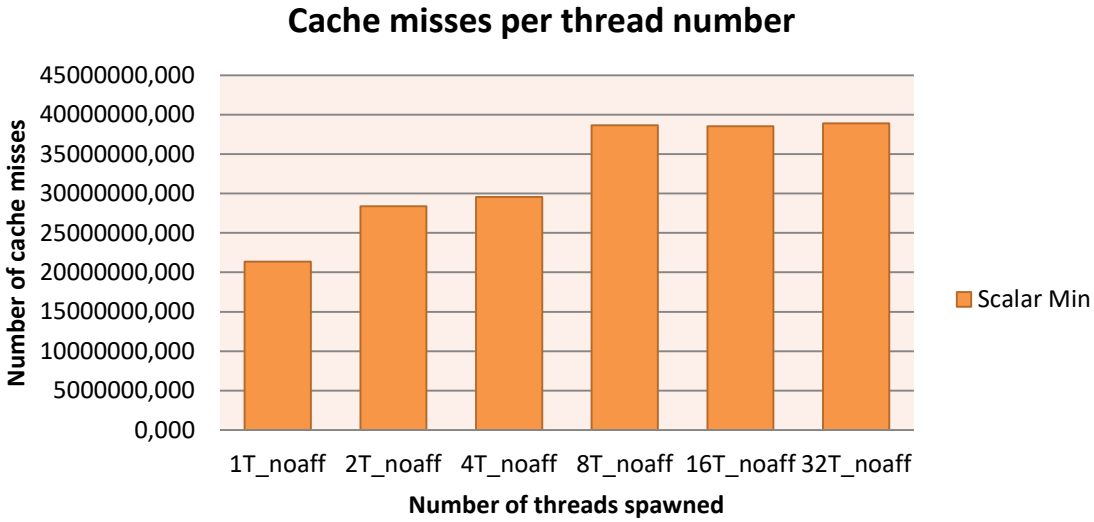
Αντίστοιχα αποτελέσματα προκύπτουν και για το Min πρόβλημα. Οι χρόνοι εκτέλεσεις φαίνονται στο γράφημα 25 και ο αριθμός των cache misses στα 26 και 27.



Γράφημα 25: Χρόνος εκτέλεσης ανά αριθμό νημάτων - Min

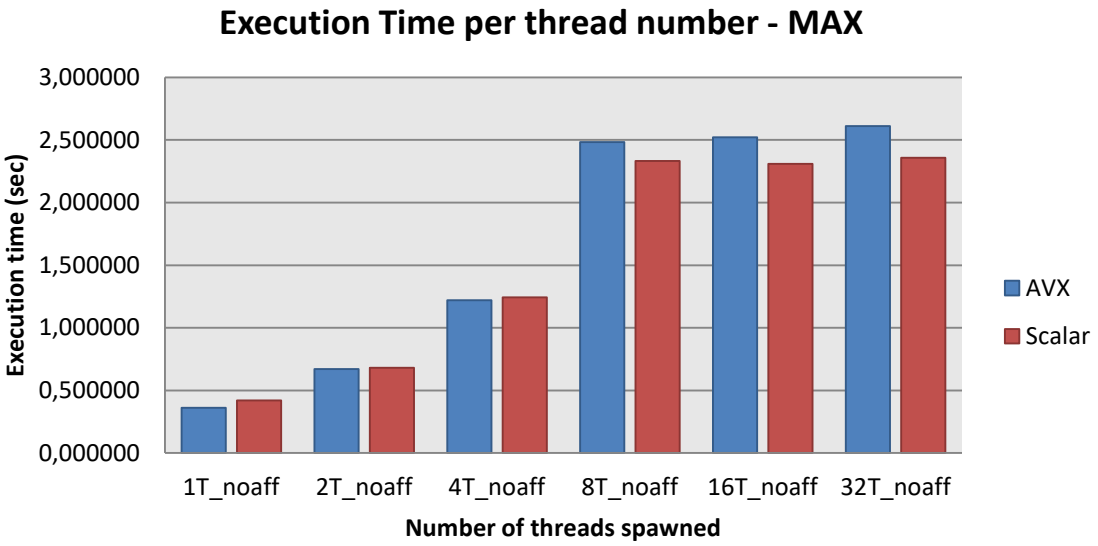


Γράφημα 26: Αριθμός cache misses ανά αριθμό νημάτων - AVX Min

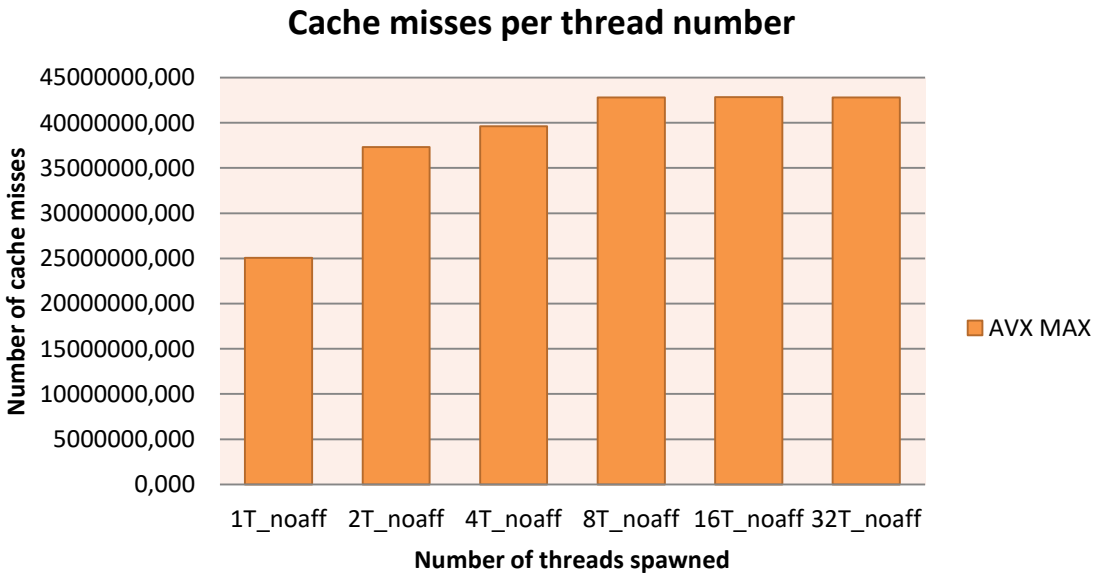


Γράφημα 27: Αριθμός cache misses ανά αριθμό νημάτων - Scalar Min

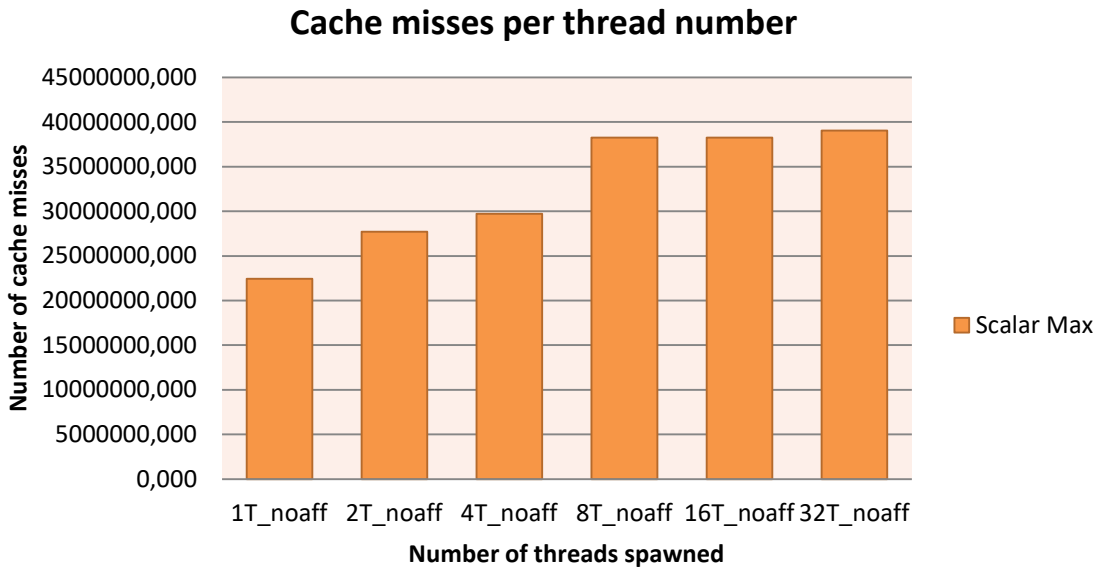
Τέλος, τα αποτελέσματα του δυϊκώς αντιστρόφου προβλήματος του Min, δηλαδή το Max, παρουσιάζονται στα γραφήματα 28 έως και 30.



Γράφημα 28: Χρόνος εκτέλεσης ανά αριθμό νημάτων - Max



Γράφημα 29: Αριθμός cache misses ανά αριθμό νημάτων - AVX Max



Γράφημα 30: Αριθμός cache misses ανά αριθμό νημάτων - Scalar Max

Κεφάλαιο 5 – Συμπεράσματα και μελλοντική μελέτη

Οι ετερογενείς αρχιτεκτονικές αποτελούν το παρόν και το μέλλον των υπολογιστικών συστημάτων, καθότι το speed-up που προσφέρουν στις εφαρμογές κάθε άλλο παρά αμελητέο μπορεί να θεωρηθεί. Τόσο η «δύναμη» της SIMD αρχιτεκτονικής, μέσω του προτύπου AVX της Intel, όσο και αυτή της SIMT αρχιτεκτονικής, μέσω του προτύπου CUDA της NVIDIA, έγινε φανερή μέσα από την παρούσα διπλωματική. Αμφότερες έχουν θετικά και αρνητικά χαρακτηριστικά και η απόδοσή τους εξαρτάται από την εκάστοτε εφαρμογή. Η συγγραφή κώδικα με χρήση AVX εντολών βελτιώνει την απόδοση της εκτέλεσης του προγράμματος και αν για κάποια κομμάτια κώδικα χρησιμοποιηθεί και το μοντέλο της SIMT αρχιτεκτονικής, ούτως ώστε να προκύψει ένα αποτέλεσμα με ασύγχρονο τρόπο, τότε η απόδοση της εφαρμογής μπορεί να βελτιωθεί ακόμα περισσότερο. Όλα αυτά βέβαια με τις ανάλογες απαιτήσεις σε μνήμη.

Ως μελλοντική μελέτη προς κάθε ενδιαφερόμενο αυτού του κλάδου, συνίσταται η χρησιμοποίηση του προτύπου AVX-512, το οποίο υποστηρίζεται από τις νεότερες γενιές επεξεργαστών της Intel. Επιπλέον, οι αλγόριθμοι που χρησιμοποιήθηκαν ή και γενικότερα πάσης φύσεως αλγόριθμοι, θα μπορούσαν να δοκιμαστούν και σε άλλες ετερογενείς αρχιτεκτονικές όπως αυτή των FPGA’s. Η χρησιμοποίηση high-end hardware αντί του commodity hardware που χρησιμοποιήσαμε για τους σκοπούς της παρούσας εργασίας αποτελεί επίσης ένα πεδίο αναζήτησης. Τέλος, τελεστές όπως JOIN, GROUP BY, ORDER BY αλλά και analytic functions θα είχανε αρκετό ενδιαφέρον να υλοποιηθούν και να συγκριθούν ανά χρησιμοποιούμενη αρχιτεκτονική.

Αναφορές

[1] Wikipedia, "wikipedia.org," 2 March 2019. [Online]. Available: https://en.wikipedia.org/wiki/Heterogeneous_computing. [Accessed 11 May 2019].

[2] Wikipedia, "wikipedia.org," 1 January 2019. [Online]. Available: https://en.wikipedia.org/wiki/Advanced_Vector_Extensions. [Accessed 11 May 2019].

[3] Intel, "software.intel.com," Intel Corporation, 21 June 2011. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions. [Accessed 11 May 2019].

[4] Wikipedia, "wikipedia.org," 3 January 2019. [Online]. Available: https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads. [Accessed 11 May 2019].

[5] J. Larkin, 14 November 2016. [Online]. Available: http://www.icl.utk.edu/~luszczek/teaching/courses/fall2016/cosc462/pdf/GPU_Fundamentals.pdf. [Accessed 11 May 2019].

[6] Stackoverflow, "stackoverflow.com," 1 October 2010. [Online]. Available: https://stackoverflow.com/questions/3841877/what-is-a-bank-conflict-doing-cuda-opencl-programming. [Accessed 11 May 2019].

[7] N. Gupta, "http://cuda-programming.blogspot.com," 1 February 2013. [Online]. Available: http://cuda-programming.blogspot.com/2013/02/bank-conflicts-in-shared-memory-in-cuda.html. [Accessed 11 May 2019].

[8] Stackoverflow, "stackoverflow.com," 1 August 2012. [Online]. Available: https://stackoverflow.com/questions/11908142/clarifying-memory-transactions-in-cuda. [Accessed 9 June 2019].

[9] M. Giles, 15 February 2010. [Ηλεκτρονικό]. Available: http://people.maths.ox.ac.uk/~gilesm/old/pp10/lec2_2x2.pdf. [Πρόσβαση 11 May 2019].

[10] "el.wikipedia.org," [Online]. Available: https://el.wikipedia.org/wiki/%CE%A5%CF%80%CE%B5%CF%81%CE%BD%CE%B7%CE%BC%CE%AC%CF%84%CF%89%CF%83%CE%B7.

[11] E. Athanasaki, N. Anastopoulos, K. Kourtis, N. Koziris, "Exploring the performance limits of simultaneous multithreading for memory intensive applications".

[12] L. Carter, J. Ferrante, D. Tullsen, N. Mitchell , "ILP versus TLP on SMT," in *Proceedings of the1999 ACM/IEEE conference on supercomputing (CDROM)*, November 1999.

[13] D. Tullsen, N. Tuck, " Initial observations of the simultaneous multithreading Pentium 4 processor," in *Proceedings of the 12th international conference on parallel architectures and compilationtechniques (PACT '03)*, New Orleans, LA, September 2003.

[14] M. Harris, "developer.nvidia.com," 18 March 2010. [Online]. Available: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf. [Accessed 11 May 2019].

[15] Wikipedia, "wikipedia.org," 22 May 2019. [Online]. Available: https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle. [Accessed 29 May 2019].

[16] Stackoverflow, "stackoverflow.com," 3 May 2011. [Online]. Available: https://stackoverflow.com/questions/5736968/why-is-cuda-pinned-memory-so-fast. [Accessed 2 June 2019].

[17] M. Boyer, "University of Virginia," [Online]. Available: https://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html. [Accessed 9 June 2019].